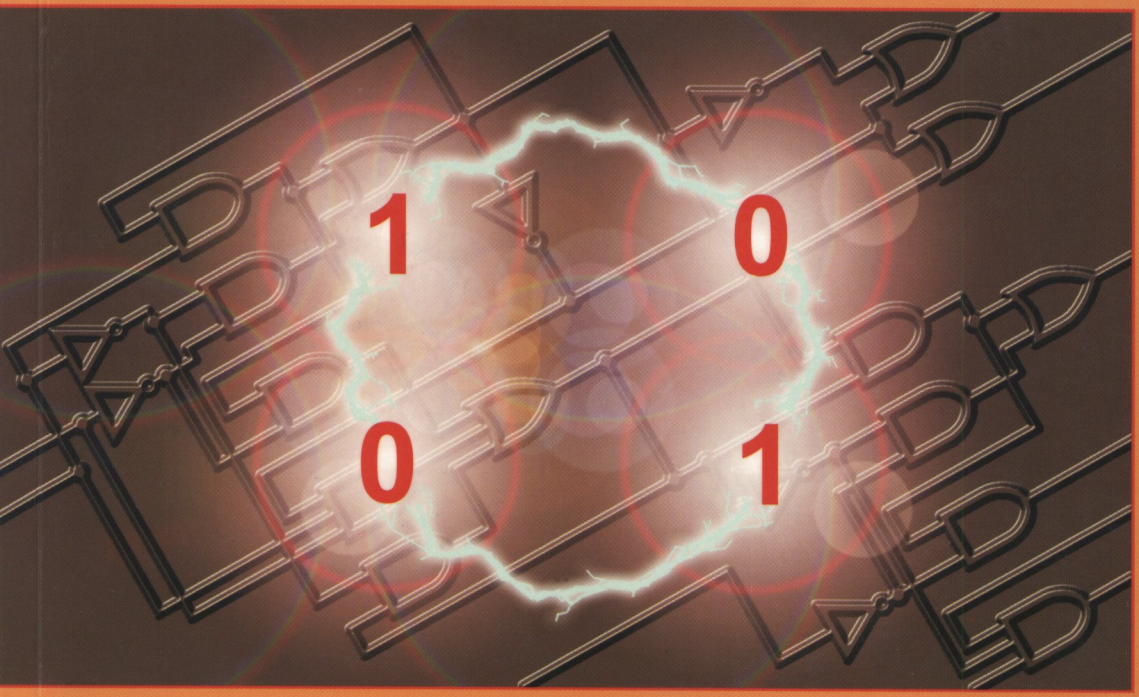
Andrzej Skorupski

PODSTAWY TECHNIKI CYFROWEJ



WKŁ

ISBN 83-206-1390-6

Technika cyfrowa stanowi bazę dla inżyniera informatyka. Znajomość jej podstaw pozwala zrozumieć działanie urządzeń cyfrowych, a w szczególności komputerów. Jest to także wiedza niezbędna dla projektantów wszelkiego rodzaju sprzętu cyfrowego. Wieloletnie doświadczenie Autora zdobyte w kraju i za granicą umożliwiło odpowiednie dobranie przedstawianych zagadnień oraz prosty sposób ich prezentacji.

Książka będzie pomocna studentom wszystkich kierunków informatycznych uczelni technicznych. Może być także wykorzystywana przez studentów innych uczelni oraz przez osoby pragnące doszkolić się w zakresie informatyki.

ISBN 83-206-1390-6
9 788320 613902

Wydawnictwa
Komunikacji i Łączności

PODSTAWY
TECHNIKI CYFROWEJ

Andrzej Skorupski

PODSTAWY TECHNIKI CYFROWEJ

Warszawa

Wydawnictwa Komunikacji i Łączności

Projekt okładki:
Cezary Stępień

Redaktor merytoryczny:
Elżbieta Gawin

Redaktor techniczny:
Jadwiga Majewska

Przygotowanie do druku:
ARKATRONIC SC

681.32-181.48
W książce omówiono kody liczbowe oraz podstawowe 4 działania w arytmetyce dwójkowej. Przedstawiono metodykę projektowania układów logicznych, zarówno kombinacyjnych jak i sekwencyjnych. Zaprezentowano typowe kombinacyjne bloki logiczne jak dekodery, multipleksery, sumatory i komparatory oraz bloki sekwencyjne, a mianowicie rejestry i liczniki. Omówiono także typową strukturę złożonych układów logicznych, tj strukturę mikroprogramowaną.

Książka może być pomocna studentom wszystkich kierunków informatycznych uczelni technicznych. Może być także używana przez studentów innych uczelni oraz przez osoby pragnące doszkolić się w zakresie informatyki.

ISBN 83-206-1390-6

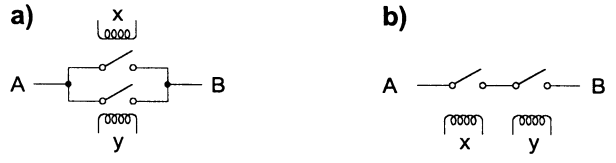
© Copyright by Wydawnictwa Komunikacji i Łączności sp. z o.o.
Warszawa 2001.

Wydawnictwa Komunikacji i Łączności sp. z o.o.
ul. Kazimierzowska 52, 02-546 Warszawa
tel. (0-22) 849-27-51, fax (0-22) 849-23-22
Dział handlowy tel. (0-22) 849-27-51 w. 555
tel./fax (0-22) 849-23-45
Prowadzimy sprzedaż wysyłkową książek
Księgarnia firmowa w siedzibie wydawnictwa
tel. (0-22) 849-20-32, czynna pon.–pt. 10^{00}–18^{00}
e-mail wkl@wkl.com.pl
WKŁ w sieci Internet http://www.wkl.com.pl
Wydanie 1. Warszawa 2001.

Spis treści

1 Wprowadzenie

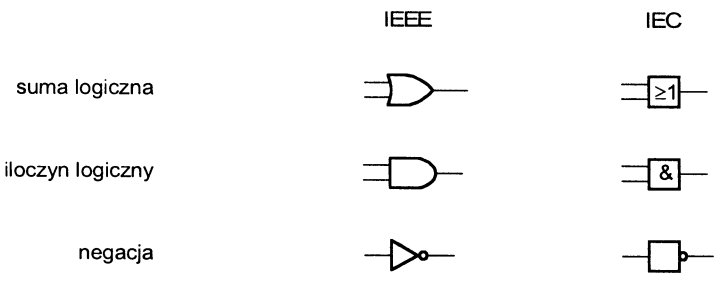
Budowanie układów cyfrowych ma swoją długą historię. Można uznać, że sięga ona nawet czasów starożytnych. Zdalne przekazywanie dwuwartościowych wiadomości było znane od dawna, np. jako zakrywanie i odkrywanie ogniska. Natura takich sygnałów ma charakter dyskretny, w odróżnieniu od sygnałów ciągłych, tj. analogowych. Takie dwuwartościowe sygnały dyskretne nazywane są współcześnie cyfrowymi, zerojedynkowymi albo binarnymi. Są one odporne na zakłócenia, mogą być przekazywane z dużą szybkością i niezawodnością i dlatego ich przetwarzanie stało się ważną dziedziną nauki i techniki zwaną techniką cyfrową. Układy i systemy, w których zachodzi przetwarzanie sygnałów cyfrowych nazywane są układami i systemami cyfrowymi (ang. *digital curcuits, digital systems*). Pierwsze układy cyfrowe były układami przekaźnikowymi, a ich opis i metodyka projektowania wykorzystywała tzw. algebrę Boole'a. W algebrze Boole'a są trzy działania na argumentach zerojedynkowych: suma logiczna (alternatywa zdarzeń), iloczyn logiczny (koniunkcja zdarzeń) i inwersja, czyli negacja. Za pomocą takich działań można określać różne funkcje, a biorąc zestaw przekaźników można zbudować układ cyfrowy realizujący daną funkcję. Przekaźnik działa w taki sposób, że jeśli w jego uzwojeniu płynie prąd, to jego styki są w jednym z dwóch położeń: mogą być zwarte albo mogą być rozwarte. Na rysunku 1.1 pokazano jak mając dwa przekaźniki można zrealizować sumę i iloczyn logiczny. Połączenie



Rysunek 1.1. Układy przekaźnikowe: a) suma logiczna i b) iloczyn logiczny

punktu A i B na rysunku 1.1a następuje, gdy styk choć jednego z dwóch przekaźników jest zwarty, a więc gdy spełniona jest suma logiczna sygnałów x i y. Połączenie punku A i B na rysunku 1.1b następuje gdy jednocześnie oba styki są zwarte, a więc gdy spełniony jest iloczyn logiczny sygnałów x i y. Negację zmiennej realizuje się poprzez konstrukcję przekaźnika (zestyki zwierne i rozwierne). Układy zbudowane z elementów przekaźnikowych nazywane są układami przełączającymi (ang. *switching circuits*). Budowanie złożonych układów cyfrowych nastręczało wiele trudności, ale nie zrażało to konstruktorów do ich wykorzystywania. W latach trzydziestych zbudowano komputer za pomocą przekaźników. Była to pierwsza maszyna obliczeniowa MARK I zbudowana przez Howarda Aikena.

Późniejszy rozwój elektroniki pozwolił na budowanie bezstykowych układów cyfrowych. Wykorzystywano do tego celu lampy elektronowe, tranzystory, układy magnetyczne itp. Układy cyfrowe zbudowane z takich elementów nazwano bramkami logicznymi. Na rysun-



Rysunek 1.2. Symbole niektórych bramek

ku 1.2 pokazano oznaczenia podstawowych bramek wg normy IEEE (amerykańska) oraz IEC (europejska). W niniejszej książce stosowane będą oznaczenia zgodne z IEEE, gdyż są one bardziej rozpowszechnione. Zastosowanie bramek do budowy układów cyfrowych spowodowało szybki rozwój komputerów, układów automatyki i elektroniki profesjonalnej. Ale był to zaledwie wstęp do prawdziwej rewolucji, którą było wprowadzenie do produkcji układów scalonych. W jednym niewielkim elemencie elektronicznym już w latach 60. można było pomieścić kilka bramek, a co ważniejsze pobierana moc była znacznie mniejsza od mocy pobieranej przez bramki budowane z elementów dyskretnych. Ale największą zaletą układów scalonych była ich cena. Te pierwsze układy scalone, tzw. małej skali integracji SSI (ang. *small scale integration*) spowodowały rozpowszechnienie techniki cyfrowej w wielu dziedzinach zastosowań. Od tej pory aż do dnia dzisiejszego trwa nieustanny rozwój technologii układów scalonych. Powstały układy średniej skali integracji MSI (ang. *medium scale integration*), a następnie wielkiej skali integracji LSI (ang. *large scale integration*). Współcześnie stosuje się układy bardzo wielkiej skali integracji VLSI (ang. *very large scale integration*). Projektanci układów cyfrowych wykonują złożone projekty wykorzystując jeden układ scalony, jak np. programowaną matrycę logiczną. Stosują przy tym bardzo zaawansowane metody komputerowego wspomagania projektowania CAD (ang. *computer aided design*).

Najpowszechniej układy cyfrowe stosowane są w komputerach i układach automatyki. Ale dotarły one już do wszystkich dziedzin techniki: od profesjonalnych układów pomiarowych, telekomunikacji, samochodów itp., aż do sprzętu powszechnego użytku jak sprzęt AGD, sprzęt multimedialny, alarmowy itp.

Znajomość układów cyfrowych jest potrzebna nie tylko projektantom, ale i użytkownikom. Obustronna znajomość podstaw tej techniki umożliwia lepszą współpracę pomiędzy tymi, którzy umieją wykorzystać swoje umiejętności inżynierskie, a tymi, którzy potrzebują rezultatów tej pracy. Dlatego niniejsza książka może być pomocna z jednej strony profesjonalistom a z drugiej strony amatorom tej wiedzy.

Celem książki jest przedstawienie jedynie logicznego aspektu projektowania układów cyfrowych. Inne aspekty, jak na przykład aspekty techniczne wymagające znajomości elektroniki nie będą tu rozwijane. Aby jednak Czytelnikowi przybliżyć praktyczną stronę problemów techniki cyfrowej przedstawiane przykłady będą opierać się na klasycznej serii układów scalonych TTL (ang. *transistor-transistor logic*). Jedną z serii tych układów oznaczono symbolami SN74xxx, gdzie xxx oznacza liczbę dwu- lub trzycyfrową. Sygnały wejściowe i wyjściowe układów TTL są reprezentowane przez dwa poziomy napięcia 0 V i +5 V. Przyjęło się jednemu poziomowi sygnału, a więc 0 V przypisać wartość zera logicznego,

a drugiemu poziomowi sygnału, a więc +5 V — wartość jedynki logicznej. Choć współcześnie nie stosuje się już układów TTL, to inne rodziny układów scalonych wykorzystują powstałe w serii TTL standardy, a między innymi struktury logiczne. Oznacza to, że większość układów scalonych TTL ma swoje odwzorowanie w nowoczesnych seriach układów scalonych, jakimi są przykładowo układy typu CMOS (ang. *complementary metal oxide silicon*). Układy te pobierają znacznie mniejszą moc niż układy TTL i dlatego umożliwiają większy stopień scalania, co predysponuje je do współczesnych zastosowań. Ponieważ jednak niniejsza książka ma charakter podstawowy, to szczegółowe zagadnienia techniczne projektowania zostaną tu pominięte.

W rozdziale drugim książki przedstawiono najczęściej stosowane kody i zapisy liczbowe oraz działania arytmetyczne na tych zapisach. W opisie tym pominięto formalizmy oraz zaawansowane metody i algorytmy, aby łatwiej można było zrozumieć prezentowane zagadnienia.

W rozdziale trzecim omówiono najprostsze układy cyfrowe, tj. układy kombinacyjne. Przedstawiono podstawowe zasady projektowania układów cyfrowych na bramkach i metody optymalizacji tych układów. Przedstawiono także wybrane bloki kombinacyjne, czyli układy kombinacyjne spełniające pewne standardowe funkcje (dekodowanie, przełączanie, komparacja i sumowanie). Bloki takie spotyka się jako układy scalone MSI. Przedstawiono także niektóre układy LSI (układy programowane) i sposoby projektowania układów cyfrowych za ich pomocą.

W rozdziale czwartym omówiono układy sekwencyjne. Pokazano metodykę projektowania układów synchronicznych i asynchronicznych. Przedstawiono elementy stosowane do ich budowy, tzw. przerzutniki. Omówiono ich rodzaje i celowość ich stosowania. Przedstawiono problemy związane z projektowaniem układów sekwencyjnych i wybrane problemy związane z ich niezawodnością. Zaprezentowano także bloki sekwencyjne: rejestry i liczniki. Na koniec pokazano sposób projektowania układu sekwencyjnego, który ma być realizowany w programowanej strukturze LSI.

Piąty rozdział książki jest poświęcony specjalnej strukturze układu cyfrowego jakim jest struktura mikroprogramowana. Przedstawiono prostą wersję takiej struktury, sposób jej wykorzystania oraz przykładowe zadanie.

Prezentowana książka może być pomocna w procesie dydaktycznym szkół zawodowych oraz w uczelniach akademickich. Książka zawiera jedynie podstawy omawianej techniki i może stanowić punkt wyjścia do dalszych studiów w tej dziedzinie. Może być także przydatna inżynierom i technikom, którzy chcą zapoznać się z tą techniką.

Autor pragnie wyrazić wdzięczność oraz podziękować Panu dr inż. Cezaremu Stępniowi za jego wielki wkład włożony w niniejszą książkę. Jego uwagi przyczyniły się do wyeliminowania wielu błędów i nadały ostateczną postać tekstowi.

Dziękuję także mgr inż. Zbigniewowi Szymańskiemu za wnikliwe przeczytanie całego tekstu i naniesienie poprawek.

Proszę Czytelników o przesyłanie wszelkich uwag dotyczących książki pod adres: ask@ii.pw.edu.pl.

Kodowanie liczb i arytmetyka 2

2.1. Kody dwójkowe

Każdą liczbę można przedstawić w różnych systemach. Najpowszechniej używa się systemu dziesiętnego. W tym systemie liczbę x przedstawia się za pomocą słowa A składającego się z n cyfr dziesiętnych (0,1,...,9) zgodnie ze wzorem

$$x = L(A) = \sum_{i=0}^{n-1} a_i 10^i$$

Na przykład wartość trzycyfrowej liczby 127 ($a_2 = 1$, $a_1 = 2$, $a_0 = 7$) oblicza się jako

$$L(127) = 1\times10^2 + 2\times10^1 + 7\times10^0$$

Mówimy wtedy o reprezentacji liczb w systemie dziesiętnym lub inaczej w systemie o podstawie 10, co zapisać można jako 127_{10}. System dziesiętny jest systemem pozycyjnym, gdzie cyfra stojąca najbardziej po lewej stronie ma wagę największą, a cyfra stojąca najbardziej po prawej stronie ma wagę najmniejszą. Podstawa systemu równa 10 oznacza, że wszystkie wagi są potęgami dziesiątki. Ale stosowane są systemy o innej podstawie. Na przykład w systemie oktalnym (o podstawie 8) wartość liczbowa słowa 127_8 wynosi:

$$L(127) = 1\times8^2 + 2\times8^1 + 7\times8^0$$

Można zapisać, że $127_8 = 87_{10}$.

W komputerach używa się systemów o podstawie 2, czyli tzw. systemów dwójkowych. Liczby naturalne można reprezentować za pomocą słów binarnych, a więc takich, które składają się z cyfr należących do zbioru {0, 1}. Przyjęło się nazywać cyfry takiego zbioru **bitami**. Jeśli dane jest n-bitowe słowo A, to wartość liczbową tego słowa określamy za pomocą wzoru

$$L(A) = \sum_{i=0}^{n-1} a_i 2^i, \quad a_i \in \{0,1\}$$

Takie przyporządkowanie liczb słowom będziemy nazywać **naturalnym kodem binarnym NKB** (ang. *natural binary code* — NBC). Konwersję 6-bitowej liczby 100011 na postać dziesiętną można wykonać według wzoru:

$$L(A) = 1\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 35_{10}$$

Algorytm konwersji odwrotnej składa się z pewnej liczby kroków, która nie jest znana z góry. Liczba tych kroków jest równa liczbie bitów poszukiwanej liczby. W pierwszym kroku algorytmu dzieli się daną liczbę przez 2. Jeśli iloraz jest całkowity, to najmniej znaczący bit a_0 jest równy 0. Jeśli iloraz nie jest całkowity, to najmniej znaczący bit a_0 jest równy 1, a jako wynik dzielenia przyjmuje się część całkowitą ilorazu. W każdym następnym

kroku algorytmu, wynik z poprzedniego kroku dzieli się przez 2 i znajduje się wartość odpowiedniego bitu a_i. I tak w każdym kroku, gdy wynik dzielenia jest całkowity, to a_i równa się 0, a gdy wynik dzielenia nie jest całkowity, to a_i równa się 1. Ponadto, w każdym kroku, jako wynik dzielenia przyjmuje się część całkowitą ilorazu odrzucając część ułamkową. Algorytm kończy się w kroku, w którym wynik dzielenia będzie równy 0.

Przykład 2.1. Znaleźć reprezentację liczby 89_{10} w naturalnym kodzie binarnym.

Rozwiązanie. Kolejne kroki algorytmu:

krok 1) $89/2 = 44\frac{1}{2}$ — 1
krok 2) $44/2 = 22$ — 0
krok 3) $22/2 = 11$ — 0
krok 4) $11/2 = 5\frac{1}{2}$ — 1
krok 5) $5/2 = 2\frac{1}{2}$ — 1
krok 6) $2/2 = 1$ — 0
krok 7) $1/2 = \frac{1}{2}$ — 1

Po siódmym kroku otrzymano wynik dzielenia równy 0, a więc liczbę 89_{10} można przedstawić jako 7-bitową liczbę 1011001. Najbardziej znaczący bit a_6 otrzymano w ostatnim kroku algorytmu. Aby dokonać sprawdzenia wyniku należy obliczyć:

$$L(1011001) = 1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = 64+16+8+1 = 89_{10}$$ ⊠*)

W zaprezentowanych systemach pozycyjnych posługujemy się skończonym, niepustym zbiorem symboli zwanych cyframi. W systemie dziesiętnym jest ich 10, a w systemie dwójkowym 2. Uporządkowany zbiór cyfr tworzy słowa. Długością słowa nazywa się liczbę cyfr w słowie. Przyporządkowanie słowom wartości liczbowych nazywa się kodowaniem, a kodem liczbowym nazywa się sposób w jaki słowom przyporządkowuje się liczby. Zbiór wszystkich słów dwójkowych o długości 10 może reprezentować wszystkie liczby całkowite od 0 do 1023. Mówimy, że każdą z tych liczb można zapisać 10-bitowym słowem w kodzie NKB. Stosuje się także inne kody dwójkowe. Przedstawione zostaną teraz dwa z nich: **kod dwójkowo-dziesiętny BCD** (ang. *binary coded decimal*) i **kod minus-dwójkowy** o historycznym znaczeniu. Kodowanie BCD polega na tym, że każda cyfra liczby zapisanej w systemie dziesiętnym jest przedstawiana za pomocą grupy czterech cyfr binarnych zwanych **tetradą** (ang. *nibble*). Przykładową liczbę 89_{10} można przedstawić za pomocą dwóch tetrad: 1000 1001. Pierwsza z nich koduje dziesiętną cyfrę 8, a druga — 9. Można zauważyć, że dla przedstawienia liczby w kodzie BCD zwykle trzeba więcej bitów niż w kodzie NKB. Dla przykładowej liczby 89_{10} trzeba było 7 bitów w kodzie NKB i 8 bitów w kodzie BCD. Liczbę dziesiętną 100 przedstawia się także na 7 bitach w kodzie NKB, ale już na 12 bitach (trzech tetradach) w kodzie BCD.

Przykład 2.2. Przedstawić liczbę 127 w kodzie BCD.

Rozwiązanie. Kodując poszczególne cyfry na kolejnych tetradach otrzymamy:
0001 0010 0111 ⊠

W kodzie minus-dwójkowym można przedstawiać zarówno liczby dodatnie jak i ujemne. W tym kodzie podstawą systemu jest -2. Jeśli dane jest n-bitowe słowo A, to wartość liczbową tego słowa określamy za pomocą wzoru:

---

*) Symbol ⊠ oznacza koniec przykładu lub zestawu zadań.

$$L(A) = \sum_{i=0}^{n-1} a_i(-2)^i$$

Stąd dla każdego i parzystego waga pozycji jest dodatnia, gdyż $(-2)^i$ jest dodatnie. Natomiast dla każdego i nieparzystego waga pozycji jest ujemna, gdyż $(-2)^i$ jest ujemne. Sposób wyznaczania wartości liczbowej słowa A ilustruje przykład 2.3.

Przykład 2.3. Jaką wartość liczbową ma słowo $A = 11001$ w kodzie minus-dwójkowym.

Rozwiązanie

$L(11001) = 1\times(-2)^4 + 1\times(-2)^3 + 0\times(-2)^2 + 0\times(-2)^1 + 1\times(-2)^0 = 16_{10} - 8_{10} + 1_{10} = 9_{10}$ ⊠

Aby znaleźć słowo w kodzie minus-dwójkowym odpowiadające postaci dziesiętnej należy zmodyfikować pokazany wcześniej algorytm konwersji postaci dziesiętnej na kod NKB. Modyfikacja polega na tym, że jeśli jedynka pojawi się na pozycjach o nieparzystej potędze, to wynik dzielenia przez 2 nie zostaje zmniejszony o 1/2 ale zwiększony o tę wartość:

krok 1) $89/2 = 44\frac{1}{2}$ — 1
krok 2) $44/2 = 22$ — 0 $i = 1$,
krok 3) $22/2 = 11$ — 0 $i = 2$,
krok 4) $11/2 = 5\frac{1}{2}$ — 1 $i = 3$, zwiększenie ilorazu o 1/2
krok 5) $6/2 = 3$ — 0 $i = 4$,
krok 6) $3/2 = 1\frac{1}{2}$ — 1 $i = 5$, zwiększenie ilorazu o 1/2
krok 7) $2/2 = 1$ — 0 $i = 6$,
krok 8) $1/2 = \frac{1}{2}$ — 1 $i = 7$, zwiększenie ilorazu o 1/2
krok 9) $1/2 = \frac{1}{2}$ — 1 $i = 8$.

Zatem słowo 110101001 w kodzie minus-dwójkowym odpowiada liczbie $+89_{10}$. Dla sprawdzenia poprawności otrzymanego wyniku oblicza się: $1\times(-2)^8 + 1\times(-2)^7 + 0\times(-2)^6 +$
$+1\times(-2)^5 + 0\times(-2)^4 + 1\times(-2)^3 + 0\times(-2)^2 + 0\times(-2)^1 + 1\times(-2)^0 =$
$= 256_{10} - 128_{10} - 32_{10} - 8_{10} + 1_{10} = 89_{10}$

Zadania dla Czytelnika

1. Uzasadnić algorytm konwersji liczb dziesiętnych na kod minus-dwójkowy.
2. Znaleźć słowa kodu minus-dwójkowego będące wartością liczb −32, −109 i +127.
3. Przedstawić w kodzie dwójkowym i kodzie BCD liczby +58, +312 i +983. ⊠

2.2. Arytmetyka stałopozycyjna

2.2.1. Kody stałopozycyjne

W komputerach i innych złożonych systemach cyfrowych określa się dwa rodzaje kodowania liczb: kody stałopozycyjne i kody zmiennopozycyjne. Kody stałopozycyjne mają ustalone miejsce rozdziału części całkowitej i ułamkowej, czyli miejsce przecinka, co oznacza, że dokładność reprezentacji (odległość na osi liczbowej pomiędzy sąsiednimi liczbami reprezentowanymi słowami o danej długości) jest stała. Dla kodów zmiennopozycyjnych dokładność reprezentacji zależy od wartości wykładnika (wyjaśniono to w rozdziale 2.3). Odległość na osi liczbowej sąsiednich liczb może być zmieniana wartością wykładnika, tak

jak zmienia się różnica pomiędzy takimi samymi wartościami masy mierzonymi w dekagramach i kilogramach.

Kodowanie stałopozycyjne liczb opiera się na przedstawionych wcześniej kodach binarnych. Dla reprezentacji liczb dodatnich i ujemnych najpowszechniej stosowane są dwa kody:

— kod znak-moduł (ZM),
— kod uzupełnienia do dwóch U2.

Kod znak-moduł (ang. *sign-magnitude*) został utworzony przez dodanie do słowa z kodu NKB, na początku każdego słowa, jednego bitu reprezentującego znak liczby. Bit ten jest bitem wskazującym czy liczba jest dodatnia czy ujemna. Przyjmuje się, że gdy liczba jest ujemna, to wartość tego bitu jest równa 1, a dalsze bity reprezentują moduł liczby w kodzie NKB. Gdy liczba jest dodatnia, to wartość tego bitu jest równa 0. Zakres reprezentowanych liczb zależy od długości słowa. Za pomocą n-bitowego słowa (wraz z dodatkowym bitem znaku) można przedstawiać liczby z zakresu:

$$-(2^{n-1}-1) \le L(A) \le +(2^{n-1}-1)$$

Na przykład za pomocą słowa 8-bitowego można przedstawiać liczby od -127_{10} do $+127_{10}$. Liczba +19 będzie przedstawiana w tym kodzie jako 00010011, a liczba −19 jako 10010011. W kodzie ZM występują dwie reprezentacje zera. Dla ośmiobitowych słów jest to zarówno słowo 00000000, jak i słowo 10000000. Podwójna reprezentacja liczby 0 jest wadą tego kodu, gdyż stwarza pewne problemy przy realizacji algorytmów arytmetycznych.

Drugim często stosowanym kodem stałopozycyjnym jest kod uzupełnień do dwóch U2 (ang. *2's complement*). Stosowano także kod uzupełnień do jedności U1 (ang. *1's complement*) i zanim zostanie przedstawiony kod U2 omówiony będzie najpierw kod U1. W kodzie tym waga najbardziej znaczącej pozycji ma wagę ujemną równą sumie wszystkich innych wag. Wartość liczbową można obliczyć ze wzoru:

$$L(A) = -a_{n-1}(2^{n-1}-1) + \sum_{i=0}^{n-2} a_i 2^i$$

Stąd wynika, że w kodzie tym liczby dodatnie są reprezentowane tak jak w kodzie NKB pod warunkiem, że długość słowa będzie dostatecznie duża, aby na najbardziej znaczącej pozycji było zero. Przykładowo liczba 5_{10} nie może być reprezentowana przez trzybitowe słowo 101, ale przez co najmniej czterobitowe słowo 0101 (także przez pięciobitowe słowo 00101, sześciobitowe 000101 itd.). Natomiast liczby ujemne na najbardziej znaczącej pozycji mają jedynkę, a pozostałe bity mają przeciwne wartości niż bity słowa kodu NKB reprezentującego moduł tej liczby. Inaczej mówiąc jest to uzupełnienie do samych jedynek (odjęcie od samych jedynek) modułu przedstawianej liczby w kodzie NKB. Zakres liczb tego kodu jest taki sam jak dla kodu znak-moduł. Liczba zero ma także dwie reprezentacje: dodatnią 00000000 i ujemną 11111111. Liczba +19 może być przedstawiona w kodzie U1 na 8 pozycjach jako 00010011, a liczba −19 jako 11101100.

Dla kodu uzupełnień do dwóch U2 wartość liczbową można obliczyć ze wzoru:

$$L(A) = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Stąd wynika, że także w tym kodzie liczby dodatnie są reprezentowane tak jak w kodzie NKB i pod tym samym warunkiem, gdyż waga najbardziej znaczącej pozycji także jest ujemna. Różnica pomiędzy kodem U2 a kodem U1 polega na tym, że różna jest wartość tej

wagi. Jak wynika ze wzoru jest ona o 1 większa i dlatego dla ujemnej liczby x reprezentacja jej w kodzie U2 jest takim samym słowem jak liczby $x+1$ w kodzie U1. Przykładowa liczba +19 będzie przedstawiana jako 00010011, a liczba −19 jako 11101101. Zakres liczb reprezentowanych w tym kodzie na n pozycjach wynosi:

$$-2^{n-1} \le L(A) \le +(2^{n-1}-1)$$

Liczba zero ma tylko jedną reprezentacje 000000...00.

Celem przedstawienia liczb ułamkowych w opisanych kodach należy oddzielić n cyfr części całkowitej od m cyfr części ułamkowej przecinkiem (jako znak oddzielający będzie używana kropka — ang. *point*). Wówczas najmniej znacząca cyfra będzie miała wagę 2^{-m}. Wartość liczbową takiego słowa ułamkowego w kodzie U1 można obliczyć ze wzoru:

$$L(A) = -a_{n-1}(2^{n-1} - 2^{-m}) + \sum_{i=-m}^{n-2} a_i 2^i$$

Natomiast wartość liczbową słowa ułamkowego w kodzie U2 można obliczyć ze wzoru:

$$L(A) = -a_{n-1} 2^{n-1} + \sum_{i=-m}^{n-2} a_i 2^i$$

Przykładowo słowo 0011.0100 reprezentuje liczbę $+3.25_{10}$, a słowo 1001.0100 w kodzie U1 reprezentuje liczbę $-6\frac{11}{16}$. Natomiast słowo 1001.0100 reprezentuje w kodzie U2 liczbę −6.75.

Zadanie dla Czytelnika. Przedstawić w kodzie U1 i U2 liczby +12.625 oraz −2.5. ⊠

Oprócz omówionych już trzech kodów stosuje się także kod dwójkowo-dziesiętny, czyli kod BCD uzupełniony bitem znaku. Przykładowo liczba +19 będzie przedstawiana jako 0 0001 1001, a liczba −19 jako 1 0001 1001. Innym, stosowanym w zmiennopozycyjnej reprezentacji liczb, kodem stałopozycyjnym jest kod z przesunięciem liczb na osi liczbowej względem liczb kodu NKB o pewną ustaloną wartość. Kod ten nosi nazwę **kodu spolaryzowanego** lub kodu z obciążeniem (ang. *bias*). Tutaj przedstawiony będzie kod z przesunięciem o 2^{n-1}, czyli o połowę zakresu. W takim przypadku liczby przedstawia się w taki sposób, że zero jest reprezentowane przez n-bitowe słowo 1000...00, co odpowiada liczbie 2^{n-1} w kodzie NKB. Wartość liczbową n-bitowego słowa w kodzie spolaryzowanym można obliczyć ze wzoru:

$$L(A) = -2^{n-1} + \sum_{i=0}^{n-1} a_i 2^i$$

Na przykład liczbę +19 można przedstawić 8-bitowym słowem kodu polaryzowanego jako 10010011 (w kodzie NKB jest to $+147 = +128+19$), a liczbę −19 jako 01101101 (w kodzie NKB jest to $+109 = -128+19$). Podczas konwersji dodatniej liczby dziesiętnej na kod spolaryzowany dodaje się jedynkę na najbardziej znaczącej pozycji. Podczas konwersji ujemnej liczby dziesiętnej na kod spolaryzowany szuka się słowa kodu NKB odpowiadającego liczbie $2^{n-1}+L(A)$. Dlatego liczbę −19 na 8 pozycjach można otrzymać biorąc $+128+(-19)=+109$. Zakres liczb kodu spolaryzowanego wynosi:

$$-2^{n-1} \le L(A) \le +(2^{n-1}-1)$$

Liczba zero ma tylko jedną reprezentacje 1000...000.

W tablicy 2.1 przedstawiono 9-bitowe liczby kodów ZM, U1, U2 i spolaryzowanego oraz 13-bitowe liczby kodu BCD.

Tablica 2.1. Reprezentacja liczb w różnych zapisach

Liczba	ZM	U1	U2	BIAS	BCD
−127	111111111	100000000	100000001	00000001	1000100100111
−126	111111110	100000001	100000010	00000010	1000100100110
.	.	.	.	.	.
.	.	.	.	.	.
−1	100000001	111111110	111111111	01111111	1000000000001
0	x00000000	x11111111	000000000	10000000	x000000000000
1	000000001	000000001	000000001	10000001	0000000000001
2	000000010	000000010	000000010	10000010	0000000000010
3	000000011	000000011	000000011	10000011	0000000000011
4	000000100	000000100	000000100	10000100	0000000000100
.	.	.	.	.	.
.	.	.	.	.	.
+126	011111110	011111110	011111110	11111110	0000100100110
+127	011111111	011111111	011111111	11111111	0000100100111

2.2.2. Dodawanie i odejmowanie liczb stałopozycyjnych

Liczby dodatnie w kodach U1, U2 i ZM są reprezentowane tak jak w kodzie NKB. Dodawanie takich liczb wykonuje się w identyczny sposób dla wszystkich kodów. Wartość bitu na i-tej pozycji sumy zależy nie tylko od wartości składników na i-tych pozycjach, ale i od wyników sumowania na mniej znaczących pozycjach. Inaczej mówiąc proces sumowania rozpoczyna się od najmniej znaczącej pozycji, gdzie dodawane są dwa bity (najmniej znaczące bity dwóch składników). Dodając dwa bity, które są jedynkami w wyniku otrzymuje się zero oraz tzw. przeniesienie c (ang. *carry*). Dlatego w czasie sumowania na i-tej pozycji dodawane są trzy bity: dwa bity a_i i b_i z odpowiednich pozycji składników i wchodzące na tę pozycję przeniesienie c_i z mniej znaczącej pozycji. Suma ta daje wynik y_i. Na i-tej pozycji powstaje także przeniesienie wychodzące c_{i+1}. Zasadę obliczania sumy y_i oraz przeniesienia c_{i+1} pokazano w tablicy 2.2.

Tablica 2.2. Sposób obliczania sumy dwóch liczb i przeniesienia

a_i	b_i	c_i	y_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

W następnym rozdziale jest opisany szczegółowiej sposób realizacji sumowania (por. rysunek 3.15). Przeniesienie na danej pozycji powstaje wtedy, gdy co najmniej dwa bity są jedynkami, natomiast wynik sumowania jest równy 1, gdy jest nieparzysta liczba jedynek składników.

Można zauważyć, że sumowanie dwóch liczb o określonej liczbie bitów może dać wynik, który wykracza poza zakres wartości liczbowych reprezentowanych w danym kodzie na danej liczbie bitów. Wtedy powstaje tzw. **nadmiar** (ang. *overflow*).

Przykład 2.4. Dodać dwie liczby (+25 i +45) w kodach ZM, U1 i U2 oraz w kodzie BCD.

Rozwiązanie. Założymy, że w kodach ZM, U1 i U2 będą to liczby 8-bitowe, a w kodzie BCD będą 9-bitowe (dwie tetrady i bit znaku).

	ZM, U1, U2	BCD
+25	0 0011001	0 0010 0101
+44	0 0101100	0 0100 0100
+69	0 1000101	0 0110 1001

Suma 8-bitowych przykładowych liczb w kodach ZM, U1 i U2 jest prawidłowa i na najbardziej znaczącej pozycji nie powstało przeniesienie, zatem nie powstał nadmiar i mówimy, że wynik mieści się na założonej liczbie pozycji dwójkowych. W czasie dodawania liczb w kodach ZM, U1 i U2 na czwartej pozycji (licząc od prawej strony — waga 2^3) powstało przeniesienie i przeszło aż do siódmej pozycji (waga — 2^6). Mówimy, że nastąpiła propagacja przeniesienia przez trzy pozycje. Natomiast w czasie dodawania liczb w kodzie BCD na obu tetradach wyniku pojawiły się słowa należące do kodu BCD. Słowa te stanowią wynik dodawania. ⊠

Przykład 2.5. Dodać dwie liczby (+89 i +45) w kodach ZM, U1 i U2 oraz w kodzie BCD.

Rozwiązanie. Założymy, że w kodach ZM, U1 i U2 będą to liczby 8-bitowe, a w kodzie BCD 9-bitowe.

	ZM, U1, U2	BCD
+89	0 1011001	0 1000 1001
+45	0 0101101	0 0100 0101
+134	1 0000110	1100 1110

Dodawanie 8-bitowych liczb w kodach ZM, U1 i U2 nie dało poprawnego rezultatu. W wyniku dodawania na siedmiu bitach otrzymano liczbę 6. Ale na najbardziej znaczącej pozycji jest jedynka, co świadczy o tym, że wynik jest ujemny. Wynika to z faktu, że na pozycji o wadze 2^6 powstało przeniesienie. Oznacza to, że wynik nie mieści się na założonej liczbie bitów. Rzeczywiście liczby 134 nie da się przedstawić na 7 bitach. Prawidłowe dodawanie takich liczb można przeprowadzić tylko na większej liczbie bitów. Dalej pokazano dodawanie tych samych liczb, ale na słowach 9-bitowych.

	ZM, U1, U2
+89	0 01011001
+45	0 00101101
+134	0 10000110

Nieprawidłowy wynik otrzymano także dodając binarnie słowa w kodzie BCD. Na obu tetradach pojawiły się słowa nie należące do kodu BCD. W takim przypadku proces sumowania rozszerza się o krok korekcyjny. Polega on na tym, że jeśli na i-tej tetradzie powstało przeniesienie albo wynik sumowania nie jest cyfrą dziesiętną, to do wyniku dodaje się 6 (jest to liczba nie wykorzystanych w kodzie BCD kombinacji zer i jedynek) oraz ewentualne przeniesienia powstające na $(i-1)$-ej tetradzie. Dla naszego przykładu korekcja wyniku będzie jak poniżej:

		1100	1110
		0110	0110
		0010	0100
	C =	(1)	(1)
nadmiar	C = 1	0011	0100

Otrzymany wynik nie mieści się na dwóch tetradach i powstał nadmiar. Rzeczywiście wynik dodawania liczb 89 i 45 jest przecież trzycyfrowy. ⊠

Podsumowując przedstawione rozważania można stwierdzić, że projektując układy realizujące operacje arytmetyczne należy uwzględnić zakres argumentów tych operacji oraz zakres wyniku. Ponieważ często zakłada się, że wynik operacji powinien mieć taką samą długość jak argumenty, to w czasie wykonywania operacji arytmetycznych trzeba kontrolować zakres wyniku. Przedstawione w rozdziale 3 układy arytmetyczne wykonujące działanie dodawania są wyposażone w sygnał wyjściowy sygnalizujący nadmiar. Układy te sygnalizują także inne cechy wyniku jak: znak, wynik jest równy zero, wynik zawiera parzystą liczbę jedynek itp. Bity te w układach arytmetycznych nazywane są znacznikami (ang. *flag*).

Rozpatrzmy teraz odejmowanie liczb oddzielnie dla każdego z kodów. W podawanych dalej przykładach działanie odejmowania będzie prezentowane jako dodawanie dwóch argumentów, w którym zamiast odjemnej dodaje się liczbę z przeciwnym znakiem.

W kodzie ZM algorytm dodawania dwóch liczb o przeciwnych znakach polega na:
— porównaniu modułów liczb aby jako znak wyniku przyjąć znak liczby o większym module,
— znalezieniu różnicy modułów przez odjęcie mniejszego modułu od większego modułu.

Przykładowo odejmując liczbę +10 od liczby +42 trzeba do +42 dodać −10. Moduł liczby dodatniej jest większy, więc znak wyniku będzie dodatni. Odejmując od 42 liczbę 10 otrzymamy:

$$\begin{array}{rr} 42 & 00101010 \\ \underline{-10} & \underline{00001010} \\ 32 & 00100000 \end{array}$$

a więc wynik jest prawidłowy, gdyż słowo 00100000 reprezentuje liczbę +32.

W podanym przykładzie nie wystąpiło na żadnej pozycji odejmowanie 1 od 0. Jeśli zachodzi taki przypadek, to oznacza on potrzebę pobrania pożyczki z bardziej znaczącej pozycji. Pożyczając jedynkę z $(i+1)$ pozycji na i-tą pozycję należy uwzględnić fakt, że na tej pozycji ma ona wagę dwukrotnie większą. Dlatego na i-tej pozycji wykonuje się odejmowanie $2-1$ i w wyniku tej operacji otrzymuje się 1. Odejmując od dwójkowej liczby A liczbę B mamy, że na i-tej pozycji od bitu a_i odejmuje się bit b_i uwzględniając wchodzącą na tę pozycję pożyczkę p_i z mniej znaczącej pozycji. Zasadę obliczania różnicy y_i oraz pożyczki wychodzącej p_{i+1} pokazano w tablicy 2.3.

Tablica 2.3. Sposób obliczania różnicy dwóch liczb z uwzględnieniem pożyczki

a_i	b_i	p_i	y_i	p_i+1
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Dalej zilustrowano zaciąganie pożyczki przez kilka pozycji na przykładzie odejmowania liczby 2 od 16:

$$\begin{array}{r} 10000 \\ \underline{-\,00010} \\ 01110 \end{array} \quad = \quad \begin{array}{r} 01120 \\ \underline{-\,00010} \\ 01110 \end{array}$$

W tym przykładzie odejmowanie jedynki od zera wystąpiło na pozycji drugiej od prawej. Trzeba zatem wziąć pożyczkę z pozycji trzeciej od prawej. Przyjmuje ona wartość 2 na pozycji drugiej od prawej. Ale ponieważ na trzeciej pozycji jest 0, to trzeba wziąć pożyczkę z czwartej pozycji. Ale i tam jest 0, więc trzeba wziąć pożyczkę z piątej pozycji. Dopiero tam jest jedynka. Zatem pierwotna odjemna $10000 = 1\times2^4 = 16_{10}$ została zamieniona na $01120 = 1\times2^3 + 1\times2^2 + 2\times2^1 = 16_{10}$.

Z podanych rozważań można wysnuć wniosek, że inaczej realizuje się odejmowanie w kodzie ZM (por. tabl. 2.2) niż dodawanie w tym kodzie (por. tabl. 2.3). Jest to wada kodu ZM, która powoduje, że jest on stosowany stosunkowo rzadko.

Przykład 2.6. Dodać w kodzie ZM liczby −42 i −22.

Rozwiązanie. Porównanie modułów prowadzi do przyjęcia dodatniego znaku wyniku. Odejmując od 42 liczbę 22 otrzymamy:

$$\begin{array}{rl} 42 & 00101010 \\ \underline{-22} & \underline{00010110} \\ 20 & 00010100 \end{array} \quad \boxtimes$$

Przedstawiona wada kodu ZM nie występuje w kodach uzupełnieniowych. Działania odejmowania i dodawania liczb w tych kodach można wykonywać w tym samym układzie. Ze wzoru na wartość liczby $L(A)$ w kodzie U1 mamy, że

$$L(A) = -a_{n-1}(2^{n-1}-1) + \sum_{i=0}^{n-2} a_i 2^i$$

Stąd dla liczb ujemnych ($a_{n-1} = 1$) pozostałe bity, czyli $(n-1)$-bitowe słowo odpowiada liczbie w kodzie NKB, którą można obliczyć ze wzoru:

$$\sum_{i=0}^{n-2} a_i 2^i = (2^{n-1}-1) + L(A)$$

Dla liczb dodatnich ($a_{n-1} = 0$) pozostałe bity czyli $(n-1)$-bitowe słowo odpowiada wartości $L(A)$ w kodzie NKB. Dodając do ujemnej liczby A dodatnią liczbę B otrzymamy, że ich suma wynosi $(2^{n-1}-1) + L(A) + L(B)$.

Ponieważ zakres liczb w kodzie U1 wynosi $-(2^{n-1}-1) \le L(A) \le +(2^{n-1}-1)$, to aby nie powstało przeniesienie na najbardziej znaczącą pozycję (pozycję znaku) moduł liczby dodatniej B musi być mniejszy od modułu liczby ujemnej. Wynik wtedy jest ujemny i jedynka na najbardziej znaczącej pozycji pozostaje. W przeciwnym razie dodanie przeniesienia na najbardziej znaczącej pozycji zmieni znak wyniku na 0 (wynik ma być dodatni). Wartość liczbowa wyniku będzie równa $L(A) + L(B) - 1$, a więc nieprawidłowa. W takim przypadku należy wykonać krok korekcyjny i dodać do wyniku 1.

Rozpatrzmy algorytm operacji odejmowania w kodzie U1 na przykładzie liczb −22 i +6. Przedstawiając liczbę −22 na 8 bitach bierzemy $(2^{n-1}-1) + L(A) = 127 - 22 = 105$, co w kodzie NKB jest słowem 11101001. Dodając słowo 00000110 (+6) otrzymamy:

$$\begin{array}{rr} 11101001 & (-22) \\ \underline{00000110} & \underline{(+6)} \\ 11101111 & (-16) \end{array}$$

W tym przypadku nie powstało przeniesienie na najbardziej znaczącą pozycję i wynik jest poprawny.

Rozpatrzmy teraz algorytm odejmowania w kodzie U1 na przykładzie liczb –22 i +42:

$$\begin{array}{rrr} & 11101001 & (-22) \\ & \underline{00101010} & \underline{(+42)} \\ (1) & 00010011 & (+19) \end{array}$$

Dodając odpowiednie pozycje obu liczb otrzymamy pokazany wyżej wynik (liczba +19 w kodzie NKB) oraz bit przeniesienia. Zgodnie z tym co zostało już powiedziane należy wykonać krok korekcyjny, a więc wynik zmodyfikować przez dodanie jedynki (przeniesienia) do najmniej znaczącej pozycji wyniku.

$$\begin{array}{rr} 11101001 & (-22) \\ \underline{00101010} & \underline{(+42)} \\ 00010011 & (+19) \\ \underline{\qquad\quad 1} & \\ 00010100 & \end{array}$$

Jak łatwo zauważyć, po tej modyfikacji wynik jest poprawny (jest to liczba +20 w kodzie NKB). Sprawdźmy także działanie przy zamienionej kolejności argumentów.

$$\begin{array}{rrr} & 00101010 & (+42) \\ & \underline{11101001} & \underline{(-22)} \\ (1) & 00010011 & (+19) \\ & \underline{\qquad\quad 1} & \\ & 00010100 & \end{array}$$

Zamiana kolejności argumentów nie zmienia poprawności wyniku.

Korekcja wyniku w przypadkach powstania przeniesienia na najbardziej znaczącą pozycję jest poważną wadą kodu U1, gdyż wydłuża czas dodawania. W krańcowym przypadku, gdy podczas dodawania jedynki przeniesienie będzie propagować przez wszystkie pozycje, to czas dodawania zwiększy się dwukrotnie. Wady tej nie ma kod U2, co zostanie pokazane w dalszym ciągu tego rozdziału.

Rozpatrzmy jeszcze na koniec dodawanie dwóch liczb ujemnych. W tym przypadku na najbardziej znaczących pozycjach obu liczb są jedynki i dlatego na pewno powstanie przeniesienie. Aby nie przekroczyć dopuszczalnego zakresu suma pozostałych bitów musi być mniejsza od $(2^{n-1}-1)$. W przeciwnym przypadku nie powstanie przeniesienie na najbardziej znaczącą pozycję i dodając dwie jedynki znaku otrzyma się w wyniku zero, a więc liczbę dodatnią. Rozpatrzmy przykład dodawania liczb –22 i –42.

$$\begin{array}{ll} 11101001 & (-22) \\ \underline{11010101} & (\underline{-42}) \\ 10111110 & \\ \underline{\qquad\quad 1} & \\ 10111111 & (-64) \end{array}$$

W przykładzie tym powstało przeniesienie na najbardziej znaczącą pozycję i wynik jest ujemny, a więc prawidłowy (po korekcji jest prawidłowa wartość). Rozpatrzmy przykład dodawania liczb −86 i −42.

$$\begin{array}{ll} 10101001 & (-86) \\ \underline{11010101} & (\underline{-42}) \\ 01111110 & (+126) \\ \underline{\qquad\quad 1} & \\ 01111111 & (+127) \end{array}$$

W tym przykładzie tym nie powstało przeniesienie na najbardziej znaczącą pozycję i otrzymaliśmy wynik dodatni. Krok korekcyjny nie zmienił znaku. Można zauważyć, że gdyby wynik sumowania był o 1 większy, to wynik byłby poprawny. Rozpatrzmy przykład dodawania liczb −85 i −42.

$$\begin{array}{ll} 10101010 & (-85) \\ \underline{11010101} & (\underline{-42}) \\ 01111111 & (+127) \\ \underline{\qquad\quad 1} & \\ 10000000 & (-127) \end{array}$$

W przypadku wynik jest poprawny.

Konieczność wykonywania kroku korekcyjnego, jeśli na najbardziej znaczącej pozycji powstaje przeniesienie, jest wadą zapisu U1. Zapis U2 nie ma tej wady. Ze wzoru na wartość liczby $L(A)$ w kodzie U2 mamy, że

$$L(A) = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Stąd dla liczb ujemnych ($a_{n-1} = 1$) pozostałe bity, czyli $(n-1)$-bitowe słowo odpowiada w kodzie NKB liczbie:

$$\sum_{i=0}^{n-2} a_i 2^i = 2^{n-1} + L(A)$$

Dla liczb dodatnich ($a_{n-1} = 0$) pozostałe bity, czyli $(n-1)$-bitowe słowo odpowiada liczbie $L(A)$ w kodzie NKB.

Dodając do ujemnej liczby A dodatnią liczbę B otrzymamy, że ich suma wynosi:

$$2^{n-1} + L(A) + L(B)$$

a więc dodawanie liczb z przeciwnym znakiem w kodzie U2 daje poprawny wynik bez kroku korekcyjnego.

Ponieważ zakres liczb w kodzie U2 wynosi $-2^{n-1} \leq L(A) \leq +2^{n-1}$, to dodawanie liczb z przeciwnym znakiem nie może spowodować nadmiaru, czyli przekroczenia zakresu.

Zaobserwujmy to na przykładzie:

(+42)	00101010	(−42)	11010110
(−22)	11101010	(+22)	00010110
(+20)	00010100	(−20)	11101100

Natomiast dodając dwie liczby ujemne można przekroczyć dopuszczalny zakres i wtedy na najbardziej znaczącej pozycji powstanie zero, co oznacza, że wynik jest dodatni. Na przykład:

(−42)	11010110
(−122)	10000110
(+92)	01011100

W tym przypadku podczas dodawania dwóch liczb ujemnych powstała liczba dodatnia. Wskazuje to na powstanie nadmiaru. Gdyby oba argumenty były reprezentowane przez 9-bitowe słowo, to powstałoby przeniesienie na najbardziej znaczącą pozycję i dodając trzy jedynki otrzymalibyśmy w wyniku jedynkę na pozycji znaku i suma byłaby ujemna (otrzymalibyśmy $-256+92=-164$):

(−42)	111010110
(−122)	110000110
(−164)	101011100

Układy realizujące opisane działania muszą być wyposażone w sygnalizację ewentualnego przekroczenia zakresu wyniku, tj. nadmiaru. Nadmiar może powstać przy dodawaniu liczb z tym samym znakiem, tzn. gdy oba argumenty są bądź dodatnie bądź ujemne. Aby wykryć powstanie nadmiaru trzeba porównać znak wyniku ze znakami argumentów.

Zadania dla Czytelnika

1. Dodać 8-bitowe liczby w kodzie U1 i U2: −23 i +44, −127 i +99, 0 i +77.
2. Dodać w kodzie BCD liczby: +37 i +26, +49 i +55. ⊠

2.2.3. Mnożenie

Podobnie jak w przypadku systemu dziesiętnego tak i w systemie dwójkowym liczby mnoży się w tylu krokach z ilu bitów składają się czynniki. Jeśli mnoży się dwie trzycyfrowe liczby w systemie dziesiętnym, to wykonuje się to w trzech krokach (mnożenie przez każdą pozycję). Wynik może być wówczas sześciocyfrowy. Podobnie jeśli mnożymy dwie liczby ośmiobitowe w systemie dwójkowym, to mnożenie wykonuje się w 8 krokach, a wynik może być 16-bitowy. Mnożenie liczb dziesiętnych jakie wykonujemy „na kartce" polega na tym, że w jednym kroku mnożymy jedną cyfrę mnożnika przez mnożną, a iloczyn, zwany cząstkowym, wpisujemy z odpowiednim przesunięciem. Na końcu algorytmu sumujemy wszystkie iloczyny cząstkowe. Ponieważ w urządzeniach cyfrowych stosuje się zwykle sumatory dwuargumentowe, to algorytm mnożenia musi zostać tak zmodyfikowany, aby już w każdym kroku wykonywać sumowanie iloczynów cząstkowych, a nie sumowanie wielu składników na końcu algorytmu. Dlatego w każdym kroku algorytmu mnożenia liczb dwójkowych

wykonywane są dwie operacje: dodawania i przesunięcia iloczynu cząstkowego o jedną pozycję w prawo. Jednym ze składników dodawania jest zawsze iloczyn cząstkowy, a drugi składnik zależy od aktualnej wartości najmniej znaczącego bitu mnożnika LSb (ang. *last significant bit*). Jeśli jest on równy 1, to składnik jest mnożną, a jeśli jest on równy 0, to składnik jest też zerem (słowem dwójkowym o danej długości reprezentującym liczbę 0). Dalej pokazano przykład mnożenia dwóch słów w kodzie NKB i kolejne słowa reprezentujące iloczyny cząstkowe. Ponieważ przykładowe czynniki są czterobitowe (+5 i +6), to iloczyny cząstkowe są ośmiobitowe. Algorytm rozpoczyna się od zapisania mnożnika na młodszych bitach ośmiobitowego słowa (druk wytłuszczony). Następnie wykonywane są cztery kroki składające się z dodawania i przesunięcia. W zależności od wartości bitu na najmniej znaczącej pozycji ośmiobitowego słowa, do 4 najbardziej znaczących bitów dodawane jest albo 0 albo mnożna. Jeśli wartość tego bitu jest 0, to dodaje się 0, a jeśli jest 1, to dodaje się mnożną.

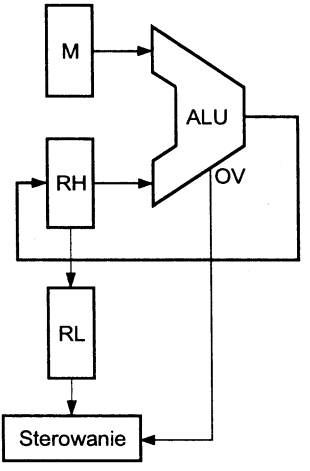
	(+5)	0101	mnożna
	(+6)	0110	mnożnik
	zapisanie argumentów	0000 **0110**	
krok 1	dodanie 0	0000 **0110**	
	przesunięcie	0000 **0011**	
krok 2	dodanie +5	0101 **0011**	
	przesunięcie	0010 **1001**	
krok 3	dodanie +5	0111 **1001**	
	przesunięcie	0011 **1100**	
krok 4	dodanie 0	0011 **1100**	
	przesunięcie	0001 1110	

Z przykładu widać jak bity mnożnika (wytłuszczone) są zastępowane bitami wyników cząstkowych, aż po czwartym kroku ośmiobitowe słowo zawiera wynik. W przykładzie jest to słowo reprezentujące wartość +30 w kodzie NKB.

Algorytm mnożenia liczb w kodzie NKB można przedstawić następująco:

1. Ustalić długość słowa wyniku (suma liczby bitów mnożnika i mnożnej) i wpisać na mniej znaczącej części mnożnik.
2. Do bardziej znaczącej części wyniku dodawać, w zależności od wartości najmniej znaczącego bitu wyniku częściowego albo mnożną (gdy LSb = 1) albo 0 (gdy LSb = 0).
3. Przesunąć wynik cząstkowy o jedną pozycję w prawo.
4. Jeśli wykonano odpowiednią liczbę kroków (liczba bitów argumentów) to algorytm kończy się, w przeciwnym wypadku powtarza się krok 2 i 3.

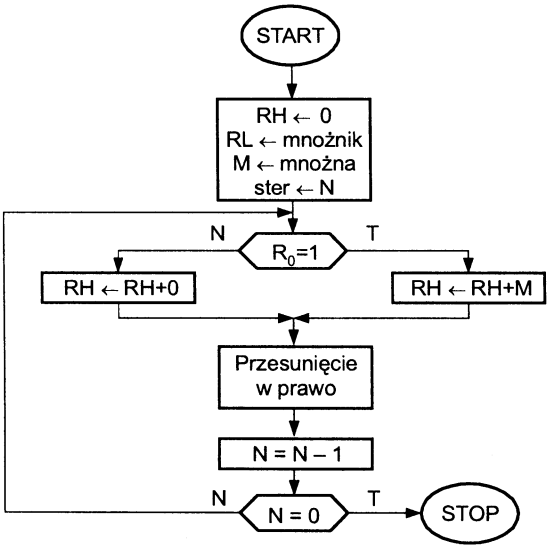
W rzeczywistych układach cyfrowych miejscem dla wyników cząstkowych jest tzw. **rejestr** (ang. *register*). Podobnie w rejestrze przechowuje się mnożną. Przedstawiony algorytm realizuje się w układzie składającym się z dwóch rejestrów (będą opisane w dalszej części książki) i z układu sumatora ALU (ang. *arithmetic-logical unit*) pokazanym na rysunku 2.1. W takim układzie jeden z rejestrów jest tzw. **rejestrem przesuwającym R**. Jest to rejestr o podwójnej długości przeznaczony do pamiętania wyników cząstkowych i składa się z dwóch części: rejestru RH pamiętającego bardziej znaczącą połówkę rejestru R i rejestru RL pamiętającego mniej znaczącą połówkę rejestru R. Mnożna jest zapisywana do rejestru M.



Rysunek 2.1. Uproszczony układ mnożenia

W pokazanym na rysunku 2.1 układzie rolą układu sterowania jest określenie liczby kroków oraz kontrola wartości najmniej znaczącego bitu rejestru RL. Ważnym zadaniem tego układu jest także badanie ewentualnego nadmiaru, który może powstać w każdym kroku dodawania. Układ sterowania jest dlatego podłączony do wyjścia układu ALU, na którym w każdym cyklu ustawiana jest wartość znacznika nadmiaru OV.

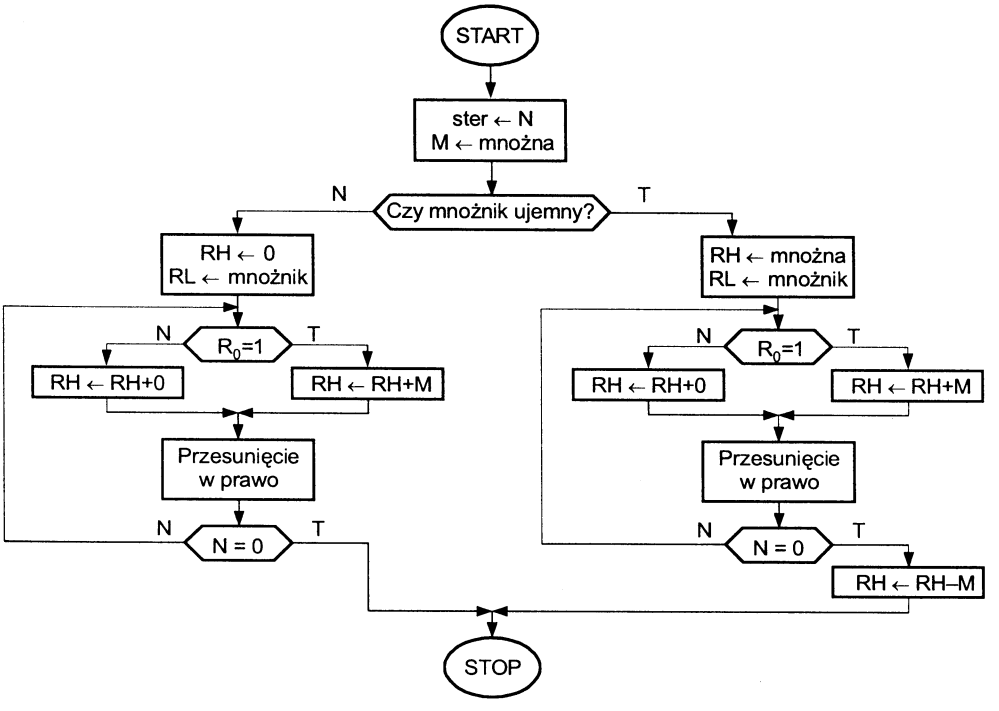
Omówiony algorytm mnożenia w kodzie NKB można przedstawić w postaci sieci działań, co pokazano na rysunku 2.2. Sieć działań nie uwzględnia przypadku powstania nadmiaru. Jeśli w czasie mnożenia wystąpi nadmiar, to inne zasoby systemu powinny decydować o dalszej akcji. Jedną z możliwości jest zwiększenie długości słowa argumentów. W innych przypadkach wystarczy sygnalizować ten fakt operatorowi.



Rysunek 2.2. Algorytm mnożenia w kodzie NKB

Po załadowaniu rejestrów R i M jest badany najmniej znaczący bit rejestru R nazwany R_0. W zależności od wartości tego bitu do rejestru RL dodawana jest mnożna (gdy $R_0 = 1$) lub zero (gdy $R_0 = 0$). Potem zawartość rejestru R jest przesuwana w prawo, a następnie licznik kroków jest dekrementowany (zmniejszany o 1) i badany jest jego aktualny stan. Jeśli wykonana liczba kroków wynosi N, to licznik został wyzerowany i następuje koniec algorytmu. Jeśli liczba wykonanych kroków jest mniejsza od N, to wraca się do badania najmniej znaczącego bitu rejestru R.

Przedstawiony algorytm mnożenia liczb w kodzie NKB można modyfikować na różne sposoby, aby dało się przystosować go do realizacji algorytmów mnożenia liczb w innych kodach. Dla kodu ZM algorytm ten należy uzupełnić możliwością wyznaczenia bitu znaku. Po wymnożeniu modułów argumentów, bit znaku jest wyznaczany w taki sposób, że jeśli znaki argumentów są takie same, to znak wyniku jest dodatni (0), natomiast jeśli znaki są różne, to znak wyniku jest ujemny (1). Jak pokazano dalej czynność ta, w układach cyfrowych, jest realizowana w sposób bardzo prosty.



Rysunek 2.3. Algorytm mnożenia w kodzie U1

Algorytmy mnożenia w kodach U1 i U2 wymagają dokładniejszego omówienia. Przypadki mnożenia dwóch liczb dodatnich pominięto, gdyż wówczas stosuje się bezpośrednio algorytm jak dla kodu NKB. Szczególną uwagę należy zwrócić na przykłady mnożenia liczb z różnymi znakami. Jak pokazano dalej, algorytmy mnożenia (w zapisach U1 i U2) różnią się w zależności od znaku mnożnika.

Rozpatrzmy najpierw algorytm mnożenia w kodzie U1 przedstawiony w postaci sieci działań na rysunku 2.3. W zależności od znaku mnożnika rejestr RH jest zapisywany w różny sposób. Jeśli znak mnożnika jest dodatni, to do rejestru RH jest zapisywane 0. Jeśli znak

mnożnika jest ujemny, to do rejestru RH jest zapisywana mnożna. Następne kroki algorytmu są takie same aż do wyzerowania licznika kroków. Wtedy jeśli mnożnik jest ujemny następuje korekcja wyniku przez odjęcie mnożnej od zawartości rejestru RH. Realizacja tego algorytmu musi uwzględniać w czasie operacji dodawania i przesuwania, że są to operacje w kodzie U1. Powoduje to konieczność wykonania kroku korekcyjnego dodawania, gdy powstanie przeniesienie na najbardziej znaczącej pozycji. Ponadto należy zwrócić uwagę na konieczność wpisywania na pozycję znaku tej samej wartości w czasie przesuwania. Jest to tzw. **przesunięcie arytmetyczne** (ang. *aritmetic shift*) co oznacza, że na najbardziej znaczącą pozycję rejestru zostaje wpisana jego poprzednia wartość. Wynika to z konieczności zachowania znaku liczby. Należy pamiętać także o dwóch reprezentacjach 0 w tym kodzie. Algorytm ten zostanie zilustrowany kilkoma przykładami.

Przykład 2.7. Pomnożyć dwie liczby w kodzie U1: +2 przez −3.

Rozwiązanie. Przyjmijmy najpierw, że mnożnik jest dodatni (+2) a mnożna ujemna (−3). Wtedy algorytm jest podobny do algorytmu mnożenia w kodzie NKB. Różnica polega na tym, że w kodzie U1 ładowanie rejestru RH jest różne w zależności od znaku mnożnej. Jeśli mnożna jest dodatnia (mnożenie dwóch liczb dodatnich jest wtedy identyczne jak w kodzie NKB), to rejestr RH ładuje się samymi zerami (w kodzie U1 jest to +0). Natomiast gdy mnożna jest ujemna, to rejestr RH ładuje się samymi jedynkami (w kodzie U1 jest to −0).

		mnożnik (+2)	mnożna (−3)
		1111 0010	1100
krok 1	dodanie 0	1111	
	przesunięcie	1111 1001	
krok 2	dodanie −3	1100	
	przesunięcie	1110 0100	
krok 3	dodanie 0	1110	
	przesunięcie	1111 0010	
krok 4	dodanie 0	1111	
	przesunięcie	1111 1001	(wynik −6) ⊠

Przykład 2.8. Pomnożyć w kodzie U1 liczby +6 (mnożnik) i −5 (mnożna).

Rozwiązanie

		mnożnik (+6)	mnożna (−5)
		1111 0110	1010
krok 1	dodanie 0	1111	
	przesunięcie	1111 1011	
krok 2	dodanie −5	1001	
	korekcja dodawania +1	1010	
	przesunięcie	1101 0101	
krok 3	dodanie −5	0111	(UWAGA! — powstał nadmiar)
	korekcja dodawania +1	1000	
	przesunięcie	1100 0010	
krok 4	dodanie 0	1100	
	przesunięcie	1110 0001	(wynik −30)

Na początku algorytmu zapełniony został rejestr wyniku cząstkowego: w bardziej znaczącej części tego rejestru wpisano – 0, a w mniej znaczącą część wpisano mnożnik. W pierwszym kroku algorytmu do bardziej znaczącej części rejestru wyniku częściowego dodano 0, gdyż najmniej znaczący bit tego rejestru (najbardziej po prawej stronie) jest 0. Następnie dokonano przesunięcia zawartości tego rejestru w prawo. W drugim kroku do bardziej znaczącej części rejestru wyniku częściowego dodano mnożną, gdyż najmniej znaczący bit rejestru jest 1. Ponieważ wystąpiło przeniesienie z najbardziej znaczącej pozycji, to należy wykonać korekcję wyniku, czyli dodać jedynkę do najmniej znaczącej pozycji. Następnie należy wykonać przesunięcie arytmetyczne. W trzecim kroku do bardziej znaczącej części rejestru wyniku częściowego dodano mnożną, gdyż najmniej znaczący bit rejestru R jest 1. Jak można zauważyć zawartość rejestru R po tej operacji wynosi 0111 (+7). Jest to błąd, gdyż dodając dwie liczby ujemne nie można otrzymać liczby dodatniej. Oznacza to, że dla mnożenia takich liczb przyjęto zbyt krótkie słowo. Układ ALU wykonujący dodawanie sygnalizuje nadmiar. W trzecim kroku dodaje się do liczby –2 (1101) liczbę –5 (1010). Wynik takiego działania powinien być –7 (1000). Aby prawidłowo otrzymać taki wynik należy wziąć większą liczbę bitów, np. 5:

	11101	(–2)
	+11010	(–5)
	10111	
korekcja dodawania +1	1	
	11000	(–7)

W podanym przykładzie wykonano korekcję czterobitowego słowa, co przypadkowo dało wynik poprawny. Jak łatwo sprawdzić po czwartym kroku zawartość rejestru R odpowiada liczbie –30 w zapisie U1, więc wynik całego mnożenia jest prawidłowy. ⊠

Przykład 2.9. Pomnożyć w kodzie U1 liczby –6 (mnożnik) i –5 (mnożna).

Rozwiązanie. Ponieważ w tym przykładzie mnożnik jest ujemny, to rejestr RH jest ładowany mnożną. Uwzględniając doświadczenie z powstawaniem nadmiaru z poprzedniego przykładu wykonamy ten przykład biorąc słowa pięciobitowe.

		mnożnik (–6)	mnożna (–5)
	wpisanie mnożnej i mnożnika	11010 11001	11010
krok 1	dodanie –5	10100	
	korekcja (+1)	10101	
	przesunięcie	11010 11100	
krok 2	dodanie 0	11010	
	przesunięcie	11101 01110	
krok 3	dodanie 0	11101	
	przesunięcie	11110 10111	
krok 4	dodanie –5	11000	
	korekcja	11001	
	przesunięcie	11100 11011	

krok 5	dodanie −5	10110		
	korekcja	10111		
	przesunięcie	11011	11101	
	korekcja wyniku	11011	11101	
	odjęcie −5 (dodanie +5)	00101	00000	
		00000	11101	
	korekcja +1	00000	11110	(wynik +30) ⊠

Przykład 2.10. Pomnożyć w kodzie U1 liczby −6 (mnożnik) i +5 (mnożna).

Rozwiązanie. W tym przykładzie wykonamy mnożenie biorąc słowa czterobitowe. Ponieważ mnożnik jest ujemny, to rejestr RH, na początku algorytmu, jest ładowany mnożną.

	mnożnik (−6) mnożna (+5)			
	wpisanie mnożnej i mnożnika	0101	1001	0101
krok 1	dodanie +5	1010		
	przesunięcie	0101	0100	(na najbardziej znaczącej pozycji wpisano 0, gdyż przed przesunięciem była liczba 10_{10} — 01010)
krok 2	dodanie 0	0101		
	przesunięcie	0010	1010	
krok 3	dodanie 0	0010		
	przesunięcie	0001	0101	
krok 4	dodanie +5	0110		
	przesunięcie	0011	0010	
	korekcja wyniku	0011	0010	
	odjęcie −5	1010	1111	
		1110	0001	(wynik −30) ⊠

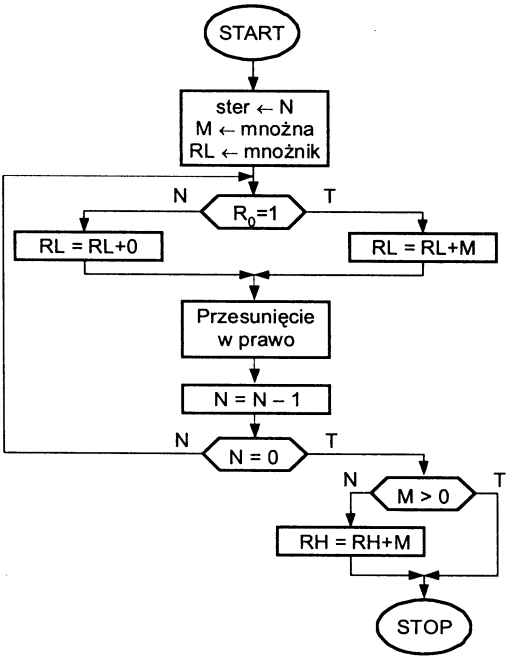
Analizując algorytm postępowania przy mnożeniu liczb w kodzie U1 należy zwrócić uwagę na:

1) początkową zawartość rejestru wyniku częściowego o podwójnej długości,
2) sposób przeprowadzenia korekcji wyniku.

W przypadku gdy mnożna była ujemna, do bardziej znaczącej części wyniku dodawano liczbę dodatnią uzupełniając mniej znaczącą część wyniku zerami. W przypadku gdy mnożna była dodatnia, do bardziej znaczącej części wyniku dodawano liczbę ujemną uzupełniając mniej znaczącą część wyniku jedynkami.

Algorytm mnożenia w kodzie U2 jest łatwiejszy od algorytmu w kodzie U1. Wynika to z faktu, że w kodzie U2 nie przeprowadza się korekcji dodawania, a po drugie nie wykonuje się wstępnego zapisywania rejestru RH w zależności od znaku mnożnika. Sieć działań algorytmu mnożenia dwóch liczb w kodzie U2 pokazano na rysunku 2.4. W tym przypadku także istnieje konieczność korekcji wyniku w zależności od znaku mnożnika.

Algorytm mnożenia liczb w kodzie U2 rozpoczyna się od wpisania mnożnika do mniej znaczącej części rejestru podwójnej długości RL. Do bardziej znaczącej części tego rejestru



Rysunek 2.4. Algorytm mnożenia w kodzie U2

RH wpisuje się zera, tak jak dla mnożenia w systemie NKB. Dalej wszystkie kroki algorytmu przebiegają jak dla mnożenia w systemie NKB, natomiast zakończenie algorytmu jest różne w zależności od znaku mnożnika. Jeśli jest on dodatni, to nie wymaga się wykonania kroku korekcyjnego. Jeśli natomiast jest on ujemny, to wykonuje się krok korekcyjny. Korekcja polega na odjęciu mnożnej od bardziej znaczącej części zawartości rejestru wyniku częściowego, czyli dodania mnożnej z przeciwnym znakiem. Rozpatrzmy to na przykładach.

Przykład 2.11. Pomnożyć w kodzie U2 liczbę +6 (mnożnik) przez liczbę –5 (mnożna).

Rozwiązanie

		mnożnik (+6)	mnożna (–5)
		0000 0110	1011
krok 1	dodanie 0	0000	
	przesunięcie	0000 0011	
krok 2	dodanie –5	1011	
	przesunięcie	1101 1001	
krok 3	dodanie –5	1000	
	przesunięcie	1100 0100	
krok 4	dodanie 0	1100	
	przesunięcie	1110 0010	(wynik –30). ⊠

Przykład 2.12. Pomnożyć liczbę −6 (mnożnik) przez liczbę −5 (mnożna).

Rozwiązanie

		mnożnik (−6)	mnożna (−5)
		0000 1010	1011
krok 1	dodanie 0	0000	
	przesunięcie	0000 0101	
krok 2	dodanie −5	1011	
	przesunięcie	1101 1010	
krok 3	dodanie 0	1101	
	przesunięcie	1110 1101	
krok 4	dodanie −5	1001	
	przesunięcie	1100 1110	
	korekcja wyniku	1100	
	odjęcie −5 (dodanie +5)	0101	
		0001 1110	(wynik +30)

W tym przypadku algorytm mnożenia wymaga kroku korekcyjnego. Korekcja polega na odjęciu mnożnej od bardziej znaczącej części zawartości rejestru wyniku częściowego, czyli dodania jej z przeciwnym znakiem. ⊠

Przykład 2.13. Pomnożyć liczbę −6 (mnożnik) przez liczbę +5 (mnożna).

Rozwiązanie

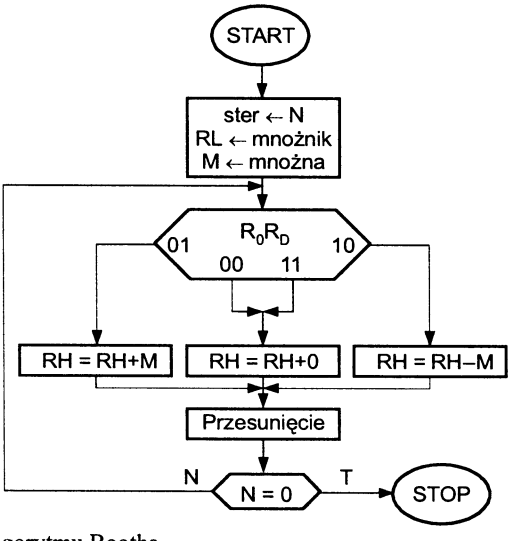
	mnożnik (−6)	mnożna (+5)
	0000 1010	0101
dodanie 0	0000	
przesunięcie	0000 0101	
dodanie +5	0101	
przesunięcie	0010 1010	
dodanie 0	0010	
przesunięcie	0001 0101	
dodanie +5	0110	
przesunięcie	0011 0010	
korekcja wyniku	1011	
dodanie −5	1110 0010	(wynik −30) ⊠

Po tych przykładach rozpatrzmy jeszcze jeden algorytm. W zapisie U2 można osiągnąć przyśpieszanie działań przez „grupowanie bitów". Przyśpieszenie polega na tym, że w niektórych krokach algorytmu pomija się operację dodawania. Na rysunku 2.5 pokazano sieć działań tzw. algorytmu Bootha, który polega na „grupowaniu pary bitów". W każdym kroku algorytmu modyfikacja zawartości rejestru RH następuje w zależności od wartości pary najmniej znaczących bitów rejestru wyniku częściowego. I tak jeśli para ta jest 00 lub 11, to w danym kroku do bardziej znaczącej części rejestru dodaje się 0. Jeśli para ta jest 01, to w danym kroku do bardziej znaczącej części rejestru dodaje się mnożną, a gdy para ta jest 10, to od bardziej znaczącej części rejestru odejmuje się mnożną.

Przykład 2.14. Wykonać algorytm Bootha dla liczb −6 (mnożnik) i +5 (mnożna) zapisanych w kodzie U2.

Rozwiązanie

	mnożnik (−6)	mnożna (+5)
	0000 1010	0101
dodanie 0	0000	
przesunięcie	0000 0101	
odjęcie +5	1011	
przesunięcie	1101 1010	
dodanie +5	0010	
przesunięcie	0001 0101	
odjęcie +5	1100	
przesunięcie	1110 0010	
	1110 0010	(wynik −30)

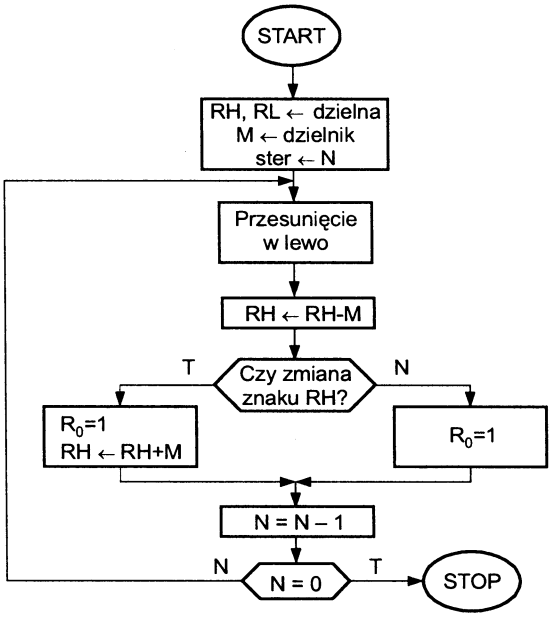


Rysunek 2.5. Sieć działań algorytmu Bootha

Pierwszy krok algorytmu Bootha zaczyna się od wpisania 0 do bardziej znaczącej części rejestru wyniku częściowego RH i mnożnika do mniej znaczącej części RL. Rejestr RL jest uzupełniany jedną pozycją po prawej stronie najmniej znaczącego bitu. Na pozycji tej jest wpisywane 0. Jak widać z przykładu algorytm Bootha, podobnie jak poprzednie algorytmy, wykonuje się w 4 krokach. ⊠

2.2.4. Dzielenie

Dzielenie liczb dwójkowych jest działaniem trudniejszym od ich mnożenia. Wynika to z faktu, że iloraz trzeba zapisać na skończonej liczbie bitów. Oznacza to, że wynik dzielenia może być przedstawiony tylko z pewną dokładnością. Z tego względu większość algorytmów dzielenia (jest ich bardzo dużo, a tu zostanie zaprezentowany jedynie naj-



Rysunek 2.6. Algorytm dzielenia

prostszy) daje w wyniku iloraz oraz resztę. Resztę R można zdefiniować na różne sposoby. Jednym z nich jest przyjęcie, że $R = D - IM$, gdzie D jest dzielną, I ilorazem i M dzielnikiem. Na rysunku 2.6 pokazano sieć działań algorytmu dzielenia. W stosunku do algorytmu mnożenia zmienia się:

1. Kolejność operacji w danym kroku (najpierw przesunięcie a potem operacja warunkowa).
2. Dodawanie zamienia się na odejmowanie.
3. Kierunek przesuwania zamienia się z prawa na lewo.

W algorytmie należy założyć liczbę kroków i długość wyniku. W arytmetyce stałopozycyjnej często zakłada się, że iloraz i reszta powinny być słowami o długości równej dzielnej i dzielnikowi. Założymy ponadto, że dzielenie będzie wykonywane w podobnym układzie jak w przypadku mnożenia, a więc z wykorzystaniem rejestru podwójnej długości dla dzielnej i pojedynczej długości dla dzielnika. Przykłady ilustrują przedstawiony algorytm.

Przykład 2.15. Podzielić w kodzie NKB liczbę +54 (dzielna) przez liczbę +5 (dzielnik).

Rozwiązanie

		dzielna (+54)	dzielnik (+5)
		0011 0110	0101
krok 1	przesunięcie	0110 110_	
	odejmowanie	0101	
		0001 1101	
krok 2	przesunięcie	0011 101_	
	odejmowanie	0101	
		wynik ujemny	
	przywrócenie	0011 1010	

krok 3	przesunięcie	0111 010_
	odejmowanie	0101
		0010 **0101**
krok 4	przesunięcie	0100 101_
	odejmowanie	0101
		wynik ujemny
	przywrócenie	0100 **1010**

Na początku algorytmu do rejestru podwójnej długości wpisano dzielną i do rejestru pojedynczej długości dzielnik. Każdy krok składa się z przesunięcia w lewo zawartości rejestru podwójnej długości oraz odjęcia od bardziej znaczącej części tego rejestru wartości dzielnika. Jeśli wynik odejmowania jest dodatni, to na najmniej znaczącą pozycję tego rejestru wpisuje się jedynkę, a w rejestrze wyniku częściowego podwójnej długości pozostaje otrzymana różnica. Jeśli natomiast wynik odejmowania jest ujemny, to na najmniej znaczącą pozycję rejestru wpisuje się zero, a bardziej znaczącą część tego rejestru przywraca się do postaci sprzed odejmowania. Po ostatnim kroku w mniej znaczącej części rejestru podwójnej długości jest wynik (jest to liczba całkowita), a w bardziej znaczącej części jest reszta. W przedstawionym przykładzie wytłuszczono otrzymywane bity wyniku. Po czwartym kroku iloraz jest słowem 1010, co oznacza liczbę 10_{10} (zawartość RL), a reszta jest słowem 0100, co oznacza 4_{10} (zawartość RH). ⊠

Pokazany przykład ilustruje fakt, że dzielna może być podwójnej długości, a dzielnik pojedynczej. Wówczas zarówno wynik jak i reszta są pojedynczej długości.

W zapisie ZM dzielenie wykonuje się według podanego algorytmu dla modułów liczb, a wartość bitu znaku ustala się biorąc znaki argumentów: jeśli są takie same, to wynik jest dodatni, a jeśli są różne, to wynik jest ujemny.

Przykład 2.16. Podzielić w kodzie U1 liczbę +54 (dzielna) przez liczbę −7 (dzielnik).

Rozwiązanie

		dzielna (+54)	dzielnik (−7)
		0 0011 0110	1000
krok 1	przesunięcie	0 0110 110_	
	odejmowanie	1000	
		wynik ujemny	
	przywrócenie	0110 **1100**	
krok 2	przesunięcie	0 1101 100_	
	odejmowanie	1000	
		0110 **1001**	
krok 3	przesunięcie	0 1101 001_	
	odejmowanie	1000	
		0110 **0011**	
krok 4	przesunięcie	0 1100 011_	
	odejmowanie	1000	
		0101 **0111**	

Otrzymany wynik jest poprawny: część całkowita wynosi 7 a ułamkowa 5. ⊠

2.3. Arytmetyka zmiennopozycyjna

2.3.1. Kody zmiennopozycyjne

Zapis zmiennopozycyjny wprowadzono, aby zwiększyć dokładność obliczeń. Odstęp pomiędzy sąsiednimi liczbami reprezentowanymi w kodach stałopozycyjnych zależał od liczby bitów. Przy stałej długości słowa zwiększanie zakresu reprezentowanych liczb odbywa się kosztem dokładności, czyli odstępu pomiędzy sąsiednimi liczbami. Zmiennopozycyjny zapis przedstawia liczby za pomocą trzech słów binarnych: jednego jednobitowego słowa znaku Z, n-bitowego słowa mantysy S (ang. *significant* lub *mantissa*) oraz m-bitowego słowa wykładnika E (ang. *exponent*). Wykładnik jest określany przy założeniu tzw. podstawy lub bazy. Najczęściej zakłada się, że jest ona równa 2. Wtedy liczbę dziesiętną A przedstawia się jako:

$$L(A) = (-1)^Z \times L(S) \times 2^{L(E)}$$

gdzie $L(S)$ i $L(E)$ są liczbami dziesiętnymi reprezentowanymi przez słowa S i E. W dalszym ciągu będziemy przez S i E oznaczać zarówno słowa reprezentacji dwójkowej, jak i odpowiadające im wartości liczbowe.

W reprezentacji zmiennopozycyjnej zakłada się, oprócz bazy 2 (stosowana jest czasem baza 10), że wartość mantysy jest normalizowana. Mantysa musi spełniać warunek:

$$0.5 \leq S < 1$$

Z tego założenia wynika, że każda liczba ma swoją jedyną reprezentację w zapisie zmiennopozycyjnym, której mantysa ma postać 0.1*xxxxxx*, gdzie *xxxxxx* są pozycjami binarnymi mantysy. Pozycje 0.1 nie muszą być zapamiętywane, a więc nie wchodzą w skład reprezentacji (zaoszczędzenie miejsca w rejestrach). Ponieważ wykładnik powinien przyjmować wartości zarówno dodatnie (liczby większe od 1), jak i ujemne (liczby mniejsze od 1), to najczęściej jest prezentowany w zapisie spolaryzowanym. Wtedy najmniejszą reprezentowaną liczbą jest liczba, dla której $S = 0.5$, a E przyjmuje największą ujemną wartość (słowo składające się z samych zer). Z samego przedstawienia liczb zmiennopozycyjnych (patrz wzór powyżej) wynika, że $L(A)$ nie może być równe 0. Dlatego w zapisie zmiennopozycyjnym liczba zero ma swoją specjalną reprezentację. Jest to najczęściej liczba, dla której S i E są słowami składającymi się z samych zer.

Dla prezentacji zmiennopozycyjnej opracowano wiele norm, lecz najpowszechniej stosowany jest standard amerykański IEEE 754 lub IEEE 854 (ten ostatni dopuszcza możliwość różnych baz). Standard IEEE 754 dopuszcza dwa podstawowe formaty liczb: pojedynczej precyzji (32 bity) i podwójnej precyzji (64 bity) z możliwościami ich rozszerzenia (pierwszy ponad 43 bity i drugi ponad 79 bitów). Format pojedynczej precyzji zakłada bit znaku, 23-bitową mantysę i 8-bitowy wykładnik. Format podwójnej precyzji zakłada bit znaku, 52-bitową mantysę i 11-bitowy wykładnik. Format pojedynczej precyzji ma zakres wykładnika [−126, +127] i zakres formatu ok. 2^{128}, czyli ok. 3.8×10^{38}. Dokładność tego formatu wynosi ok. 2^{-23} czyli ok. 10^{-7}. Format podwójnej długości ma zakres wykładnika [−1022, +1023] i zakres formatu ok. 2^{1024}, czyli ok. 9×10^{307}. Dokładność tego formatu wynosi ok. 2^{-52}, czyli ok. 10^{-15}. Przykłady omawiane w dalszej części tej książki będą, dla większej przejrzystości, wykorzystywać liczby o mniejszej długości niż przewidują standardy.

Przykład 2.17. Przedstawić w zapisie dziesiętnym liczbę zmiennopozycyjną:

Z	S	E
0	0101000	1001

Rozwiązanie. Z bitu znaku wynika, że jest to liczba dodatnia. Rzeczywista mantysa, po dodaniu trzech pierwszych znaków liczby 0.1 wynosi 0.10101000, a więc jest to liczba $\frac{1}{2}+\frac{1}{8}+\frac{1}{32} = 0.5 + 0.125 + 0.03125 = 0.65625$. Wykładnik w kodzie spolaryzowanym odpowiada liczbie +1. Zatem poszukiwana liczba to $+0.65625\times 2^1 = 1.3135$. ⊠

Przykład 2.18. Przedstawić w zapisie zmiennopozycyjnym (mantysa 8-bitowa i wykładnik 4-bitowy) liczbę dziesiętną 17.25.

Rozwiązanie. Najpierw przedstawia się liczbę w kodzie binarnym jako 0010001.0100. Kropka (zamiennie będzie nazywana przecinkiem) oddziela część całkowitą od części ułamkowej. Następnie trzeba znaleźć mantysę poprzez normalizację tej liczby. Wykonuje się to przez przesuwanie przecinka w lewo, aż do momentu gdy najbardziej znacząca jedynka znajdzie się po prawej stronie przecinka. Tak jest, gdy liczba jest większa od 1.

Jeśli natomiast po lewej stronie przecinka nie ma jedynek, to jest to liczba ułamkowa i jeśli jest konieczna jej normalizacja, to przecinek przesuwa się w prawo, aż do momentu gdy najbardziej znacząca jedynka znajdzie się bezpośrednio po prawej stronie przecinka. W naszym przykładzie przesuwa się przecinek w lewo o 5 pozycji: 00.100010100. Ponieważ przesuwanie przecinka o jedną pozycję oznacza bądź podzielenie jej przez 2 (przesuwanie w lewo) bądź pomnożenie jej przez 2 (przesuwanie w prawo), to dla zachowania jej wartości wykładnik powinien być równy liczbie przesunięć, a znak wykładnika zależy od kierunku przesuwania (w lewo — dodatni i w prawo ujemny). W naszym przykładzie przesuwanie było o 5 pozycji w lewo, a więc wykładnik wynosi +5, co w zapisie 4-bitowym spolaryzowanym przedstawia się jako 1101. Zatem zapis zmiennopozycyjny przykładowej liczby jest: 0 00010100 1101. ⊠

2.3.2. Działania na liczbach zmiennopozycyjnych

Bardzo często szybkość działania komputerów podaje się w jednostkach zwanych MFLOP, tzn. w milionach operacji zmiennopozycyjnych na sekundę (ang. *mega floating point operation*). Posługiwanie się taką jednostką lepiej oddaje szybkość obliczeń niż liczba wykonywanych rozkazów na sekundę MIPS (ang. *mega instruction per second*). Projektanci komputerów ciągle dążą do opracowania jak najszybszych układów realizujących operacje zmiennopozycyjne.

Dodawanie i odejmowanie

Operacje dodawania i odejmowania w zapisie zmiennopozycyjnym składają się z kilku kroków. Aby wykonać działania na mantysach trzeba najpierw przeprowadzić procedurę wyrównania wykładników. Procedura ta jest wykonywana w taki sposób, że mantysa liczby mniejszej (o mniejszym wykładniku) jest zmniejszana. Innymi słowy mniejszy wykładnik jest zwiększany aż do osiągnięcia wartości większego wykładnika. Zmniejszania mantysy dokonuje się poprzez przesuwanie jej w prawo o tyle pozycji ile wynika z konieczności wyrównania wykładników.

Niech będą dane dwie liczby (pominiemy ich znaki) $L(A) = S_A \times 2^{\alpha}$ i $L(B) = S_B \times 2^{\beta}$. Załóżmy, że $\alpha < \beta$ co oznacza, że liczba B jest większa od liczby A. Wyrównanie

wykładników wymaga zwiększenia wykładnika α, czyli zmniejszenia mantysy liczby A, tj. przesunięcia przecinka mantysy S_A w lewo $\beta - \alpha$ razy. Wówczas otrzymamy, że:

$$L(A) = (S_A \times 2^{-(\beta-\alpha)}) \times 2^{(\alpha+(\beta-\alpha))} = (S_A \times 2^{(\alpha-\beta)}) \times 2^{\beta}$$

Oczywiście tak otrzymana nowa mantysa $S'_A = S_A \times 2^{(\alpha-\beta)}$ może nie być znormalizowana i taką bierze się do dalszych działań. W drugim kroku można obliczyć sumę albo różnicę liczb A i B sumując, albo odejmując mantysy i biorąc wykładnik większej liczby:

$$A + B = (S_A \times 2^{(\alpha-\beta)} + S_B) \times 2^{\beta}$$
$$A - B = (S_A \times 2^{(\alpha-\beta)} - S_B) \times 2^{\beta}$$

W ogólnym przypadku tak otrzymane mantysy sumy i różnicy liczb mogą być nie znormalizowane. Dlatego w trzecim kroku algorytmu dodawania lub odejmowania należy znormalizować mantysę wyniku i odpowiednio do tego zmienić wartość wykładnika. Działanie dodawania może spowodować zwiększenie wykładnika jedynie o 1, gdyż z sumowania mantys nie może powstać liczba większa od 1, natomiast działanie odejmowania może spowodować konieczność zmniejszenia wykładnika o więcej niż 1.

Przykład 2.19. Dodać dwie liczby zmiennopozycyjne A i B o 8-bitowej mantysie i 4-bitowym wykładniku:

A	—	0	00101001	1101
B	—	0	00011010	1010

Rozwiązanie. Kolejne kroki algorytmu są następujące:

1. Porównanie wykładników $\alpha = 5$ i $\beta = 2$ wskazuje na konieczność przesunięcia przecinka mantysy liczby B o 3 pozycje w lewo. Otrzymamy:

$$S_A = 0.100101001 \qquad S'_B = S_B \times 2^{(\beta-\alpha)} = 0.000100011$$

2. Suma S_A i S'_B wynosi $S_A + S_B = 0.101001100$ i nie wymaga normalizacji, tj. wykonania trzeciego kroku normalizacji, a zatem suma $A + B$ jest liczbą zmiennopozycyjną: 0 01001100 1101.

Sprawdzenie. Sprawdźmy wyniki tych działań na liczbach dziesiętnych:

$$A = (2^{-1} + 2^{-4} + 2^{-6} + 2^{-9}) \times 2^5 = 2^4 + 2^1 + 2^{-1} + 2^{-4} = 18.5625$$
$$B = (2^{-1} + 2^{-5} + 2^{-6} + 2^{-8}) \times 2^2 = 2^1 + 2^{-3} + 2^{-4} + 2^{-6} = 2.203125$$

Suma tych liczb wynosi zatem 20.765625. ⊠

Natomiast otrzymany wynik wynosi

$$A + B = (2^{-1} + 2^{-3} + 2^{-6} + 2^{-7}) \times 2^5 = 2^4 + 2^2 + 2^{-1} + 2^{-2} = 20.75$$

Sprawdzenie zatem nie potwierdziło prawidłowości obliczeń, gdyż wynik dodawania dziesiętnego (20.765625) różni się od obliczonego o 0.015625. Fakt ten staje się oczywisty, gdy weźmie się pod uwagę przesuwanie mantysy podczas wyrównywania wykładników. Niektóre jedynki mantysy przesuwanej mogą zostać pominięte. W naszym przykładzie usunięto jedynkę o wadze 2^{-8} i ona właśnie spowodowała błąd o $2^{-8} \times 2^2 = 2^{-6} = 0.015625$.

Przykład 2.20. Odjąć od liczby A liczbę B.

Rozwiązanie

1. Pierwszy krok algorytmu odejmowania jest taki sam jak algorytmu dodawania.
2. Mantysa różnicy $A - B = 0.100000110$.

3. Normalizacja nie jest potrzebna i otrzymujemy wynik:
$(A-B)$ 0 00000110 1101.

Sprawdzenie. Dziesiętnie różnica $(A-B)$ wynosi 16.359375. Natomiast otrzymany wynik $0.51171875 \times 2^5 = 16.375$. Powstały błąd jest tej samej natury co poprzednio, a mianowicie błąd wynikający z usunięcia bitów przy przesuwaniu. ⊠

Przykład 2.21. Dane są liczby: A — 0 10000000 1001
B — 0 01000000 1010
Odjąć od liczby A liczbę B.

Rozwiązanie. Kolejne kroki algorytmu są następujące:

1. Porównanie wykładników $\alpha = 1$ i $\beta = 2$ wskazuje na konieczność przesunięcia przecinka mantysy liczby A o 1 pozycję w lewo. Po wyrównaniu wykładników otrzymamy $S_A = 0.0110000000$.
2. Różnicę S_A i S_B oblicza się jak w zapisie ZM. Trzeba wybrać większy moduł i przyjąć odpowiedni znak wyniku. W przykładzie S_B jest większe od S_A i dlatego obliczając $S_A - S_B$ w rzeczywistości oblicza się $S_B - S_A$ i przyjmuje znak minus.

$$S_B = 0.101000000$$
$$S_A = 0.011000000$$
$$S_B - S_A = 0.010000000$$

3. Mantysa różnicy $S_B - S_A$ wymaga normalizacji i dlatego otrzymamy $S_B - S_A = 0.100000000 \times 2^{-1}$, co oznacza konieczność zmniejszenia wykładnika o 1. Wynik odejmowania jest zatem następujący: $S_B - S_A = 1$ 00000000 1001.

Sprawdzenie. Liczba $A = +0.11000000 \times 2^1 = +1.5$ i liczba $B = +0.10100000 \times 2^2 = +2.5$. Różnica $A - B = -0.10000000 \times 2^1 = -1$. ⊠

Mnożenie i dzielenie

Działania mnożenia i dzielenia liczb zmiennopozycyjnych nie wymagają wyrównywania wykładników. Działania te wykonuje się wg wzorów:

$$A \times B = (S_A \times S_B) \times 2^{(\alpha+\beta)}$$
$$A : B = (S_A : S_B) \times 2^{(\alpha-\beta)}$$

Działania wykonuje się na mantysach oraz na wykładnikach. Po wymnożeniu lub podzieleniu mantys trzeba dokonać normalizacji. Sumowanie lub odejmowanie wykładników wykonywane jest w zapisie spolaryzowanym. Stosując ten zapis należy pamiętać, że działania w tym zapisie wymagają odpowiednich korekt wyniku. Dodawanie dwóch liczb w zapisie spolaryzowanym (wykładników liczby zmiennopozycyjnej podczas mnożenia) można wykonać jak w kodzie NKB, ale od wyniku należy odjąć współczynnik polaryzacji (100...0). Natomiast odejmując dwie liczby (podczas dzielenia) jak w kodzie NKB należy wynik skorygować, dodając do niego współczynnik polaryzacji.

Algorytm mnożenia można przedstawić następująco:

1. Pomnożyć mantysy.
2. Zaokrąglić wynik i dokonać jego normalizacji.
3. Dodać wykładniki.
4. Korekcja wykładnika wynikająca z normalizacji.

Algorytm dzielenia można przedstawić następująco:

1. Podzielić mantysy $S_A : S_B$.
2. Normalizacja ilorazu.
3. Odjąć wykładniki.
4. Korekcja wykładnika wynikająca z normalizacji ilorazu mantys oraz odejmowania wykładników.

Przykład 2.22. Pomnożyć +1.5 przez +2.5.

Rozwiązanie

1. $S_A = 0.110000000, S_B = 0.101000000$, $S_A \times S_B = 0.011110000$.
2. Po normalizacji: $S_{A\times B}$ wynosi $0.111100000 \times 2^{-1}$.
3. Dodając wykładniki otrzymamy:

$$\begin{array}{r} 1001 \\ +\,1010 \\ \hline 10011 \end{array}$$

4. Wynik sumowania wykładników należy skorygować odejmując wartość polaryzacji:

$$\begin{array}{r} 10011 \\ -\,1000 \\ \hline 1011 \end{array}$$

Ze względu na przeprowadzoną normalizację mantysy wykładnik trzeba zmniejszyć o jeden i w efekcie iloczyn $A \times B$ przedstawia się jako 0 11100000 1010.

Sprawdzenie. Otrzymana liczba to: $+0.1111000000\times 2^2 = +3.75$, co jest dokładnym wynikiem. ⊠

Przykład 2.23. Pomnożyć liczby z przykładu 2.19, tj. +18.5625 przez +2.203125.

Rozwiązanie

1. $S_A \times S_B = 0.010100011100101010$. Wymnażając mantysy otrzymuje się liczby podwójnej długości. Jeśli wynik mnożenia ma być liczbą pojedynczej długości, to wykonuje się obcięcie wyniku.
2. Po obcięciu oraz normalizacji: $S_{A\times B}$ — $0.101000111 \times 2^{-1}$.
3. Wykładnik wynosi $(1101+1010)-1000 = 10111-1000 = 1111$.
4. Korekcja wykładnika po normalizacji mantysy $1111-1001=1110$.

Zatem iloczyn $A \times B = 0$ 01000111 1110.

Sprawdzenie. Przedstawiając wynik dziesiętnie otrzymamy $0.638671875 \times 2^6 = 40.875$, podczas gdy mnożenie dziesiętnych argumentów daje wynik 40.8955078125. Powstały przez obcięcie wyniku mnożenia błąd wynosi:

$$(2^{-13}+2^{-15}+2^{-17})\times 2^7 = 2^{-6}+2^{-8}+2^{-10} = 0.0205078125.$$ ⊠

Przykład 2.24. Podzielić liczby z poprzedniego przykładu.

Rozwiązanie

1. Dzieląc S_A przez S_B otrzymamy 1.000011010 (zaleca się Czytelnikowi podzielenie mantys w kodzie NKB).
2. Po normalizacji iloraz mantys wynosi: $S_{A:B} = 0.100001101 \times 2^1$.
3. Różnica wykładników wynosi: 1011.
4. Wykładnik po korekcji wynosi: 1100.

Postać ilorazu: 0 00001101 1100.

Sprawdzenie. Otrzymany iloraz to $+0.100001101 \times 2^4 = +0.5234275 \times 2^4 = 8.375$. Natomiast dzieląc 18.5625 przez 2.203125 otrzymamy 8.425531915. Różnica powstała podczas dzielenia mantys, które wykonano w 9 krokach, gdyż zarówno dzielna jak i dzielnik były 9-bitowe. Gdyby obliczyć iloraz z większą dokładnością, to wynik mógłby być także bardziej dokładny. ⊠

Zadania dla Czytelnika

1. Wykonać algorytm mnożenia dwóch 8-bitowych liczb w kodach U1 i U2:
   $(-29, -7)$, $(-17, +11)$, $(+9, -12)$, $(+13, +22)$.
2. Wykonać algorytm Bootha dla liczb -14 i $+15$.
3. Wykonać algorytm dzielenia w kodzie U1 dla liczb $+78$ i -15.
4. Założyć format liczb zmiennopozycyjnych i wykonać 4 działania arytmetyczne dla liczb -16.25 i 122.675. ⊠

Cyfrowe układy kombinacyjne 3

3.1. Podstawy projektowania cyfrowych układów kombinacyjnych

3.1.1. Wstęp

Każdy układ cyfrowy można przedstawić jako „czarną skrzynkę" (blok) z określoną liczbą wejść i wyjść. Sygnały wejściowe i wyjściowe są sygnałami dwójkowymi (binarnymi), tj. przyjmują jedną z dwóch wartości: zero lub jeden. Kombinacja wartości sygnałów wejściowych danego układu nazywana jest **słowem wejściowym, stanem wejściowym, wektorem wejściowym** albo **wzbudzeniem układu**, natomiast kombinacja wartości sygnałów wyjściowych — **słowem (wektorem) wyjściowym, stanem wyjść** albo **odpowiedzią układu**. Działanie układu opisuje się zależnością między zbiorem słów wejściowych i zbiorem słów wyjściowych. Wyróżnia się dwie klasy układów:

— układy kombinacyjne, dla których stan wyjść w każdej chwili jest jednoznacznie określony przez stan wejść,
— układy sekwencyjne, dla których stan wyjść w danej chwili zależy od stanu wejść w tej chwili oraz od stanu wejść w chwilach poprzednich.

W tym rozdziale przedstawiono jedynie układy kombinacyjne. Omówiono sposoby ich opisu, podstawowe prawa algebry Boole'a i funkcje boolowskie opisujące zależność pomiędzy elementami zbioru słów wejściowych i elementami zbioru słów wyjściowych. Przedstawiono metody projektowania układów kombinacyjnych oraz najpowszechniej stosowane standardowe układy. Na koniec przedstawiono zasady budowania układów cyfrowych za pomocą układów programowanych.

3.1.2. Prawa algebry Boole'a

Algebra Boole'a jest algebrą z trzema operacjami na dwuwartościowych argumentach, które przyjmują wartości: 0 i 1. Rezultaty tych operacji są także dwuwartościowe. Te trzy operacje, to:

— suma logiczna (suma boolowska, alternatywa),
— iloczyn logiczny (iloczyn boolowski, koniunkcja),
— negacja (inwersja).

Dwie pierwsze operacje są n-argumentowe, a trzecia jest jednoargumentowa. Operacja **sumy logicznej** jest zdefiniowana następująco: jeżeli co najmniej jeden z argumentów jest równy 1, to wynik jest równy 1. Zatem suma logiczna jest równa 0 tylko dla przypadku, gdy wszystkie argumenty są równe 0. **Operacja iloczynu logicznego** jest zdefiniowana następująco: wynik iloczynu jest równy 1, wtedy i tylko wtedy, gdy wszystkie argumenty przyjmują wartość 1. Operacja **negacji** zmienia wartość argumentu na przeciwny.

Operacje sumy i iloczynu mają następujące własności:

1)	przemienność	$A+B=B+A,$	$A\cdot B=B\cdot A,$
2)	łączność	$(A+B)+C=A+(B+C),$	$(A\cdot B)\cdot C=A\cdot(B\cdot C),$
3)	rozdzielczość	$A+(B\cdot C)=(A+B)\cdot(A+C),$	$A\cdot(B+C)=A\cdot B+A\cdot C,$
4)	tożsamość	$A+0=A,$	$A\cdot 0=0,$
		$A+1=1,$	$A\cdot 1=A,$
		$A+A=A,$	$A\cdot A=A,$
5)	komplementarność	$A+\overline{A}=1,$	$A\cdot\overline{A}=0,$

oraz spełniają poniższe prawa:

1)	prawo de Morgana	$\overline{A+B}=\overline{A}\cdot\overline{B},$	$\overline{A\cdot B}=\overline{A}+\overline{B},$
2)	prawo sklejania	$A\cdot\overline{B}+A\cdot B=A,$	$(A+\overline{B})\cdot(A+B)=A,$
3)	prawo pochłaniania	$A\cdot\overline{B}+B=A+B.$	

Trzy omówione operacje (suma, iloczyn, negacja) przyporządkowują słowom dwójkowym (argumentom operacji) wartości dwójkowe, a więc określają pewne funkcje. Wszystkich funkcji dwóch zmiennych jest 16 (dla n zmiennych jest ich 2 do potęgi 2^n) i przedstawiono je w tablicy 3.1.

Tablica 3.1. Wszystkie funkcje dwóch zmiennych

x_1	x_0	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

$f_0 = 0$	funkcja stała
$f_1 = \overline{x_0}\,\overline{x_1} = \overline{x_0 + x_1}$	funkcja NOR
$f_2 = x_0\overline{x_1}$	funkcja iloczynu z negacją x_1
$f_3 = \overline{x_0}\,\overline{x_1} + x_0\overline{x_1} = \overline{x_1}$	funkcja negacji x_1
$f_4 = \overline{x_0}x_1$	funkcja iloczynu z negacją x_0
$f_5 = \overline{x_0}\,\overline{x_1} + \overline{x_0}x_1 = \overline{x_0}$	funkcja negacji x_0
$f_6 = \overline{x_0}x_1 + x_0\overline{x_1}$	funkcja sumy mod 2, EXOR
$f_7 = \overline{x_0}\,\overline{x_1} + \overline{x_0}x_1 + x_0\overline{x_1} = \overline{x_0 x_1}$	funkcja NAND
$f_8 = x_0 x_1$	funkcja iloczynu AND
$f_9 = \overline{x_0}\,\overline{x_1} + x_0 x_1$	funkcja równoważności
$f_{10} = x_0\overline{x_1} + x_0 x_1 = x_0$	funkcja tożsama ze zmienną x_0
$f_{11} = \overline{x_0}\,\overline{x_1} + x_0\overline{x_1} + x_0 x_1 = x_0 + \overline{x_1}$	funkcja implikacji x_0 przez x_1
$f_{12} = \overline{x_0}x_1 + x_0 x_1 = x_1$	funkcja tożsama ze zmienną x_1
$f_{13} = \overline{x_0}\,\overline{x_1} + \overline{x_0}x_1 + x_0 x_1 = \overline{x_0} + x_1$	funkcja implikacji x_1 przez x_0
$f_{14} = x_0\overline{x_1} + \overline{x_0}x_1 + x_0 x_1 = x_0 + x_1$	funkcja sumy OR
$f_{15} = 1$	funkcja stała

3.1.3. Sposoby przedstawiania funkcji boolowskich

Najczęściej stosowane są cztery sposoby opisu prostych układów cyfrowych, a tym samym przedstawiania funkcji boolowskich:
1) tablica prawdy,
2) algebraiczny zapis funkcji,
3) dziesiętny zapis funkcji,
4) mapa Karnaugha.

Tablica 3.2. Tablica prawdy funkcji f

x_2	x_1	x_0	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tablica prawdy (por. tablica 3.2) funkcji n zmiennych ma $n+1$ kolumn (n kolumn dla zmiennych wejściowych i jedna dla wartości funkcji) i 2^n wierszy, bo tyle różnych kombinacji przyjmuje n zmiennych. W tablicy 3.2 przedstawiono funkcję 3 zmiennych. Dla pięciu kombinacji zmiennych wejściowych funkcja przyjmuje wartość jeden, a dla trzech kombinacji wartość zero. Każdą kombinację zmiennych wejściowych można wyróżnić za pomocą jednej z dwóch prostych funkcji boolowskich. Prostą funkcją boolowską nazywać będziemy funkcję, którą można przedstawić za pomocą jednego iloczynu wszystkich zmiennych lub za pomocą jednej sumy wszystkich zmiennych. Prostą funkcją pierwszego rodzaju nazywać będziemy funkcję, która przyjmuje wartość 1 tylko dla jednej kombinacji zmiennych. Prostą funkcją drugiego rodzaju nazywać będziemy funkcję, która tylko dla jednej kombinacji zmiennych, przyjmuje wartość 0. Na przykład kombinacja zmiennych $x_0 = 1$, $x_1 = 0$ i $x_2 = 1$ odpowiada prostej funkcji $z_1 = x_0\bar{x}_1x_2$, która przyjmuje wartość 1 tylko dla tej kombinacji, a dla innych — wartość zero. Tę samą kombinację zmiennych można wskazać drugą prostą funkcją, a mianowicie $z_2 = \bar{x}_0 + x_1 + \bar{x}_2$, która przyjmuje wartość 0 tylko dla tej kombinacji, a dla wszystkich innych wartość 1. Każdą funkcję boolowską można przedstawić jako sumę odpowiednich prostych funkcji pierwszego rodzaju, tzw. iloczynów elementarnych (iloczyn wszystkich zmiennych — zanegowanych bądź nie) realizujących jedynki funkcji. Mówimy wtedy o postaci sumacyjnej. Można ją także przedstawić jako iloczyn odpowiednich prostych funkcji drugiego rodzaju, tzw. sum elementarnych (suma wszystkich zmiennych — zanegowanych bądź nie) realizujących zera funkcji i wtedy mówimy o postaci iloczynowej.

Algebraiczny kanoniczny zapis funkcji **wykorzystuje dwie podstawowe postacie zapisu:**
1) postać sumacyjną (alternatywną),
2) postać iloczynową (koniunkcyjną).

Przykładową funkcję opisaną jak w tablicy 3.2 można zapisać na dwa sposoby:

$$y = x_0\bar{x}_1\bar{x}_2 + \bar{x}_0x_1\bar{x}_2 + x_0\bar{x}_1x_2 + \bar{x}_0x_1x_2 + x_0x_1x_2$$
$$y = (x_0 + x_1 + x_2)(\bar{x}_0 + \bar{x}_1 + x_2)(x_0 + x_1 + \bar{x}_2)$$

Pierwszy sposób polega na zsumowaniu wszystkich iloczynów elementarnych, dla których funkcja przyjmuje wartość 1. Taką postać nazywa się kanoniczną postacią sumacyjną. Drugi sposób polega na utworzeniu iloczynu wszystkich sum elementarnych, dla których funkcja przyjmuje wartość 0. Taką postać nazywa się kanoniczną postacią iloczynową. Korzystając z praw algebry Boole'a można upraszczać te postacie tak, aby otrzymać postać sumacyjną o mniejszej liczbie składników zawierających iloczyny o mniejszej liczbie czynników lub postać iloczynową o mniejszej liczbie czynników składających się z sum o mniejszej liczbie składników. Procedury prowadzące do znalezienia takich postaci nazywane są minimalizacją funkcji boolowskich i opisano je w następnym punkcie.

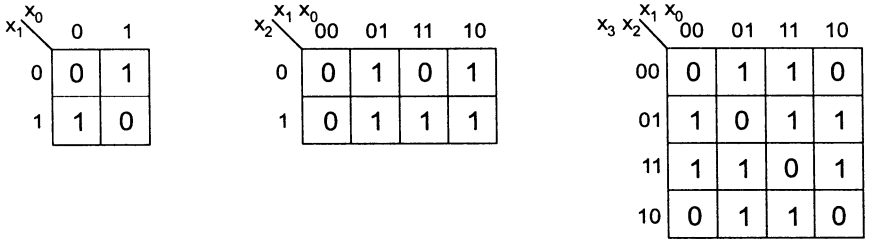
Trzecim sposobem przedstawienia funkcji boolowskiej jest **dziesiętna postać zbioru iloczynów lub sum elementarnych określających jedynki lub zera funkcji**. Odpowiednim kombinacjom zmiennych przyporządkowuje się liczby dziesiętne i one tworzą elementy zbioru. Odpowiedniość tę można ustalić na wiele sposobów, choć najwygodniejszy z nich to taki, który bezpośrednio wiąże indeksy zmiennych z wagą pozycji w zapisie dwójkowym. Kombinacji n zmiennych $x_{n-1},\ldots, x_0$ odpowiada liczba dziesiętna $L(x_n)$, gdzie

$$L(x_n) = \sum_{i=0}^{n-1} x_i 2^i$$

Kanonicznym postaciom (sumacyjnej i iloczynowej) przykładowej funkcji odpowiadają w zapisie dziesiętnym zbiory liczb dziesiętnych o postaciach:

$$y = \sum_{3}(1, 2, 5, 6, 7) \qquad y = \prod_{3}(0, 3, 4)$$

Liczby pod symbolami sumy i iloczynu wskazują na liczbę zmiennych, a zbiory liczb w nawiasach wskazują na kombinacje zmiennych odpowiadające odpowiednio jedynkom i zerom funkcji.



Rysunek 3.1. Mapy Karnaugha funkcji 2, 3 i 4 zmiennych.

Czwartym sposobem przedstawienia funkcji boolowskiej jest tzw. **mapa Karnaugha**. Jest to zapis graficzny przedstawiający wartości funkcji dla poszczególnych kombinacji zmiennych w odpowiednich polach prostokąta. Na rysunku 3.1 pokazano mapy Karnaugha funkcji 2, 3 i 4 zmiennych. Mapa funkcji dwóch zmiennych ma 4 pola. Na rysunku 3.1 przedstawiono mapę dla funkcji EXOR. Mapa funkcji 3 zmiennych ma 8 pól. Na rysunku 3.1 pokazano wypełnienie pól dla przykładowej funkcji z tablicy 3.2. Mapa dla 4 zmiennych ma 16 pól. W ogólnym przypadku mapa funkcji n zmiennych ma 2^n pól. W praktyce daje się stosować mapy do 5 zmiennych, ponieważ mapy o większej liczbie zmiennych jest trudno narysować.

Graficzne przedstawienie funkcji pozwala projektantom układów logicznych na zaobserwowanie pewnych cech funkcji, które pozwolą zrealizować ją za pomocą jak najprostszych rozwiązań. Wymaga to jednak pewnej wprawy i dla jej nabrania zaleca się Czytelnikowi samodzielne wykonywanie zadań prezentowanych w niniejszej książce.

Mapy Karnaugha tworzy się za pomocą tzw. **kodu Graya**. Kodem Graya, zwanym też kodem refleksyjnym, można przedstawiać liczby słowami o długości równej długości słów kodu NKB. Słowo kodu Graya powstaje przez dodawanie do każdej pozycji słowa z kodu NKB odpowiednich bitów tego samego słowa przesuniętego o jedną pozycję w prawo, a z lewej strony uzupełnionego zerem. Przykładowo reprezentując liczbę 27 w kodzie NKB otrzymamy 5-bitowe słowo 11011. Przesunięcie o jedną pozycję w prawo daje słowo 01101. Po dodaniu (dodaje się tylko odpowiednie pozycje bez przeniesienia) otrzyma się:

$$\begin{array}{c} 11011 \\ \underline{01101} \\ 10110 \end{array}$$

Słowo 10100 jest słowem kodu Graya odpowiadającym liczbie 27. W tablicy 3.3 pokazano przypisanie 4-bitowych słów kodu Graya słowom kodu NKB.

Tablica 3.3. Przypisanie słów kodu Graya słowom kodu NKB

	NKB	Gray		NKB	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Kod Graya ma taką cechę, że jego sąsiednie słowa różnią się tylko na jednej pozycji. Przypisując kolejne słowa kodu Graya kolejnym wierszom i kolumnom mapy Karnaugha otrzymamy, że sąsiednie pola mapy odpowiadają takim kombinacjom zmiennych, które różnią się tylko na jednej pozycji. Dlatego jeśli w sąsiednich polach mapy Karnaugha znajdują się takie same wartości, to kombinacje zmiennych odpowiadające tym polom mapy podlegają prawu sklejania, co pozwala na pewne upraszczanie funkcji (por. minimalizacja funkcji).

Rozpatrzmy mapę Karnaugha funkcji trzech zmiennych pokazaną na rysunku 3.2a. Pola mapy Karnaugha oznaczono numerami od 0 do 7. Są to liczby dziesiętne odpowiadające słowom dwójkowym stanowiącym kombinacje zmiennych. Na przykład, jeśli kombinacja zmiennych $x_2x_1x_0$ jest 110, to pole odpowiadające tej kombinacji oznaczono numerem 6. Można zauważyć, biorąc pod uwagę jedynki funkcji, że istnieją 4 pary sąsiednich jedynek.

a)

x_0 \ x_2x_1	00	01	11	10
0	0_0	0_2	1_6	1_4
1	1_1	1_3	1_7	0_5

b)

x_0 \ x_2x_1	00	01	11	10
0	0	0	1	1
1	1	1	1	0

Rysunek 3.2. a) mapa Karnaugha funkcji zadanej tablicą prawdy (tabl. 3.2), b) sklejanie par sąsiednich jedynek funkcji

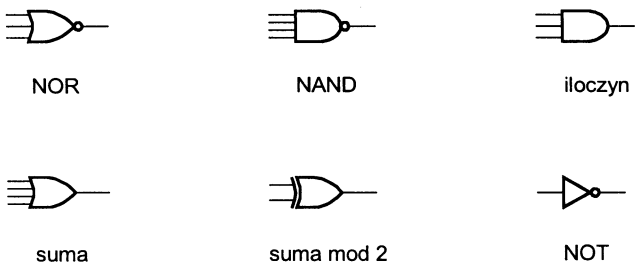
Są to pary: (1,3), (3,7), (6,7) i (4,6). Na rysunku 3.2b zaznaczono owalnym zakreśleniem wszystkie 4 pary.

Para (1,3) ma jako współrzędne x_0 i $\bar{x}_2$ (są to jedynki funkcji odpowiadające kombinacjom zmiennych $x_0\bar{x}_1\bar{x}_2$ i $x_0x_1\bar{x}_2$) i odpowiada iloczynowi $x_0\bar{x}_2$. Para (3,7) odpowiada iloczynowi x_0x_1, para (6,7) iloczynowi x_1x_2 i para (4,6) iloczynowi $\bar{x}_0x_2$.

Istnieją także trzy zera funkcji, ale tylko dwa z nich tworzą parę. Jest to para (0,2) odpowiadająca sumie $x_0 + x_2$, natomiast zero funkcji w polu oznaczonym numerem 5 odpowiada sumie $\bar{x}_0 + x_1 + \bar{x}_2$.

3.2. Projektowanie układów cyfrowych na bramkach

Kombinacyjne układy cyfrowe najczęściej budowane są za pomocą tzw. bramek. **Bramką** (ang. *gate*) nazywa się układ elektroniczny realizujący funkcję boolowską. Jak każdy układ elektroniczny, tak i bramki opisywane są wieloma parametrami zarówno funkcjonalnymi (liczba wejść, liczba wyjść, realizowana funkcja, przeznaczenie i inne), jak i elektrycznymi (pobierana moc zasilania, obciążalność prądem układów sterujących wejściami, możliwość wysterowania wejść innych układów itp.) oraz dynamicznymi (czasy zmiany sygnału na wyjściu układu, wnoszone opóźnienia i inne). Tutaj najbardziej interesować nas będą funkcje realizowane przez bramki a inne parametry (elektryczne i dynamiczne) traktowane będą pomocniczo. Bramki produkowane są jako układy scalone. Do produkcji układów scalonych stosowane są różne technologie. Najpopularniejsze spośród nich to technologia TTL (ang. *transistor-transistor logic*) i CMOS (ang. *complementary MOS*). W jednym układzie scalonym znajduje się zwykle kilka bramek. Przykładowo w jednym układzie scalonym znajdują się 4 bramki dwuwejściowe lub trzy bramki trzywejściowe lub dwie bramki czterowejściowe lub jedna bramka ośmiowejściowa. Na rysunku 3.3 pokazano oznaczenia najczęściej stosowanych bramek.



Rysunek 3.3. Najczęściej stosowane oznaczenia bramek

Pokazane na rysunku 3.3 bramki iloczynu, sumy, NAND (ang. *NOT AND*) — negacja iloczynu i NOR (ang. *NOT OR*) — negacja sumy są bramkami wielowejściowymi (na rysunku 3.3 są to 4- lub 3-wejściowe). Bramki sumy modulo 2 (ang. *exclusive or*) — EXOR występują tylko jako dwuwejściowe. Bramka inwertera jest jednowejściowa. Spotyka się także bramki w wykonaniu specjalnym. Mogą to być tzw. bramki z otwartym kolektorem (ang. *open collector*) — OC stosowane celem uzyskania możliwości zwierania wyjść bramek lub bramki trójstanowe (ang. *three-state logic*) stosowane w realizacji magistral (szyn) przesyłowych.

3.2.1. Minimalizacja funkcji boolowskich

Problem upraszczania funkcji boolowskich powstał z konieczności zmniejszania liczby elementów stosowanych do realizacji tych funkcji. Minimalizacja może dotyczyć liczby stosowanych bramek, może dotyczyć liczby układów scalonych, a może dotyczyć liczby wymaganych połączeń w elementach programowanych (współczesne układy cyfrowe realizuje się często za pomocą jednego układu scalonego, w którym umieszcza się matrycę elementów a projektant musi je odpowiednio połączyć — por. rozdz.3.4). Dla prostych funkcji (do 5 zmiennych) stosować można metodę minimalizacji wykorzystującą mapy Karnaugha. Natomiast dla bardziej złożonych funkcji stosuje się metody wykorzystujące wspomaganie komputerowe, a więc metody algorytmiczne. Podstawą tych metod jest sposób minimalizacji podany przez McCluskeya i Quine'a, który przedstawiono w rozdziale 3.2.3. Dalej podano zasady korzystania z map Karnaugha.

Rozważmy funkcję 4 zmiennych opisaną mapą Karnaugha pokazaną na rysunku 3.4a. Na rysunku 3.4b pokazano jakim wartościom dziesiętnym odpowiadają pola takiej mapy Karnaugha. Z mapy widać, że funkcja zawiera 8 jedynek i 8 zer. Zakładając, że dostępne są zarówno zmienne wejściowe jak i ich negacje, funkcję tę można przedstawić jako sumę ośmiu prostych funkcji pierwszego rodzaju, a więc można ją zrealizować za pomocą bramki sumy 8-wejściowej, do której wejść dołączono wyjścia ośmiu iloczynów 4-wejściowych. Można tę funkcję przedstawić też jako iloczyn ośmiu funkcji drugiego rodzaju i zrealizować za pomocą 8-wejściowej bramki iloczynu, do której wejść dołączono wyjścia 8 bramek sum 4-wejściowych (por. postacie kanoniczne). Zadaniem procesu minimalizacji jest znalezienie prostszego rozwiązania. Tutaj będziemy poszukiwać rozwiązania składającego się z mniejszej liczby bramek. Mogłoby to być także poszukiwanie rozwiązania składającego się z najmniejszej liczby układów scalonych lub rozwiązania wykorzystującego jak najprostszy układ wielkoscalony.

a)

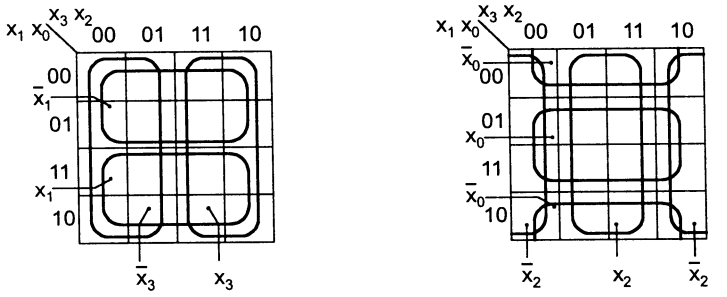
x_3x_2 \ x_1x_0	00	01	11	10
00	1	0	1	1
01	0	1	1	0
11	0	1	0	0
10	1	0	0	1

b)

x_3x_2 \ x_1x_0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

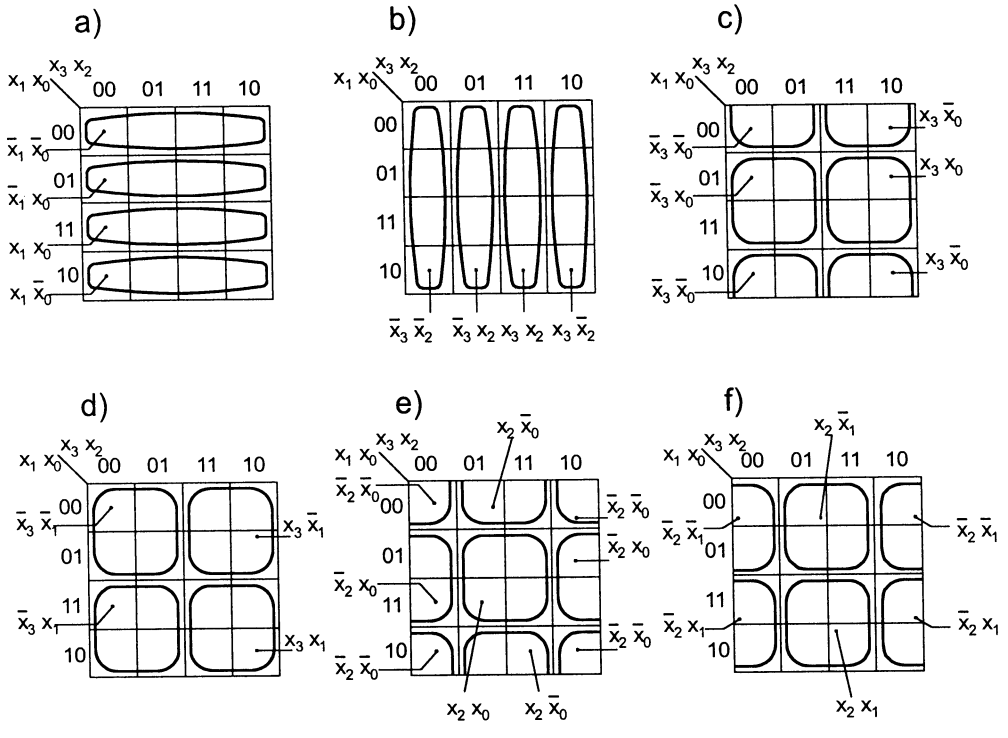
Rysunek 3.4. Mapy Karnaugha funkcji 4 zmiennych: a) przykładowa funkcja, b) oznaczenia pól

W celu znalezienia rozwiązania o najmniejszej liczbie bramek trzeba, posługując się mapą Karnaugha, znaleźć możliwie największe grupy sąsiednich jedynek (lub zer) danej funkcji. Dla funkcji 4 zmiennych największymi grupami mogą być grupy 8 elementowe. Na rysunku 3.5 pokazano grupy sąsiednich 8 pól. Z map widać, że dla 4 zmiennych takich grup jest 8 (2 razy tyle co zmiennych). Każdej takiej grupie odpowiada zmienna lub jej negacja. Grupie jedynek funkcji odpowiada zmienna będąca jej współrzędną, a każdej grupie zer funkcji odpowiada negacja współrzędnej. Mniejsze grupy sąsiednie to czwórki, a jeszcze mniejsze to pary. Na rysunku 3.6 pokazano wszystkie możliwe czwórki, a na rysunku 3.7 możliwe pary.



Rysunek 3.5. Grupy ośmiu sąsiednich pól na mapie Karnaugha dla czterech zmiennych

Na rysunku 3.6 przedstawiono grupy czterech sąsiednich pól. Każda z grup odpowiada dwuargumentowemu iloczynowi (jeśli obejmuje jedynki funkcji) lub dwuargumentowej sumie (jeśli obejmuje zera funkcji). Takich grup jest 24 dla funkcji 4 zmiennych. Na rysunku 3.6a pokazano grupy, które obejmując jedynki funkcji odpowiadają (od góry rysunku) składnikom postaci sumacyjnej: $\bar{x}_1\bar{x}_0$, $\bar{x}_1x_0$, x_1x_0 i $x_1\bar{x}_0$. Jeśli grupy te obejmują zera funkcji, to odpowiadają one czynnikom postaci iloczynowej: (x_1+x_0), $(x_1+\bar{x}_0)$, $(\bar{x}_1+\bar{x}_0)$ i $(\bar{x}_1+x_0)$. Na rysunku 3.6b przedstawiono 4 grupy pól, które obejmując jedynki funkcji odpowiadają (od lewej do prawej) składnikom postaci sumacyjnej: $\bar{x}_3\bar{x}_2$, $\bar{x}_3x_2$, x_3x_2 i $x_3\bar{x}_2$, a obejmując zera funkcji odpowiadają czynnikom postaci iloczynowej: (x_3+x_2), $(x_3+\bar{x}_2)$,



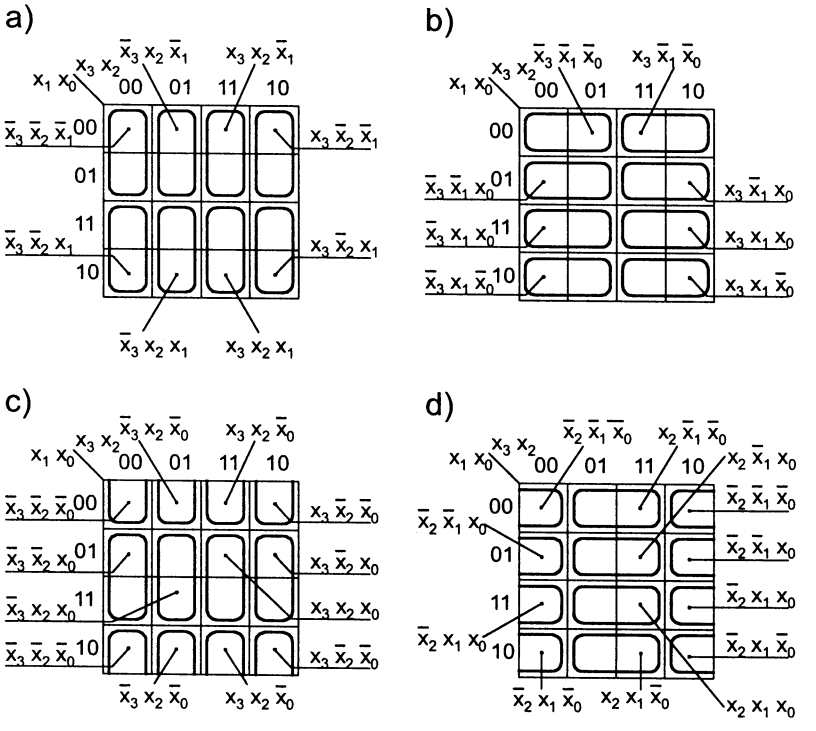
Rysunek 3.6. Grupy czterech sąsiednich pól na mapie Karnaugha dla czterech zmiennych

$(\bar{x}_3 + \bar{x}_2)$ i $(\bar{x}_3 + x_2)$. Na rysunku 3.6c przedstawiono 4 grupy pól, które obejmując jedynki funkcji odpowiadają składnikom postaci sumacyjnej: $\bar{x}_3\bar{x}_0$, $\bar{x}_3x_0$, x_3x_0 i $x_3\bar{x}_0$, a jeśli obejmują zera funkcji, to odpowiadają one czynnikom postaci iloczynowej: $(x_3 + x_0)$, $(x_3 + \bar{x}_0)$, $(\bar{x}_3 + \bar{x}_0)$ i $(\bar{x}_3 + x_0)$.

Na rysunku 3.6d przedstawiono 4 grupy pól, które obejmując jedynki funkcji odpowiadają składnikom postaci sumacyjnej: $\bar{x}_3\bar{x}_1$, $\bar{x}_3x_1$, x_3x_1 i $x_3\bar{x}_1$, a obejmując zera funkcji odpowiadają czynnikom postaci iloczynowej: $(x_3 + x_1)$, $(x_3 + \bar{x}_1)$, $(\bar{x}_3 + \bar{x}_1)$ i $(\bar{x}_3 + x_1)$. Na rysunku 3.6e przedstawiono 4 grupy pól, które obejmując jedynki funkcji odpowiadają składnikom postaci sumacyjnej: $\bar{x}_2\bar{x}_0$, $\bar{x}_2x_0$, x_2x_0 i $x_2\bar{x}_0$, a obejmując zera funkcji odpowiadają czynnikom postaci iloczynowej: $(x_2 + x_0)$, $(x_2 + \bar{x}_0)$, $(\bar{x}_2 + \bar{x}_0)$ i $(\bar{x}_2 + x_0)$. Na rysunku 3.6f przedstawiono 4 grupy pól, które obejmując jedynki funkcji odpowiadają składnikom postaci sumacyjnej: $\bar{x}_2\bar{x}_1$, $\bar{x}_2x_1$, x_2x_1 i $x_2\bar{x}_1$, a obejmując zera funkcji odpowiadają czynnikom postaci iloczynowej: $(x_2 + x_1)$, $(x_2 + \bar{x}_1)$, $(\bar{x}_2 + \bar{x}_1)$ i $(\bar{x}_2 + x_1)$.

Na rysunku 3.7 pokazano wszystkie możliwe pary sąsiednich pól. Par tych jest 32. Każda para odpowiada 3-czynnikowemu iloczynowi (jeśli obejmuje jedynki funkcji) lub 3-składnikowej sumie, jeśli obejmuje zera funkcji. Wyrażenia boolowskie odpowiadające parom na rysunku 3.7a nie zawierają zmiennej x_0, na rysunku 3.7b nie zawierają zmiennej x_2, na rysunku 3.7c nie zawierają zmiennej x_1, a na rysunku 3.7d zmiennej x_3.

Wprawne wyszukiwanie grup zer i jedynek jest warunkiem niezbędnym efektywnego projektowania układów logicznych.



Rysunek 3.7. Pary sąsiednich pól na mapie Karnaugha dla czterech zmiennych

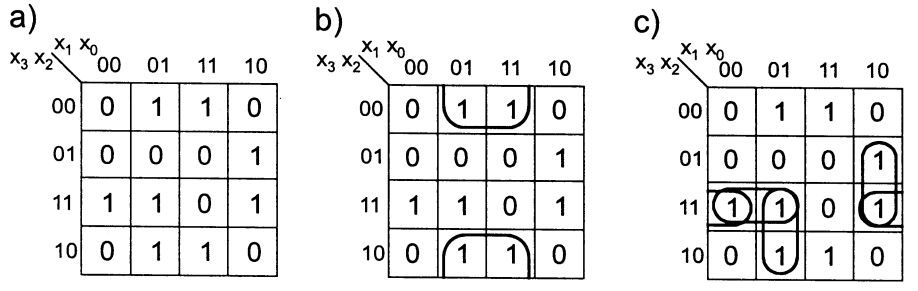
W podanych dalej algorytmach minimalizacji funkcji używamy pojęcia: **implikantu** i **implicentu**. Implikantem funkcji f nazywać będziemy inną funkcję g tych samych zmiennych, która dla wszystkich zer funkcji f przyjmuje wartość 0. Wynika stąd, że funkcja g może mieć więcej miejsc zerowych a mniej miejsc jedynkowych od funkcji f. Na przykład funkcja 4 zmiennych $y = \Sigma(2, 3, 7)$ ma 6 implikantów: $g_1 = \Sigma(2, 3)$, $g_2 = \Sigma(2, 7)$, $g_3 = \Sigma(3, 7)$, $g_4 = \Sigma(2)$, $g_5 = \Sigma(3)$ i $g_6 = \Sigma(7)$. Trzy ostatnie implikanty (zawierające tylko jedną jedynkę) nazywać będziemy implikantami prostymi. Implikanty są funkcjami, które „pokrywają" niektóre jedynki funkcji f. Natomiast implicentem funkcji f nazywać będziemy inną funkcję h tych samych zmiennych, która dla wszystkich jedynek funkcji f przyjmuje wartość 1. Wynika stąd, że funkcja h może mieć więcej miejsc jedynkowych i mniej miejsc zerowych od funkcji f. Na przykład funkcja 4 zmiennych $y = \Pi(2, 3, 7)$ ma 6 implicentów: $h_1 = \Pi(2, 3)$, $h_2 = \Pi(2, 7)$, $h_3 = \Pi(3, 7)$, $h_4 = \Pi(2)$, $h_5 = \Pi(3)$ i $h_6 = \Pi(7)$. Trzy ostatnie implicenty (zawierające tylko jedno zero) nazywać będziemy implicentami prostymi. Implicenty są funkcjami, które „pokrywają" niektóre zera funkcji f.

Procedura minimalizacji funkcji na mapie Karnaugha składa się z kilku kroków:

1. Sprawdzenie czy funkcja nie jest trywialna, tj. czy nie składa się z samych zer lub samych jedynek. Jeżeli tak jest, to funkcja jest stała i jest równa 1 (same jedynki) lub 0 (same zera). Jeżeli nie zachodzi taki przypadek trywialny, to należy przejść do punktu 2.
2. Poszukując tylko jednego z dwu rozwiązań można posłużyć się kryterium liczby jedynek funkcji. Jeśli funkcja zawiera mniej jedynek niż zer, to warto szukać postaci sumacyjnej (wówczas wykonać pkt 3), jeśli natomiast jest mniej zer, to szukać postaci iloczynowej (wykonać punkt 8). Najlepiej wyznaczyć oba rozwiązania i porównać (por. pkt 13).
3. Wyszukać wszystkie grupy jedynek o liczności 2^{n-1}. Dla przypadku 4 zmiennych są to ósemki. Wszystkie znalezione grupy są implikantami danej funkcji. Pośród nich wyróżnia się implikanty zasadnicze, tj. takie, które zawierają co najmniej jedną jedynkę nie pokrytą przez inny implikant. Implikanty zasadnicze na pewno wchodzą w skład minimalnej postaci sumacyjnej funkcji, a implikanty niezasadnicze mogą wchodzić wariantowo (p. przykład dalej).
4. Dla nie pokrytych jedynek wyszukać wszystkie grupy jedynek o liczności 2^{n-2}. Dla przypadku 4 zmiennych są to czwórki. Podobnie jak w pkt 3 wszystkie znalezione-implikanty zasadnicze wchodzą w skład minimalnej postaci sumacyjnej funkcji, a implikanty niezasadnicze mogą wchodzić wariantowo (por. przykład dalej).
5. Dla nie pokrytych jedynek wyszukać wszystkie grupy jedynek o liczności 2^{n-3}. Dla przypadku 4 zmiennych są to pary. Implikanty zasadnicze wchodzą w skład minimalnej postaci sumacyjnej funkcji, a implikanty niezasadnicze mogą wchodzić wariantowo (por. przykład dalej).
6. Dla nie pokrytych jedynek wyszukać wszystkie grupy jedynek o liczności 2^{n-4}. Dla przypadku 4 zmiennych są to pojedyncze jedynki. Wszystkie one są implikantami zasadniczymi danej funkcji, a więc wchodzą w skład minimalnej postaci sumacyjnej funkcji.
7. Wyznaczyć wszystkie możliwe minimalne postacie sumacyjne danej funkcji.
8. Wyszukać wszystkie grupy zer o liczności 2^{n-1}. Dla przypadku 4 zmiennych są to ósemki. Wszystkie znalezione grupy są implicentami danej funkcji. Pośród nich wyróżnia się implicenty zasadnicze, tj. takie które zawierają co najmniej jedno zero nie pokryte przez inny implicent. Implicenty zasadnicze wchodzą w skład minimalnej postaci iloczynowej funkcji, a implicenty niezasadnicze mogą wchodzić wariantowo (por. przykład dalej).

9. Dla nie pokrytych zer wyszukać wszystkie grupy zer o liczności 2^{n-2}. Dla przypadku 4 zmiennych są to czwórki. Wyznaczyć implicenty zasadnicze i one wchodzą w skład minimalnej postaci iloczynowej funkcji. Pozostałe wchodzą wariantowo.
10. Dla nie pokrytych zer wyszukać wszystkie grupy zer o liczności 2^{n-3}. Dla przypadku 4 zmiennych są to pary. Wyznaczyć implicenty zasadnicze i one będą wchodzić w skład minimalnej postaci iloczynowej funkcji. Pozostałe pary wchodzą wariantowo.
11. Dla nie pokrytych zer wyszukać wszystkie grupy zer o liczności 2^{n-4}. Dla przypadku 4 zmiennych są to pojedyncze zera. Wszystkie one są implicentami zasadniczymi i wchodzą w skład minimalnej postaci iloczynowej funkcji.
12. Wyznaczyć wszystkie minimalne postacie iloczynowe danej funkcji.
13. Porównać rozwiązania z pkt 7 i 12 i wybrać najlepsze, tj. zawierające najmniejszą liczbę bramek.

Rozpatrzmy podany algorytm na kilku przykładach.



Rysunek 3.8. Mapy Karnaugha przykładowej funkcji

Przykład 3.1. Zaprojektować układ kombinacyjny realizujący funkcję boolowską czterech zmiennych daną w postaci dziesiętnej $y = \Sigma(1, 3, 6, 9, 11, 12, 13, 14)$.

Rozwiązanie. Na rysunku 3.8a pokazano mapy Karnaugha danej funkcji, a na rysunkach 3.8b i 3.8c pokazano wyszukiwanie grup jedynek tej funkcji. Algorytm minimalizacji funkcji jest następujący:

Ad 1. Funkcja nie jest trywialna i zawiera zarówno zera jak i jedynki.

Ad 2. Funkcja ma 8 jedynek i 8 zer. Rozpatrzmy najpierw postać sumacyjną.

Ad 3. Nie ma grup ósemek jedynek.

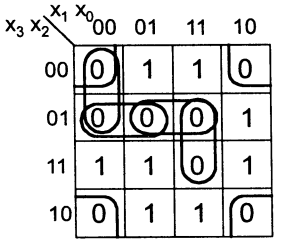
Ad 4. Jest jedna czwórka jedynek (rys. 3.8b) — $\bar{x}_2 x_0$ i ona wejdzie do postaci minimalnej.

Ad 5. Dla nie pokrytych przez czwórkę jedynek można znaleźć 4 pary jedynek pokazane na rysunku 3.8c. Są to: $x_3 x_2 \bar{x}_1$, $x_3 x_2 \bar{x}_0$, $x_3 \bar{x}_1 x_0$ i $x_2 x_1 \bar{x}_0$. Ostatnia para $x_2 x_1 \bar{x}_0$ jest implikantem zasadniczym, gdyż tylko ona pokrywa jedynkę $\bar{x}_3 x_2 x_1 \bar{x}_0$.

Ad 6. Nie ma już nie pokrytych jedynek.

Ad 7. Implikanty zasadanicze (czwórka $\bar{x}_2 x_0$ i para $x_2 x_1 \bar{x}_0$) pokrywają 6 jedynek funkcji, które odpowiadają dziesiętnym kombinacjom zmiennych: 1, 3, 6, 9, 11 i 14. Nie pokryte przez te implikanty jedynki funkcji, to 12 i 13. Do ich pokrycia wystarcza jedna para $x_3 x_2 \bar{x}_1$. Zatem jest jedna sumacyjna postać minimalna tej funkcji $y = \bar{x}_2 x_0 + x_2 x_1 \bar{x}_0 + x_3 x_2 \bar{x}_1$.

Ad 8. Nie ma grup ósemek zer.



Rysunek 3.9. Mapy Karnaugha przykładowej funkcji

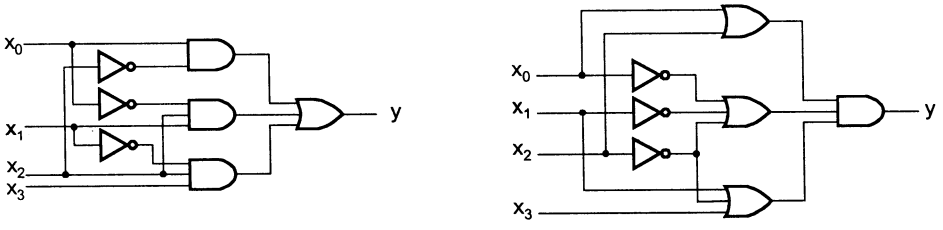
Ad 9. Jest jedna czwórka zer (na rysunku 3.9 narożniki mapy Karnaugha) — $(x_2 + x_0)$.

Ad 10. Cztery nie pokryte zera tworzą cztery pary: $x_3 + x_1 + x_0$, $x_3 + \bar{x}_2 + x_1$, $x_3 + \bar{x}_2 + \bar{x}_0$ i $\bar{x}_2 + \bar{x}_1 + \bar{x}_0$. Ostatnia para $\bar{x}_2 + \bar{x}_1 + \bar{x}_0$ jest implicentem zasadniczym, gdyż tylko ona pokrywa zero $\bar{x}_3 + \bar{x}_2 + \bar{x}_1 + \bar{x}_0$.

Ad 11. Jest jedna iloczynowa postać minimalna danej funkcji:

$$y = (x_2 + x_0)(\bar{x}_2 + \bar{x}_1 + \bar{x}_0)(x_3 + \bar{x}_2 + x_1)$$

Ad 12. Dla porównania obu otrzymanych postaci minimalnych zrealizujmy je na bramkach sumy, iloczynu i negacji.



Rysunek 3.10. Dwa równorzędne rozwiązania zadania z przykładu 3.1

Na rysunku 3.10 pokazano dwie realizacje przykładowej funkcji: jedną opartą na postaci sumacyjnej i drugą na postaci iloczynowej. Obie realizacje wymagają 4 bramek o takiej samej liczbie wejść. W pierwszym rozwiązaniu są trzy bramki trzywejściowe (jedna sumy i dwie iloczynu) oraz jedna dwuwejściowa bramka iloczynu. W drugim rozwiązaniu są trzy bramki trzywejściowe (dwie sumy i jedna iloczynu) oraz jedna dwuwejściowa bramka sumy. Dla kryterium minimalnej liczby bramek oba rozwiązania są równoważne. ⊠

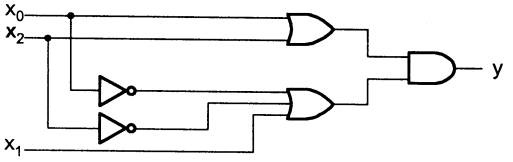
Przykład 3.2. Zaprojektować układ realizujący funkcję opisaną mapą Karnaugha z rysunku 3.2.

Rozwiązanie. Zadana funkcja jest funkcją trzech argumentów a jej mapa Karnaugha jest pokazana na rysunku 3.11. Ponieważ funkcja zawiera mniej zer niż jedynek, to zgodnie z podanym algorytmem należy rozpatrzyć najpierw grupy sąsiednich zer. Łatwo zauważyć, że są tylko 3 zera, więc nie trzeba poszukiwać czwórek, a istnieje tylko jedna para. Stąd minimalną postać iloczynową można określić od razu jako: $y = (x_0 + x_2)(\bar{x}_0 + x_1 + \bar{x}_2)$. Zatem rozwiązaniem na bramkach sumy, iloczynu i negacji jest układ pokazany na rysunku 3.12 składający się z 3 bramek: dwóch bramek sumy (jedna dwu- i druga trzywejściowa) i dwuwejściowego iloczynu.

x_0 \ x_2x_1	00	01	11	10
0	0_0	0_2	1_6	1_4
1	1_1	1_3	1_7	0_5

Rysunek 3.11. Mapa Karnaugha funkcji opisanej za pomocą tablicy prawdy (tabl. 3.2)

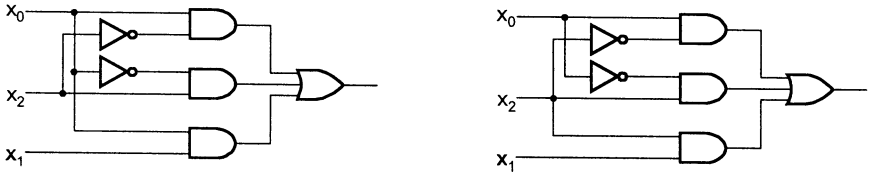
Rozpatrzmy teraz jedynki danej funkcji. Dla funkcji tej nie istnieje grupa sąsiednich czterech jedynek, a istnieją 4 pary: (1, 3), (4, 6), (3, 7) oraz (6, 7). Widać, że jedynkę funkcji o numerze 1 można zrealizować wyłącznie używając pary (1, 3), a jedynkę o numerze 4 można zrealizować wyłącznie używając pary (4, 6). Zatem pary (1, 3) i (4, 6) są implikantami zasadniczymi. Te dwie pary pokrywają cztery jedynki funkcji. Ostatnią piątą jedynkę (7) można pokryć parą (3, 7) lub (6, 7) i dlatego otrzymuje się dwie równoważne postacie sumacyjne i dwa rozwiązania: pierwsze $y_1 = x_0\bar{x}_2 + \bar{x}_0x_2 + x_0x_1$ i drugie $y_2 = x_0\bar{x}_2 + \bar{x}_0x_2 + x_1x_2$.



Rysunek 3.12. Realizacja postaci iloczynowej funkcji z przykładu 3.2

Układy realizujące te dwie postacie pokazano na rysunku 3.13. Oba rozwiązania wymagają 6 bramek, a więc więcej niż poprzedni układ realizujący postać iloczynową. ⊠

Często zachodzi przypadek, że dla danej kombinacji zmiennych wejściowych funkcja jest nieokreślona (ang. *don't care condition*). Przypadki takie opisuje się w postaci dziesiętnej podając nieokreślone kombinacje zmiennych wejściowych w drugim nawiasie, a na mapie Karnaugha umieszcza się nieokreśloności funkcji wpisując w odpowiednie kratki kreskę zamiast zera lub jedynki pokazując w ten sposób, że można wstawić tam zarówno 0 jak i 1.



Rysunek 3.13. Dwie równoważne postacie sumacyjne

Przykład 3.3. Zaprojektować układ kombinacyjny realizujący funkcję boolowską czterech zmiennych daną w postaci dziesiętnej: $y = \Sigma(2, 4, 7, 12, 14)(3, 6)$.

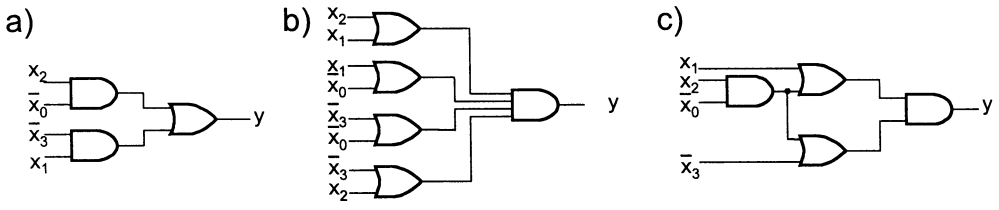
Rozwiązanie. Zgodnie z wcześniejszymi rozważaniami najpierw wyznaczymy minimalną postać sumacyjną, a następnie iloczynową.

1. Poszukujemy grup ósemek sąsiednich jedynek — rozpatrywana funkcja nie ma takich grup.

$x_3 x_2$ \ $x_1 x_0$	00	01	11	10
00	0	0	—	1
01	1	0	1	—
11	1	0	0	1
10	0	0	0	0

Rysunek 3.14. Mapa Karnaugha funkcji z przykładu 3.3

2. Poszukujemy grup czwórek sąsiednich jedynek — z mapy Karnaugha widać, że nie ma takich grup, ale przyjmując jedynki zamiast kresek widać, że można utworzyć dwie czwórki: (2, 3, 6, 7) oraz (4, 6, 12, 14) pokrywające wszystkie jedynki funkcji. Dlatego postać sumacyjna składać się będzie z dwuwejściowej bramki sumy i dwóch dwuwejściowych bramek iloczynów.
3. Wyznaczając postać iloczynową poszukujemy grup ósemek zer — rozpatrywana funkcja nie ma takich grup.
4. Poszukujemy grup czwórek sąsiednich zer — z mapy Karnaugha widać, że są cztery takie grupy: (0, 1, 8, 9), (1, 5, 9, 13), (9, 11, 13, 15) i (8, 9, 10, 11). Pokrywają one wszystkie zera funkcji, ale każda z nich jest implicentem zasadniczym, a więc postać iloczynowa tej funkcji składać się będzie z czterowejściowej bramki iloczynu oraz czterech dwuwejściowych bramek sum.
5. Z rysunków 3.15a i 3.15b, na których pokazano oba rozwiązania widać, że minimalnym rozwiązaniem przykładu jest wynik z pkt 2. ⊠



Rysunek 3.15. Układy realizujące funkcję z przykładu 3.3 postać: a) sumacyjna, b) iloczynowa, c) po faktoryzacji

Rozpatrzmy wyrażenie boolowskie opisujące układ pokazany na rysunku 3.15b:

$$y = (x_2 + x_1)(x_1 + \overline{x}_0)(\overline{x}_3 + \overline{x}_0)(\overline{x}_3 + x_2)$$

Można zauważyć, że stosując prawa algebry Boole'a podane w pkt. 3.1.2, wyrażenie to można przekształcić do postaci:

$$y = (x_1 + x_2\overline{x}_0)(\overline{x}_3 + \overline{x}_0 x_2)$$

Przekształcenie minimalnej postaci iloczynowej jeszcze bardziej uprościło dane wyrażenie. Realizacja na bramkach tej postaci jest pokazana na rysunku 3.15c. Wykonana czyn-

ność nosi nazwę **faktoryzacji**. Pokazany układ ma mniejszą liczbę bramek niż układ na rysunku 3.15b. Ten zysk okupiono stratą w szybkości działania układu. Układ pokazany rysunku 3.15b, jak i wszystkie inne dotychczas pokazywane układy, był tzw. układem dwustopniowym. Sygnały pojawiające się na wejściach układu propagują przez dwa stopnie bramek. Pierwszym stopniem są bramki iloczynu dla sumacyjnej postaci funkcji, a drugim stopniem jest bramka sumy. Dla postaci iloczynowej pierwszym stopniem są bramki sumy, a drugim stopniem jest bramka iloczynu. Oznaczając czas propagacji przez jedną bramkę jako t_p otrzyma się, że czas opóźnienia wnoszony przez układ dwupoziomowy wynosi $2t_p$. Układ otrzymany drogą faktoryzacji jest wolniejszy, gdyż jego czas opóźnienia wynosi $3t_p$. Poszukiwanie minimalnego układu drogą faktoryzacji jest procesem heurystycznym, tj. zależy od spostrzegawczości i wprawy projektanta.

3.2.2. Układy iteracyjne

Zdarza się, że w rozwiązywanym zadaniu nie jest określona liczba zmiennych. Można wtedy próbować takiej dekompozycji zadania, aby zaprojektować układ np. dla jednej lub kilku zmiennych, a następnie łączyć takie układy ze sobą. Układy takie nazywać będziemy **układami iteracyjnymi**. Zaletą układów iteracyjnych jest możliwość ich łatwego rozbudowywania dla większej liczby zmiennych. Ich wadą natomiast jest fakt, że nie stanowią one minimalnego rozwiązania pod względem liczby bramek. Drugą ich wadą jest to, że wnoszą znacznie większe opóźnienie niż mogą wnosić układy projektowane w sposób klasyczny.

Dalej przedstawiono przykładowe problemy, które stosunkowo łatwo można zdekomponować i rozwiązać w postaci układów iteracyjnych:

1. Dla n-bitowego słowa znaleźć najbardziej (najmniej) znaczącą jedynkę (zero).
2. Zaprojektować układ generujący następnik (poprzednik) n-bitowej liczby.
3. Zaprojektować układ sumujący dwie liczby w kodzie NKB.
4. Zaprojektować układ odejmujący dwie liczby w kodzie NKB.
5. Porównać dwie n-bitowe liczby.
6. Zaprojektować układ przepuszczający na wyjście wszystkie bity leżące pomiędzy skrajnymi zerami (jedynkami) w n-bitowym słowie wejściowym.
7. Zaprojektować układ, który w n-bitowym słowie wejściowym wykrywa grupy sąsiadujących ze sobą co najmniej m jedynek (zer) i zastępuje je negacjami wartości tych bitów, a na pozostałych pozycjach pozostawia wartości bitów.

Niektóre z tych problemów zostaną rozwiązane, a niektóre poleca się Czytelnikowi do samodzielnego rozwiązania.

Przykład 3.4. Zaprojektować układ arytmetycznego dodawania dwóch n-bitowych liczb zapisanych w naturalnym kodzie binarnym NKB.

Rozwiązanie. Niech będą dane dwie liczby A i B przedstawione na n pozycjach binarnych w naturalnym kodzie dwójkowym:

$$A = \sum_{i=0}^{n-1} a_i 2^i \qquad B = \sum_{i=0}^{n-1} b_i 2^i$$

Dodawanie dwóch liczb n-bitowych rozpoczyna się od dodawania bitów na najmniej znaczących pozycjach. Jak przedstawiono w rozdz. 2, w przypadku gdy są jedynkami powstaje **przeniesienie** (ang. *carry*) na bardziej znaczącą pozycję wyniku. Dlatego na tej pozycji i na wszystkich następnych, oprócz bitów wejściowych, trzeba uwzględniać ewentual-

Tablica 3.4. Tablica prawdy jednopozycyjnego sumatora

$a_ib_ic_i$	y_i	c_{i+1}	$a_ib_ic_i$	y_i	c_{i+1}
000	0	0	001	1	0
100	1	0	101	0	1
010	1	0	011	0	1
110	0	1	111	1	1

ne przeniesienia z sąsiednich mniej znaczących pozycji. Dlatego funkcje boolowskie sumatora jednopozycyjnego są funkcjami trzech zmiennych: wartości danych pozycji a_i i b_i oraz przeniesienia z mniej znaczącej pozycji c_i. Operacja jednopozycyjnego dodawania (dodawania dwóch bitów wraz z przeniesieniem) polega na obliczeniu wartości dwóch funkcji. Pierwszą z nich jest funkcja y_i będąca binarnym wynikiem dodawania trzech bitów a_i, b_i i c_i i drugą jest funkcja c_{i+1} tych samych zmiennych określająca wartość przeniesienia na następną pozycję. W tablicy 3.4 przedstawiono tablice prawdy obu tych funkcji. Układ dwuwyjściowy realizujący podaną tablicę nazywa się sumatorem jednopozycyjnem. Rozwiązanie przykładowego zadania polega na złożeniu (por. rozwiązanie tego zadania) sumatora wielopozycyjnego z wielu jednopozycyjnych sumatorów.

Projekt sumatora jednopozycyjnego

Kanoniczną postać sumacyjna funkcji y_i określa się jako

$$y_i = \overline{a_i}\,\overline{b_i}c_i + \overline{a_i}b_i\overline{c_i} + a_i\overline{b_i}\,\overline{c_i} + a_ib_ic_i$$

Taką postać można zrealizować za pomocą układu składającego się z 5 bramek: 4-wejściowej bramki sumy logicznej i czterech trzywejściowych bramek iloczynów logicznych. Do minimalizacji tej funkcji posłużymy się mapą Karnaugha pokazaną na rysunku 3.16a.

a)

c_i \ a_ib_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

y_i

b)

c_i \ a_ib_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

c_{i+1}

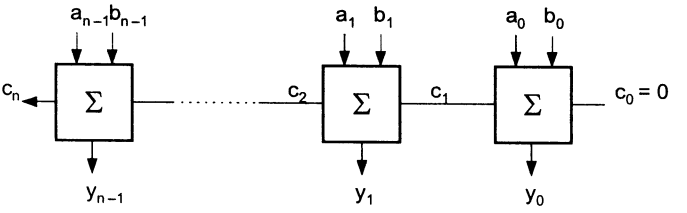
Rysunek 3.16. Mapy Karnaugha sumatora jednopozycyjnego

Dla funkcji y_i nie istnieją ani czwórki zer bądź jedynek, ani pary zer bądź jedynek. W związku z tym minimalną postacią sumacyjną i iloczynową są postacie kanoniczne. Można natomiast przeprowadzić faktoryzację i otrzymamy wówczas

$$y_i = \overline{a_i}(\overline{b_i}c_i + b_i\overline{c_i}) + a_i(\overline{b_i}\,\overline{c_i} + b_ic_i)$$

Stąd widać, że funkcję można przedstawić jako $y_i = \overline{a_i}(b_i \oplus c_i) + a_i\overline{(b_i \oplus c_i)}$, a stąd już łatwo pokazać, że $y_i = a_i \oplus b_i \oplus c_i$. Jest to rozwiązanie wykorzystujące dwie bramki EXOR.

Mapa Karnaugha funkcji c_{i+1} jest pokazana na rysunku 3.16b. Cztery jedynki funkcji i cztery zera są tak ułożone, że stanowią trzy pary, z których wszystkie są implikantami zasadniczymi. Dlatego, w sensie liczby bramek realizacja postaci sumacyjnej i iloczynowej



Rysunek 3.17. Iteracyjny sumator n-bitowy

da równoważne rozwiązania. Rozwiązanie realizujące minimalną postać sumacyjną, to $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$. Dla realizacji takiej postaci potrzeba jednej trójwejściowej bramki sumy oraz trzech dwuwejściowych bramek iloczynów.

Projekt sumatora wielopozycyjnego

Układ iteracyjny będący sumatorem n-pozycyjnym pokazano na rysunku 3.17. Jest to układ złożony z n sumatorów jednopozycyjnych. Układ ma $2n+1$ wejść: wejścia dwóch n-bitowych liczb i wejście przeniesienia c_0. Układ ma $n+1$ wyjść: n wyjść wyniku sumowania i wyjście przeniesienia c_n. Wejście przeniesienia c_0 i wyjście c_n służą do ewentualnej dalszej rozbudowy, czyli na przykład do połączenia dwóch takich układów ze sobą celem uzyskania sumatora $2n$-bitowego.

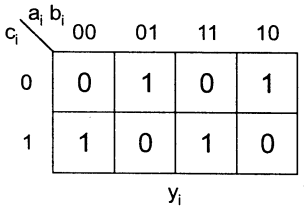
W zaprojektowanym sumatorze jednopozycyjnym sygnały wejściowe propagują przez dwa poziomy bramek (funkcja y_i jest zbudowana na bramkach EXOR, a funkcja c_{i+1} nie zawiera inwerterów). Dlatego czas opóźnienia wnoszony przez sumator jednopozycyjny wynosi $2t_p$. Stąd n-bitowy sumator iteracyjny wnosi opóźnienie wynoszące $2nt_p$. Często projektantom zależy na zbudowaniu szybkich sumatorów. Wtedy stosują oni rozwiązania iteracyjne, ale projektują dodatkowe układy przyśpieszające. Układy takie pokazano w dalszej części książki. ⊠

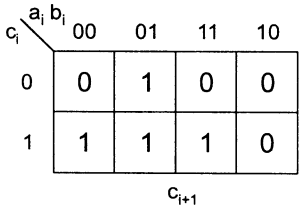
Przykład 3.5. Zaprojektować układ następnika słowa n-bitowego.

Rozwiązanie. Następnik danej liczby A można obliczyć posługując się, wcześniej omówioną procedurą dodawania. Wyznaczając $A + B$ przy założeniu, że $B = 0$ i $c_0 = 1$ otrzymamy, że funkcja wyjściowa i-tego bloku, wynosi $y_i = \text{EXOR}(a_i, c_i)$ a przeniesienie $c_{i+1} = a_i c_i$. ⊠

Przykład 3.6. Zaprojektować układ odejmowania dwóch n-bitowych liczb wejściowych danych w kodzie NKB: odjemnej $A = (a_{n-1}, a_{n-2}, ..., a_0)$ i odjemnika $B = (b_{n-1}, b_{n-2}, ..., b_0)$.

Rozwiązanie. W rozdziale 2 podano tablicę definiującą odejmowanie dwóch liczb w kodzie NKB. Przyjmując strukturę układu jak na rysunku 3.17 trzeba określić funkcję y_i





Rysunek 3.18. Mapy Karnaugha funkcji y_i i c_{i+1} dla pojedynczego bloku iteracyjnego układu odejmowania

i przeniesienie c_i. Na rysunku 3.18 przedstawiono mapy Karnaugha tych funkcji. Porównując funkcję y_i układu odejmowania z funkcją y_i sumatora widać, że są to identyczne funkcje. Różnice w układach występują w definicjach przeniesienia. Dla układu odejmującego przeniesienie jest określone funkcją $y_i = \overline{a}_i b_i + \overline{a}_i c_i + b_i c_i$. ⊠

Przykład 3.7. Zaprojektować układ wykrywający w n-bitowym słowie grupy trzech sąsiadujących ze sobą jedynek. Na n wyjściach układu pojawiają się jedynki i tylko pierwsza jedynka z wykrytej grupy jest zamieniana na zero.

Rozwiązanie. Projektując układ iteracyjny trzeba określić liczbę zmiennych i liczbę funkcji realizowanych przez pojedynczy blok, a także liczbę przeniesień pomiędzy blokami. W projektowanym układzie występuje proces zliczania (do trzech), więc przeniesienia pomiędzy blokami muszą wskazywać aktualną liczbę jedynek w grupie. Dlatego w jednym z możliwych kierunków iteracji jako przeniesienie występuje para bitów a_i i b_i wskazująca jeden z 4 przypadków: gdy jest

— 00, to poprzednia pozycja była zerem (brak grupy jedynek),
— 01, to tylko jedna poprzednia pozycja była jedynką,
— 10, to dwie poprzednie pozycje były jedynkami,
— 11, to więcej niż dwie poprzednie pozycje były jedynkami.

W układzie musi pojawić się jeszcze jeden sygnał przeniesienia i to w przeciwnym kierunku iteracji niż analizowane do tej pory. Sygnał ten, oznaczony przez q_i, jest konieczny ponieważ po wykryciu grupy trzech jedynek należy na odpowiednie wyjścia układu przesłać zanegowane bity słowa wejściowego, tj. zera. Przeniesienie to propaguje aż do napotkania bloku, na którego wejściu jest zero.

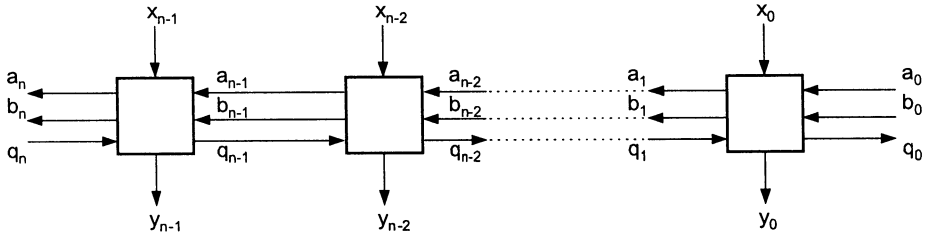
Każdy blok projektowanego układu ma 4 wejścia x_i, a_i, b_i i q_{i+1}. Na rysunku 3.19 pokazano strukturę projektowanego układu, a w tablicy 3.5 zamieszczono tablice prawdy wszystkich czterech funkcji opisujących jeden blok przykładowego układu. Obok wierszy tablicy prawdy podano komentarz celem łatwiejszego analizowania układu przez Czytelnika.

Funkcje mają nieokreśloności, gdyż nie może zajść przypadek, że $q_{i+1} = 1$ (co świadczy o wykryciu grupy jedynek, a w tym x_i równe 1) i jednocześnie $x_i = 0$.

Jeśli na danej pozycji $x_i = 1$, to generowana para $a_{i+1}b_{i+1}$ zwiększa wskazanie o 1, za wyjątkiem przypadku gdy oba bity a_i i b_i są jedynkami.

Wartości skrajne sygnałów przeniesień powinny spełniać warunki: $a_0 = 0$, $b_0 = 0$ i $q_n = 1$.

Po narysowaniu map Karnaugha można wyznaczyć minimalne postacie sumacyjne czterech omówionych funkcji:



Rysunek 3.19. Struktura układu iteracyjnego dla przykładu 3.7

Tablica 3.5. Tablice prawdy opisujące funkcje z przykładu 3.7

wejścia				wyjścia				
x_i	q_{i+1}	a_i	b_i	y_i	q_i	a_{i+1}	b_{i+1}	
0	0	0	0	1	0	0	0	brak sąsiednich jedynek
0	0	0	1	1	0	0	0	jedna sąsiednia jedynka
0	0	1	0	1	0	0	0	dwie sąsiednie jedynki
0	0	1	1	1	1	0	0	q_i zerowanie pozycji i –1
0	1	0	0	—	—	—	—	przychodzące zerowanie jest
0	1	0	1	—	—	—	—	nieprawidłowe bo $q_{i+1} = 1$
0	1	1	0	—	—	—	—	świadczy o wykryciu grupy
0	1	1	1	—	—	—	—	jedynek, a na wejściu jest 0.
1	0	0	0	1	0	0	1	pierwsza jedynka grupy
1	0	0	1	1	0	1	0	druga jedynka z grupy
1	0	1	0	1	1	1	1	trzecia jedynka z grupy
1	0	1	1	1	1	1	1	więcej niż trzecia jedynka
1	1	0	0	0	0	0	1	pierwsza jedynka z grupy
1	1	0	1	1	0	1	0	druga jedynka z grupy
1	1	1	0	1	1	1	1	trzecia jedynka z grupy
1	1	1	1	1	1	1	1	więcej niż trzecia jedynka

$$y_i = \overline{q}_{i+1} + a_i$$
$$q_i = a_i b_i + a_i x_i + b_i q_{i+1}$$
$$a_i = a_i x_i + \overline{b}_i x_i$$
$$b_i = a_i x_i + \overline{b}_i x_i \quad \boxtimes$$

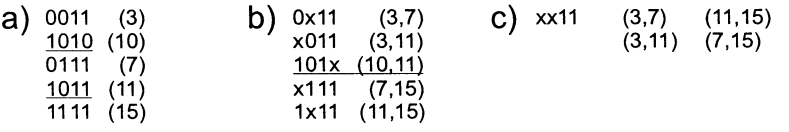
3.2.3. Minimalizacja funkcji metodą Quine'a-McCluskeya

Graficzna metoda minimalizacji układów logicznych za pomocą mapy Karnaugha wymaga od projektanta pewnej wprawy. Ale nawet wtedy udaje się projektować tą metodą jedynie proste układy. Lepszym rozwiązaniem jest metoda algorytmiczna, która umożliwia projektowanie wspomagane komputerowo. Poniżej zostanie zaprezentowana algorytmiczna metoda zwana metodą Quine'a-McCluskeya. W tym celu posłużymy się odpowiednio prostymi przykładami, aby można było procedurę śledzić na mapie Karnaugha. Metoda Quine'a-McCluskeya składa się z dwóch etapów. Pierwszy polega na tym, aby wyszukać wszystkie możliwe pary sąsiednich jedynek (zer), czwórki, ósemki itd. Następny etap polega na wyznaczeniu implikantów (implicentów) zasadniczych i na tej podstawie możliwych rozwiązań. W pierwszym etapie należy porównywać kombinacje zmiennych i wyszukiwać pary jedynek (zer) funkcji, następnie czwórki itd. Aby proces ten skrócić porównywane są jedynie te kombinacje zmiennych, którym odpowiadają słowa binarne, których liczba jedynek różni się o 1. Jeśli różnica jedynek jest większa, to słowa te różnią się na większej niż 1 liczbie pozycji. Dla funkcji n zmiennych wszystkie słowa binarne odpowiadające jedynkom funkcji dzielone są na $n + 1$ podzbiorów, z których każdy zawiera słowa o takiej samej liczbie jedynek. Dwa z tych podzbiorów (podzbiór z zerową liczbą jedynek i podzbiór z n jedynkami) mogą być co najwyżej jednoelementowe.

Przykład 3.8. Zminimalizować funkcję 4 zmiennych: $y = \Sigma(3, 7, 10, 11, 15)$.

Rozwiązanie. Podzielmy zbiór wszystkich implikantów prostych tej funkcji na pięć podzbiorów. Zbiór nie zawierający jedynek i zawierający jedną jedynkę jest pusty. Niepuste są podzbiory z dwiema jedynkami, z trzema jedynkami i podzbiór z czterema jedynkami. Zbiory te przedstawiono na rysunku 3.20a w tej właśnie kolejności poczynając od góry.

Porównywanie elementów z sąsiednich podzbiorów pozwala znaleźć pary jedynek funkcji, które pokazano na rysunku 3.20b. Wszystkie pary jedynek także podzielono na dwa podzbiory: dwujedynkowe i trzyjedynkowe. W rozpatrywanym przykładzie wszystkie 6 implikantów prostych weszło w skład par, a więc nie będzie implikantów zasadniczych spośród implikantów prostych. Znalezienie ewentualnych czwórek polega znowu na porównywaniu elementów sąsiednich podzbiorów par. W zadanym przykładzie otrzyma się jedną czwórkę $xx11$ z par (3, 7) i (11, 15) pokazaną na rysunku 3.20c. Tę samą czwórkę otrzyma się z par (3, 11) i (7, 15). Zatem implikanty zasadnicze danej funkcji — czwórka $xx11$ i para $101x$, która nie weszła w skład czwórki — pokrywają wszystkie jedynki funkcji. Minimalna liczba iloczynów pokrywająca wszystkie jedynki funkcji nazywana jest minimalnym pokryciem. Określa ono minimalną postać sumacyjną danej funkcji. Minimalna postać sumacyjna przykładowej funkcji to: $y = x_1x_0 + x_3\bar{x}_2x_1$. ⊠



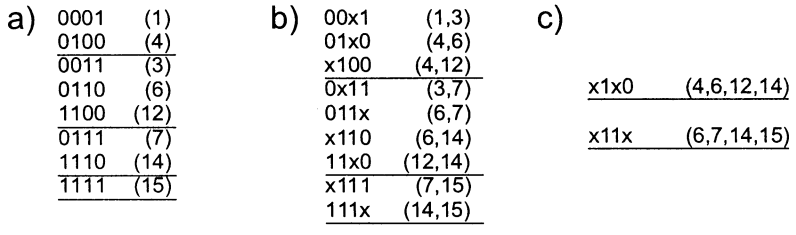
Rysunek 3.20. Metoda poszukiwania wszystkich implikantów danej funkcji

Przykład 3.9. Zminimalizować funkcję czterech zmiennych $y = \Sigma(1, 3, 4, 6, 7, 12, 14, 15)$.

Rozwiązanie. Pierwszy krok algorytmu polega na pogrupowaniu implikantów prostych co pokazano na rysunku 3.21a. Drugi krok algorytmu, to tworzenie par (rysunek 3.21b). W tym przykładzie pośród implikantów prostych nie ma implikantów zasadniczych, gdyż każdy z implikantów protych wchodzi w skład jakiejś pary. Trzeci krok algorytmu polega na wyszukiwaniu czwórek przez porównywanie par z sąsiednich zbiorów. W wyniku tej procedury powstają dwie czwórki pokazane na rysunku 3.21c.

Poszukiwanie ósemek nie daje rezultatu, ponieważ obie czwórki $x1x0$ i $x11x$ różnią się na trzech pozycjach i dlatego nie tworzą ósemki. Otrzymano 4 implikanty pokrywające wszystkie jedynki danej funkcji: dwie czwórki i dwie pary ($00x1$ i $0x11$) nie wchodzące do czwórek. ⊠

W obu przedstawionych przykładach łatwo było znaleźć minimalne pokrycie wszystkich jedynek danej funkcji. Często jednak możliwych rozwiązań jest wiele, a minimalne pokrycie niełatwe do znalezienia. Wtedy wykonuje się drugi etap minimalizacji, który ma wyłonić najlepsze pokrycie wszystkich jedynek funkcji. Robi się to przez utworzenie tzw. tablicy



Rysunek 3.21. Minimalizacja funkcji z przykładu 3.9

Tablica 3.6. Tablica Quine'a implikantów z przykładu 3.9

	x1x0	x11x	00x1	0x11
0001	—	—	Ⓥ	—
0100	Ⓥ	—	—	—
0011	—	—	v	v
0110	v	v	—	—
1100	Ⓥ	—	—	—
0111	—	v	—	v
1110	v	v	—	—
1111	—	Ⓥ	—	—

Quine'a pokazanej w tablicy 3.6. Wiersze tablicy odpowiadają jedynkom danej funkcji, a kolumny odpowiadają powstałym w wyniku sklejania implikantom. Dla przykładu 3.9 tablica Quine'a ma 8 wierszy i 4 kolumny, gdyż funkcja ma 8 jedynek a 4 implikanty pokrywają wszystkie jedynki.

Tablicę Quine'a wypełnia się zaznaczając (w tablicy 3.6 — znakiem v) pokrycia jedynek funkcji przez implikanty powstałe w pierwszym etapie minimalizacji. Na przykład implikant $x1x0$ pokrywa 4 jedynki: 0100, 0110, 1100 i 1110. Następnie wyszukuje się implikanty zasadnicze, tj. takie które muszą wejść do minimalnej postaci sumacyjnej. Wyszukiwanie ich polega na tym, że jeśli w danym wierszu występuje tylko jeden znaczek v, to oznacza to, że dana jedynka jest pokrywana tylko przez jeden implikant. Implikant ten jest w tej samej kolumnie, w której znajduje się znaczek v. W tablicy kółkami zaznaczono te znaczki v, które wskazują implikanty zasadnicze. Następnie wyznacza się możliwe pokrycia pozostałych jedynek. W przedstawianym przykładzie są trzy implikanty zasadnicze: $x1x0$, $x11x$ i $00x1$.

Pokrywają one wszystkie jedynki funkcji, a więc istnieje tylko jedno rozwiązanie minimalne składające się z dwóch czwórek ($x1x0$ i $x11x$) i jednej pary ($00x1$), tj. $y = x_2\bar{x}_0 + x_2x_1 + \bar{x}_3\bar{x}_2x_0$.

Przykład 3.10. Niech będzie dana funkcja 3 zmiennych $y = \Sigma(0, 2, 3, 4, 5, 7)$.

Rozwiązanie. Pogrupowanie implikantów i poszukiwanie par pokazano na rysunku 3.22. Można zauważyć, że brak jest czwórek.

000 (0)	0x0 (0,2)
010 (2)	x00 (0,4)
100 (4)	01x (2,3)
101 (5)	10x (4,5)
011 (3)	1x1 (5,7)
111 (7)	x11 (3,7)

Rysunek 3.22. Poszukiwanie grup jedynek funkcji z przykładu 3.10

W pierwszym etapie minimalizacji otrzymuje się 6 par, przy czym każdy implikant prosty wchodzi w skład jakiejś pary. Sporządzana w drugim etapie minimalizacji tablica Quine'a jest pokazana w tablicy 3.7.

W tym przykładzie implikanty zasadnicze nie występują. Dlatego należy znaleźć minimalne pokrycia wszystkich jedynek funkcji. W tym celu tworzy się wyrażenie

Tablica 3.7. Tablica Quine'a dla przykładu 3.10

	0x0	x00	01x	10x	1x1	x11
000	v	v	—	—	—	—
010	v	—	v	—	—	—
100	—	v	—	v	—	—
101	—	—	—	v	v	—
011	—	—	v	—	—	v
111	—	—	—	—	v	v

boolowskie opisujące wszystkie możliwe pokrycia. Oznaczmy symbolami wszystkie implikanty danej funkcji, a więc pary (0, 2), (0, 4), (2, 3), (4, 5), (5, 7) i (3, 7).

Implikantowi	(0, 2) — 0*x*0	przypiszemy	symbol	*a*
„	(0, 4) — *x*00	„	„	*b*
„	(2, 3) — 01*x*	„	„	*c*
„	(4, 5) — 10*x*	„	„	*d*
„	(5, 7) — 1*x*1	„	„	*e*
„	(3, 7) — *x*11	„	„	*f*

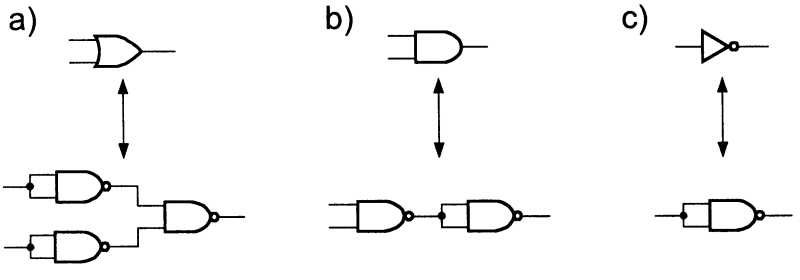
Aby zrealizować pierwszą jedynkę (000) trzeba wziąć parę *a* lub *b*. Aby zrealizować drugą jedynkę (010) trzeba wziąć parę *a* lub *c*. Aby zrealizować trzecią jedynkę (100) trzeba wziąć parę *b* lub *d*. Aby zrealizować czwartą jedynkę (101) trzeba wziąć parę *d* lub *e*. Aby zrealizować piątą jedynkę (011) trzeba wziąć parę *c* lub *f*. Aby zrealizować szóstą jedynkę (111) trzeba wziąć parę *e* lub *f*. Wyrażenie boolowskie

$$(a+b)(a+c)(b+d)(d+e)(c+f)(e+f)$$

określa wszystkie możliwe rozwiązania. Po wymnożeniu otrzymać można postać sumacyjną tego wyrażenia. Każdy składnik tej postaci wyznacza jedno możliwe rozwiązanie, czyli pokrycie wszystkich jedynek funkcji. Zadaniem projektanta jest znalezienie minimalnego pokrycia, czyli składników o jak najmniejszej liczbie czynników. W podanym przykładzie tylko dwa składniki będą trzyliterowe: *adf* i *bce*. One określają dwa minimalne rozwiązania: $y = \bar{x}_2\bar{x}_0 + x_2\bar{x}_1 + x_1x_0$ lub $y = \bar{x}_1\bar{x}_0 + \bar{x}_2x_1 + x_2x_0$. ⊠

3.2.4. Realizacja funkcji z wykorzystaniem bramek NAND i NOR

Rozważania prowadzone w poprzednim punkcie dotyczyły rozwiązań wykorzystujących bramki sumy, iloczynu i negacji. Mówimy, że te trzy operacje (a tym samym trzy rodzaje

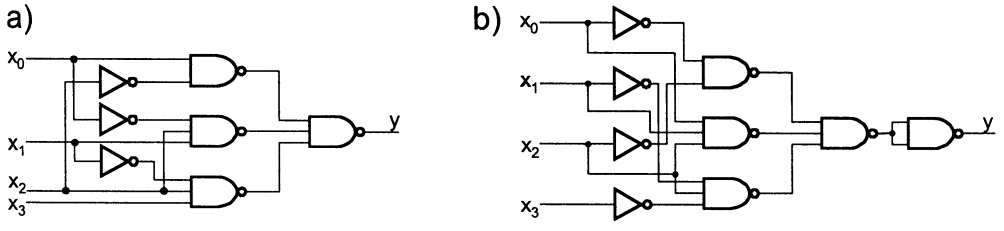


Rysunek 3.23. Realizacje bramek sumy, iloczynu i negacji za pomocą bramek NAND

bramek) stanowią zestaw funkcjonalnie pełny, tj. taki, że dowolną funkcję można zapisać za pomocą tych trzech operacji, a funkcję zrealizować za pomocą trzech odpowiednich rodzajów bramek. Istnieją inne zestawy funkcjonalnie pełne. Tu zostaną przedstawione dwa z nich: zestaw z bramkami NAND i zestaw z bramkami NOR.

Na rysunku 3.23 pokazano jak można układ z bramkami sumy, iloczynu i negatorami przekształcać na układ wykorzystujący bramki NAND.

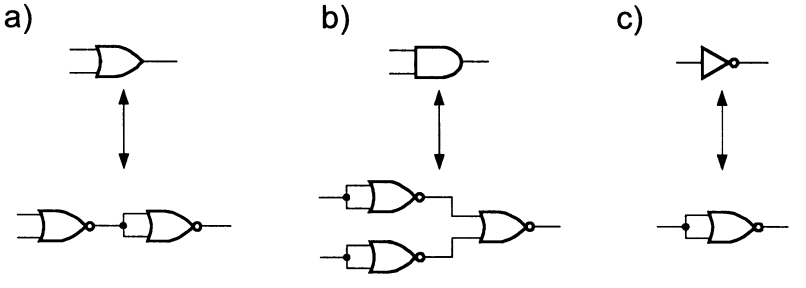
Prosty sposób znalezienia realizacji funkcji boolowskiej za pomocą bramek NAND polega na tym, że w układzie zbudowanym na bramkach iloczynu, sumy i negacji realizującym minimalną postać sumacyjną lub minimalną postać iloczynową zamienia się te bramki na bramki NAND według sposobu pokazanego na rysunku 3.23. Następnie należy wyeliminować podwójne zanegowania, aby otrzymać układ trzypoziomowy. Na rysunku 3.24a pokazano zbudowany na bramkach NAND układ odpowiadający postaci sumacyjnej (rysunek 3.20), ale dla przejrzystości pozostawiono negatory wejściowe.



Rysunek 3.24. Realizacja na bramkach NAND rozwiązania przykładu 3.1: a) postaci sumacyjnej, b) postaci iloczynowej.

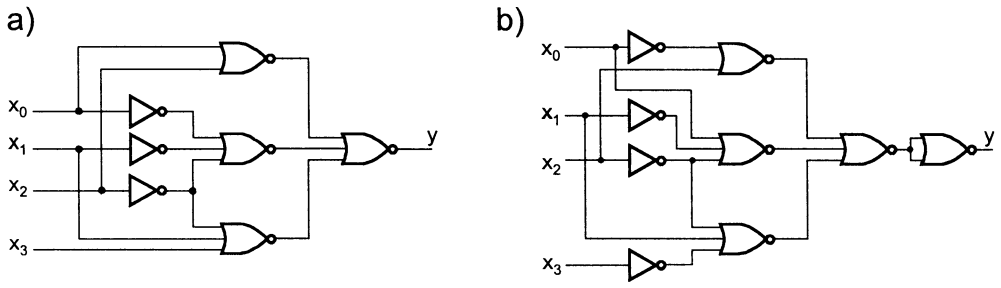
Na rysunku 3.24b pokazano układ na bramkach NAND realizujący postać iloczynową tej samej funkcji. Uzyskano rozwiązanie czteropoziomowe.

Analogicznie można pokazać jak można przekształcić układ zbudowany na bramkach sumy, iloczynu i negacji na układ logiczny z bramkami NOR. Na rysunku 3.25 pokazano odpowiedniość bramek sumy, iloczynu i negacji z bramkami NOR.



Rysunek 3.25. Realizacje bramek sumy, iloczynu i negacji za pomocą bramek NOR

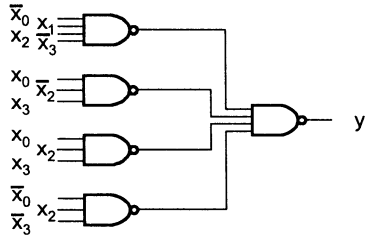
Pokazany na rysunku 3.26a układ na bramkach NOR realizuje postać iloczynową drugiego rozwiązania z przykładu 3.1. Układ ten jest trzypoziomowy. Realizując postać sumacyjną otrzyma się rozwiązanie czteropoziomowe pokazane na rysunku 3.26b.



Rysunek 3.26. Realizacja na bramkach NOR rozwiązania przykładu 3.1: a) postaci iloczynowej, b) postaci sumacyjnej

Zadania dla Czytelnika

1. Dany jest układ zbudowany na bramkach NAND jak na rysunku 3.27.
   a) Wypełnić mapę Karnaugha odpowiadającą temu układowi.
   b) Znaleźć minimalną postać sumacyjną funkcji realizowanej przez dany układ.
   c) Zrealizować na bramkach NAND układ składający się z najmniejszej liczby układów scalonych.
   d) Z ilu i jakich bramek składa się rozwiązanie tego zadania jeśli wzbudzenie $y = x_3x_2x_1\bar{x}_0$ (1110) nie występuje (funkcja jest nieokreślona)?



Rysunek 3.27. Przykładowy układ zbudowany z bramek NAND

2. Dane są dwie funkcje: $y_1 = \Sigma(1, 2, 3, 6)$ i $y_2 = \Pi(0, 2)$. Zaprojektować układ realizujący obydwie funkcje. Czy istnieje rozwiązanie wykorzystujące tylko jeden układ scalony zawierający 4 dwuwejściowe bramki NAND?
3. Zaprojektować układ sprawdzający, czy liczba jedynek w trzybitowym słowie wejściowym jest większa lub równa 2. Wykorzystać tylko bramki NAND.

3.2.5. Projektowanie układów wielowyjściowych

Jeśli przed projektantem stoi zadanie zrealizowania zespołu kilku różnych funkcji tych samych zmiennych, to poprawne rozwiązanie uzyskuje się nie tyle przez minimalizację poszczególnych funkcji, lecz raczej przez takie przedstawienie funkcji, aby miały one jak największe części wspólne. Niezbyt skomplikowane zadania wprawny projektant rozwiązuje stosując mapy Karnaugha, ale trudno wtedy o optymalne rozwiązanie. Natomiast przypadki, gdy zmiennych wejściowych jest więcej niż 5 muszą być rozwiązywane algorytmicznie.

Zwykle tego typu problemy rozwiązuje się za pomocą systemów komputerowych CAD (ang. *computer aided design*). Algorytm poszukiwania minimalnego rozwiązania dla zespołu funkcji opiera się na algorytmie Quine'a-McCluskeya. Algorytm ten zmodyfikowano w taki sposób, że elementy tworzonych podzbiorów są oznaczane ich przynależnością do danej funkcji. Następnie tablica Quine'a jest tworzona w taki sposób, że jej wiersze stanowią jedynki wszystkich funkcji (pogrupowane dla każdej funkcji oddzielnie), natomiast kolumny odpowiadają znalezionym implikantom. Ten sposób ułatwia wyszukanie minimalnej liczby implikantów pokrywających wszystkie jedynki wszystkich funkcji. Ze względu na to, że najczęściej problemy takie są rozwiązywane za pomocą systemów CAD tutaj pokazany zostanie jedynie prosty przykład.

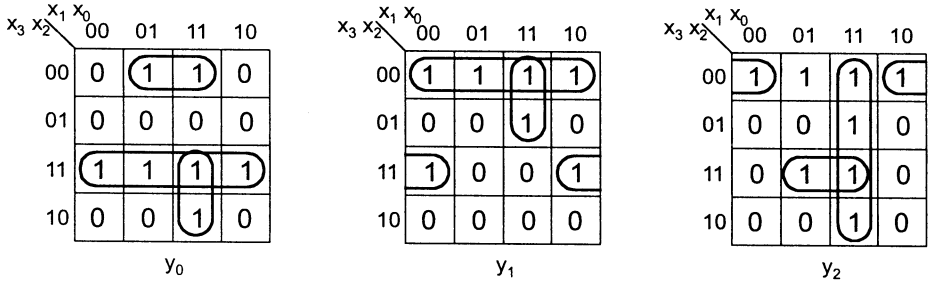
Przykład 3.11. Dane są trzy funkcje 4 zmiennych:

$$y_0 = \Sigma(1,3,11,12,13,14,15)$$
$$y_1 = \Sigma(0,2,3,7,11,13,15)$$
$$y_2 = \Sigma(0,2,3,7,11,13,15)$$

Znaleźć ich minimalne rozwiązanie.

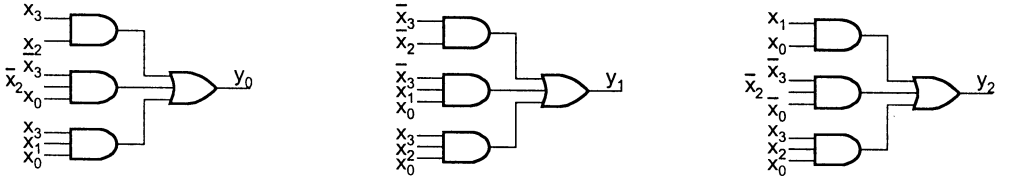
Rozwiązanie. Aby prześledzić rozwiązanie posłużmy się mapami Karnaugha danych funkcji dla wyznaczenia ich minimalnych postaci sumacyjnych. Mapy te są pokazane na rysunku 3.28. W wyniku minimalizacji przeprowadzonej dla każdej funkcji oddzielnie otrzymamy rozwiązania składające się z jednej czwórki i dwóch par, co zapisać można jako:

$$y_0 = \Sigma(12,13,14,15)(1,3)(11,15)$$
$$y_1 = \Sigma(0,1,2,3)(3,7)(12,14)$$
$$y_2 = \Sigma(3,7,11,15)(0,2)(13,15)$$

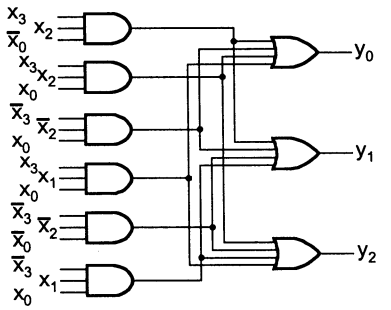


Rysunek 3.28. Mapy Karnaugha zespołu funkcji z przykładu 3.11.

Realizując te postacie otrzymamy układ pokazany na rysunku 3.29 składający się z 12 bramek (3 dwuwejściowe bramki iloczynów, 6 bramek trzywejściowych iloczynów i trzech trzywejściowych bramek sumy).



Rysunek 3.29. Realizacja minimalnych postaci sumacyjnych funkcji y_0, y_1 i y_2 z przykładu 3.11



Rysunek 3.30. Układ realizujący zespół funkcji z przykładu 3.11

Można zauważyć (a projektując w sposób algorytmiczny znaleźć), że otrzymane czwórki można rozbić na pary w taki sposób, aby powstały pary wspólne dla dwóch (lub więcej) funkcji. I tak czwórkę (12, 13, 14, 15) rozbić można na pary (12, 14) i (13, 15), czwórkę (0, 1, 2, 3) na pary (0, 2) i (1, 3) i czwórkę (3, 7, 11, 15) na pary (3, 7) i (11, 15). Otrzymamy wówczas postacie:

$$y_0 = \Sigma(12,14)(13,15)(1,3)(11,15)$$
$$y_1 = \Sigma(0,2)(1,3)(3,7)(12,14)$$
$$y_2 = \Sigma(3,7)(11,15)(0,2)(13,15)$$

Na rysunku 3.30 pokazano układ realizujący powyższe funkcje i składający się z 9 bramek (6 bramek iloczynów trzywejściowych i trzech czterowejściowych bramek sumy). ⊠

3.3. Standardowe bloki realizujące funkcje boolowskie

Podczas projektowania złożonych układów logicznych często zachodzi konieczność wielokrotnego wykorzystania tego samego układu realizującego określoną funkcję. Pewne funkcje bardzo często występują w różnych projektach. Wykorzystali to producenci układów scalonych oferując układy standardowe realizujące te funkcje. Tutaj zostaną przedstawione 4 grupy układów standardowych zwanych dalej blokami:

1) dekodery i kodery,
2) multipleksery i demultipleksery,
3) sumatory,
4) komparatory.

W układach tych wyróżnia się wejścia informacyjne i wejścia sterujące — określające czynności, które dany blok ma aktualnie wykonywać. Dalej kolejno będą omawiane wymienione grupy bloków.

3.3.1. Dekodery i kodery

Dekoderem nazywa się układ przekształcający słowa jednego kodu w słowa innego. Jednym z najpopularniejszych dekoderów jest układ przekształcający słowa n-bitowego kodu NKB na słowa kodu „1 z 2^n". W tablicy 3.8 pokazano takie przekształcenie dla trzybitowych słów kodu NKB.

Tablica 3.8. Przyporządkowanie słów kodu „1 z 2^n" słowom kodu NKB

kod NKB	kod „1 z 2^n"	kod NKB	kod „1 z 2^n"
000	00000001	100	00010000
001	00000010	101	00100000
010	00000100	110	01000000
011	00001000	111	10000000

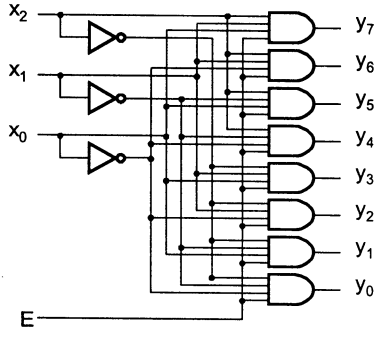
Układ takiego dekodera ma n wejść i 2^n wyjść. Każdemu słowu wejściowemu jest przypisany jeden bit słowa wyjściowego i tym samym jedno wyjście układu. Jeżeli dane słowo pojawi się na wejściu układu, to na odpowiednim wyjściu pojawi się 1, a na wszystkich innych będą zera. Liczba zakodowana kombinacją bitów na wejściu określa indeks tego wyjścia, na którym ma się pojawić jedynka. Częściej używa się dekoderów, które wyróżniają daną kombinację przyjmując na odpowiednim wyjściu wartość 0, podczas gdy na pozostałych wyjściach są 1.

Dekoder umożliwia wykrywanie odpowiednich kombinacji słów wejściowych. Przykładowo układ dekodera można stosować w zamkach szyfrowych. Jeżeli jakieś wyjście dekodera steruje otwieraniem zamka, to tylko podanie właściwej kombinacji zmiennych wejściowych spowoduje jego otworzenie. W technice komputerowej typowym zastosowaniem dekodera jest wybieranie odpowiednich słów z pamięci. Każdej kombinacji wejściowej odpowiada sygnał wyjściowy sterujący daną komórką pamięci. Kombinację bitów wybierającą daną komórkę nazywa się adresem. Dlatego często wejścia dekodera nazywa się wejściami adresowymi. Im więcej słów w pamięci tym więcej wyjść dekodera i tym więcej wejść dekodera. Jeśli pamięć ma 1 M słów ($1\ M = 1024 \times 1024 = 2^{20}$), to dekoder ma 20 wejść. Budowanie dekoderów o dużej liczbie wejść nastręcza trudności technologiczne i dlatego stosuje się pewne specjalne rozwiązania pokazane w dalszej części książki.

a)

E	x_2	x_1	x_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	x	x	x	0	0	0	0	0	0	0	0

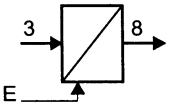
b)



Rysunek 3.31. Dekoder 3-wejściowy: a) tablica prawdy dekodera trzybitowego kodu NKB, b) schemat dekodera

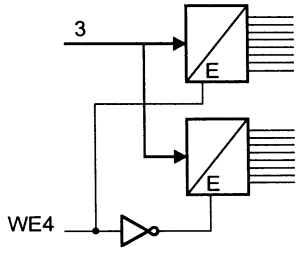
Na rysunku 3.31a przedstawiono tablicę prawdy 3-wejściowego, a więc 8-wyjściowego dekodera. Realizację takiego układu pokazano na rysunku 3.31b. Układ ma dodatkowe wejście E (ang. *enable*) nazywane wejściem zezwalającym. Jeśli $E = 0$, to działanie układu zostaje zablokowane. Mówimy, że wówczas układ nie jest aktywny. W takim przypadku na wyjściu dekodera pojawiają się same 0, a więc słowo wyjściowe nie jest z kodu „1 z 2^n". Często układy są wyposażane w zanegowane wejście E i wówczas układ jest aktywny, gdy na tym wejściu jest 0. Symbol graficzny 3-wejściowego dekodera pokazano na rysunku 3.32.

Dekoderom i innym blokom standardowym stawia się wymaganie, aby istniała możliwość łączenia takich układów ze sobą w celu zwiększenia dekodowanych kombinacji



Rysunek 3.32. Oznaczenie dekodera 3-wejściowego

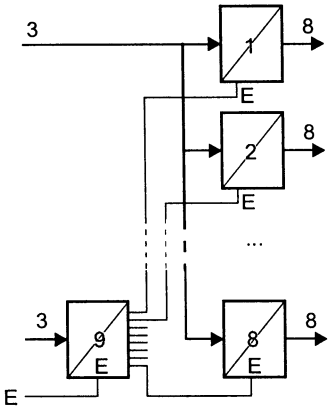
sygnałów wejściowych. Na rysunku 3.33 pokazano jak z dwóch dekoderów o trzech wejściach można zbudować jeden dekoder 4-wejściowy (16-wyjściowy). Należy zwrócić uwagę na wykorzystanie do tego celu wejścia *E*. Jeżeli na wejściu *E* będzie 1, to aktywny będzie dekoder górny, a dolny nie będzie aktywny. Brak aktywności oznacza, że na wszystkich wyjściach pojawią się zera (jeśli kod wyjściowy jest kodem „1 z 2^n") lub jedynki (jeśli kod



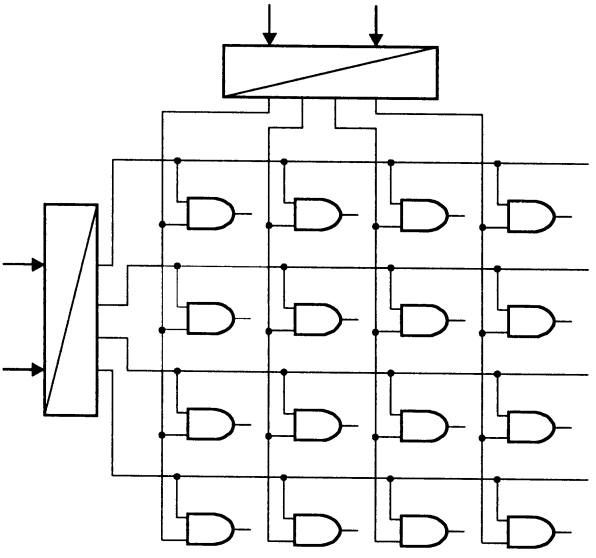
Rysunek 3.33. Dekoder 4-wejściowy zbudowany z dwóch dekoderów 3-wejściowych

wyjściowy jest kodem „0 z 2^n"). Jeżeli natomiast na to wejście zostanie podane 0, to aktywny będzie dekoder dolny, a górny nie będzie aktywny. Zatem oba dekodery pracują naprzemiennie w zależności od stanu na wejściu *WE4*. Wejście *WE4* można w tym przypadku traktować jako dodatkowe wejście adresowe. Oba układy wraz z bramką inwertera tworzą dekoder o słowach wyjściowych w kodzie „1 z 2^4".

Strukturę taką można rozbudowywać. Na rysunku 3.34 pokazano dekoder o 6 wejściach i 64 wyjściach zbudowany z 9 dekoderów 3-wejściowych. Jest to struktura dwustopniowa zwana kaskadową. Na pierwszym stopniu znajduje się dekoder sterujący wejściami *E* de-

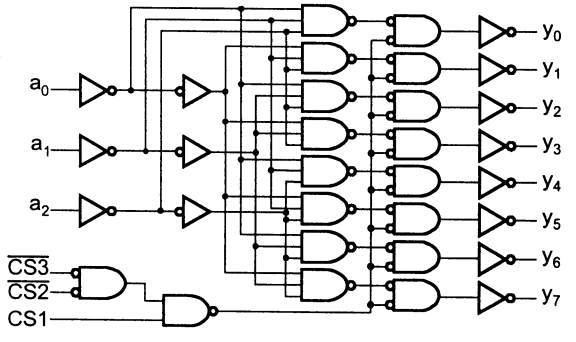


Rysunek 3.34. Dekoder 6-wejściowy zbudowany z dekoderów 3-wejściowych

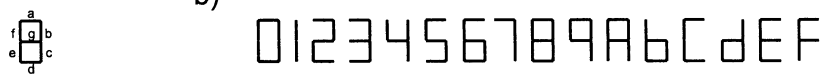


Rysunek 3.35. Dekoder współrzędnościowy o 16 wyjściach

koderów drugiego stopnia. Na drugim stopniu znajduje się 8 dekoderów, przy czym tylko jeden z nich może być aktywny w danej chwili ze względu na sposób sterowania sygnałami dekodera pierwszego stopnia. Istnieje możliwość dalszej rozbudowy takich układów. Postępując analogicznie można złożyć dekoder 12-wejściowy (4096-wyjściowy). Do tego celu trzeba użyć 65 dekoderów o 6 wejściach adresowych. Układ taki ma tę wadę, że gdy wzrasta liczba jego stopni, to razem z nią wzrasta czas propagacji sygnału od wejścia do wyjścia. Istnieje także inne rozwiązanie problemu zbudowania dekodera o dużej liczbie wejść, które pokazano na rysunku 3.35. Jest to tzw. dekoder współrzędnościowy. Składa się on z matrycy bramek dwuwejściowych sterowanych z dwóch dekoderów. Jeśli liczbę wejść dekodera współrzędnościowego oznaczymy przez n i liczba ta jest parzysta, to matryca bramek jest kwadratowa a liczba wejść dekoderów sterujących matrycą wynosi $n/2$. Jeśli n jest liczbą nieparzystą, to matryca jest prostokątna i liczby wejść dekoderów sterujących różnią się o 1.



Rysunek 3.36. Typowy dekoder SN 74138



Rysunek 3.37. Wyświetlanie znaków za pomocą wyświetlacza siedmiosegmentowego

Do tej pory założono, że omawiane dekodery składają się z określonej liczby bramek iloczynów, jeśli na wyjściu ma być kod „1 z *n*". Zgodnie z tym co powiedziano wcześniej, bramki te można zamienić na bramki NOR podając na ich wejścia zanegowane zmienne. Jeśli na wyjściu dekodera ma być kod „0 z *n*", to dekoder taki można zbudować z bramek sumy logicznej lub z bramek NAND. Typowy dekoder produkowany jako układ scalony pokazano na rysunku 3.36. Jest to układ, oznaczany symbolem SN 7138, 3-wejściowy i 8-wyjściowy z trzema wejściami dostępu. Dekoder jest aktywny, gdy $CS1 = 1$, $\overline{CS2} = 0$ i $\overline{CS3} = 0$.

Wśród dekoderów dostępnych na rynku jako układ scalony spotyka się dekoder przekształcający czterobitowy kod NKB na kod sterujący tzw. wyświetlaczem siedmiosegmentowym. Wyświetlacz siedmiosegmentowy jest to zestaw siedmiu elementów świecących ułożonych jak pokazano na rysunku 3.37a. Jak widać z rysunku za pomocą takiego wyświetlacza można wyświetlić wszystkie cyfry od 0 do 9, a także inne znaki (litery od A do F). Na rysunku 3.37b pokazano jak można wyświetlić uproszczone znaki tzw. **kodu szesnastkowego** (ang. *hexadecimal*). Uproszczenie polega na tym, że litery B i D wyświetlane są jako małe.

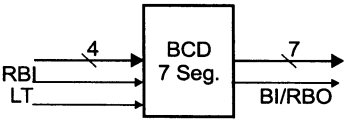
Dekoder sterujący wyświetlaczem siedmiosegmentowym ma 4 wejścia i 7 wyjść. W tablicy 3.9 przedstawiono jego tablicę prawdy. Sposób realizacji tych funkcji może być różny: na bramkach sumy, na bramkach iloczynu, na bramkach NAND lub na bramkach NOR. Ponieważ często występuje także potrzeba wyświetlania większej liczby cyfr, to dekodery takie wyposaża się w dodatkowe wejścia i wyjścia. Typowym układem średniej skali integracji TTL jest dekoder typu SN 7447 pokazany na rysunku 3.38.

Dekoder SN 7447 oprócz omówionych wcześniej wejść i wyjść ma jeszcze trzy dodatkowe końcówki:

— dwa wejścia sterujące:
  LT — wejście testowe (zapalają się wszystkie segmenty), umożliwiające sprawdzenie, czy któryś z segmentów nie uległ uszkodzeniu,
  RBI — wejście wygaszania (wyłączenie wszystkich segmentów),
— końcówkę dwukierunkową (stanowiącą wejście lub wyjście):
  BI/RBO — jeśli końcówka jest wejściem, to służy do wygaszenia danego segmentu, jeśli końcówka jest wyjściem, to służy do wygaszenia segmentów bardziej znaczących cyfr.

Tablica 3.9. Tablica prawdy dekodera siedmiosegmentowego

	wej.	segmenty abcdefg		wej.	segmenty abcdefg	
0	0000	1111110	8	1000	1111111	
1	0001	0110000	9	1001	1110011	
2	0010	1101101	A	1010	1110111	
3	0011	1111001	B	1011	0011111	
4	0100	0110011	C	1100	1001110	
5	0101	1011011	D	1101	0111101	
6	0110	1011111	E	1110	1001111	
7	0111	1110000	F	1111	1000111	



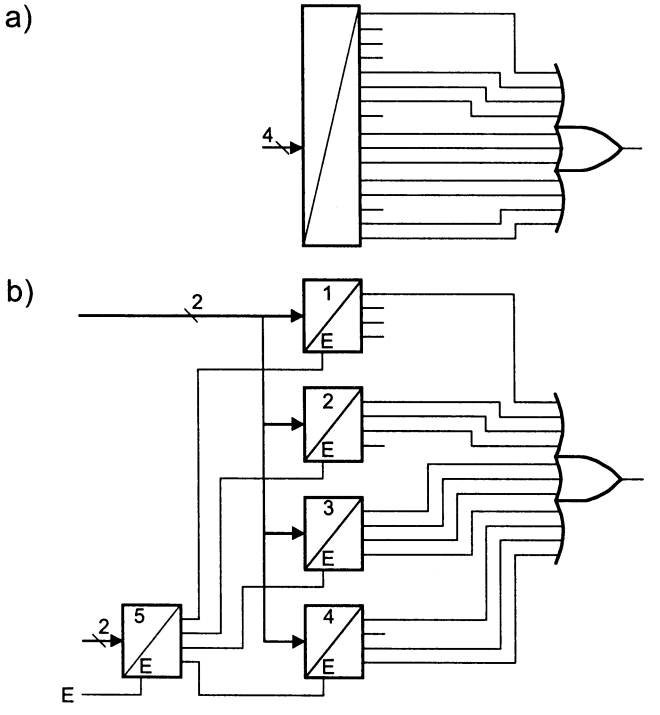
Rysunek 3.38. Wejścia i wyjścia dekodera kodu siedmiosegmentowego

Dekoder 7447 ma wyjścia zanegowane, tzn. na wyjściu jest kod „0 z *n*". Dlatego segmenty przyłączonego wskaźnika siedmiosegmentowego mają zapalać się wówczas, gdy na wyjściu dekodera jest zero logiczne.

Dekodery można wykorzystywać do realizacji funkcji boolowskich. Ponieważ każde wyjście dekodera odpowiada implikantom prostym funkcji tych zmiennych, które podawane są na wejścia dekodera, to poprzez odpowiednie sumowanie wyjść dekodera można uzyskać postać sumacyjną dowolnej funkcji. Sposób realizacji funkcji pokazano na rysunku 3.39.

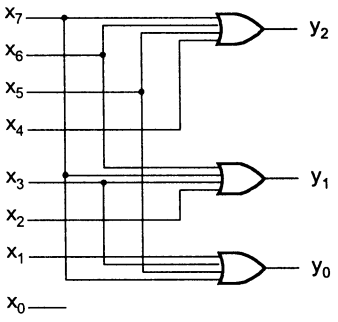
Na rysunku 3.39a pokazano jak za pomocą dekodera o 4 wejściach zrealizować funkcję odpowiadającą jednemu z wyjść (dla segmentu F) dekodera siedmiosegmentowego. Na rysunku 3.39b pokazano jak dekoder o 4 wejściach zastąpić dekoderami o 2 wejściach. Układ realizuje tę samą funkcję *f* dekodera kodu siedmiosegmentowego. Dekoder o 4 wejściach zastąpiono 5 dekoderami o dwóch wejściach.

Koderem (lub enkoderem) nazywa się układ kombinacyjny przekształcający kody tak, że słowa kodu wejściowego są dłuższe niż słowa kodu wyjściowego. Na przykład koderem



Rysunek 3.39. Realizacja funkcji *f* dekodera kodu siedmiosegmentowego za pomocą: a) dekodera czterowejściowego, b) dekoderów dwuwejściowych

a)



b)

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Rysunek 3.40. Kodowanie słów kodu „1 z 8": a) układ kodera, b) tablica prawdy

nazywa się układ zamieniający kod „1 z *n*" na kod NKB. Na rysunku 3.40 pokazano koder słowa kodu „1 z 8" na kod binarny NKB wraz z tablicą prawdy przypisującą słowom kodu „1 z 8" odpowiednie słowa kodu NKB. Można zauważyć, że słowo wyjściowe kodera wskazuje binarnie numer wejścia (indeks zmiennej wejściowej), na którym jest jedynka.

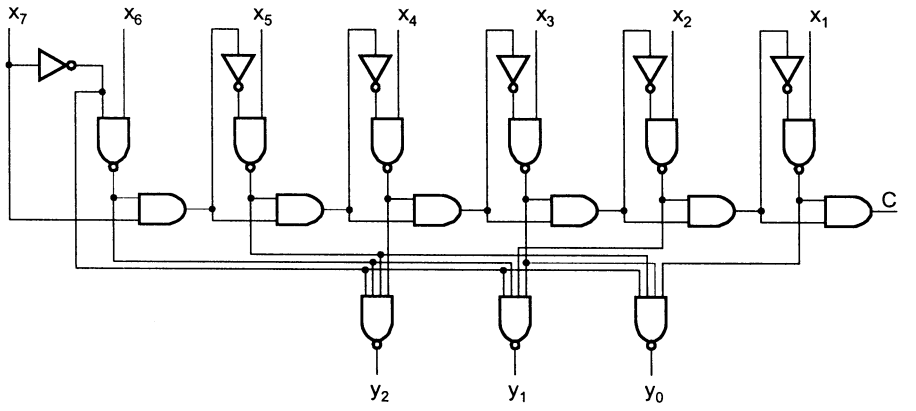
Zauważmy, że działanie układu nie jest określone w przypadku, gdy na wejściu pojawi się więcej jedynek. Przykładowo w komputerach spotyka się często układ zwany **koderem priorytetowym**. Jest to układ kombinacyjny, którego wejściom przypisano priorytety, tj. kolejność ich ważności. Oznacza to, że na wyjściu układu powinien pojawić się numer tego spośród wejść, na którym jest jedynka i które ma najwyższy priorytet. Rozpatrzmy układ o 8 wejściach $x_7 - x_0$ i załóżmy, że najwyższy priorytet przypiszemy wejściu x_7, a najniższy wejściu x_0. Wówczas jeśli na wejściu pojawi się słowo $1xxx\ xxxx$ ($x_7 = 1$, x = dowolność), to na wyjściu układu powinno być wskazanie na wejście o numerze 7. Ponieważ w takim układzie słowo wyjściowe jest trzybitowe, to na wyjściu pojawią się trzy jedynki. Jeśli na wejściu pojawi się słowo $01xx\ xxxx$ ($x_6 = 1$), to na wyjściu układu powinno być wskazanie na wejście o numerze 6 itd. Celem zaprojektowania takiego układu należy wyznaczyć trzy funkcje boolowskie, których tablice prawdy przedstawiono w tablicy 3.10.

Wyznaczając funkcje y_2, y_1 i y_0 z tablicy prawdy otrzymamy, że:

$$y_2 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 x_4$$
$$y_1 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 x_2$$
$$y_0 = x_7 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 x_1$$

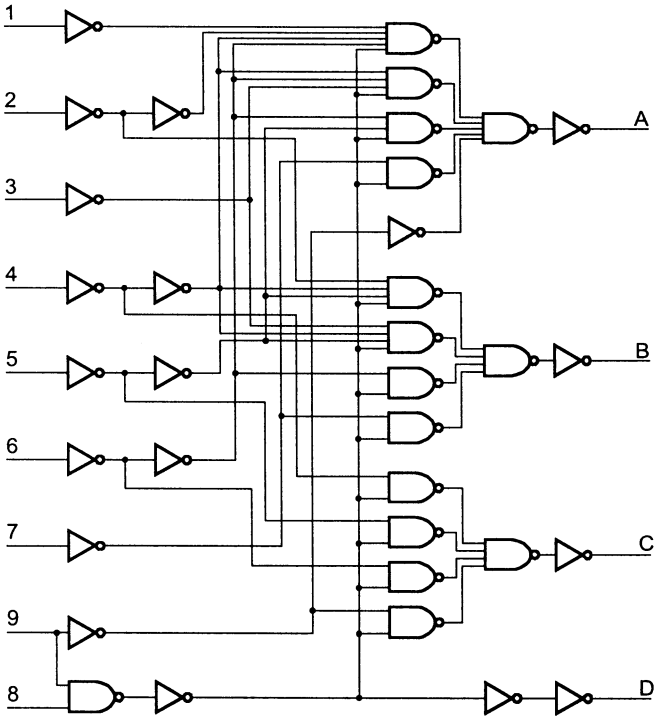
Tablica 3.10. Uproszczona tablica prawdy dekodera priorytetowego

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_2	y_1	y_0
1	x	x	x	x	x	x	x	1	1	1
0	1	x	x	x	x	x	x	1	1	0
0	0	1	x	x	x	x	x	1	0	1
0	0	0	1	x	x	x	x	1	0	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	0	0	1	0	0	0



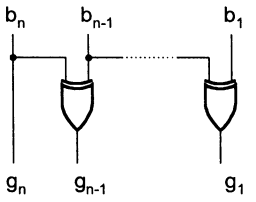
Rysunek 3.41. Iteracyjny koder priorytetowy

Jednym z możliwych rozwiązań realizacji takich funkcji jest zbudowanie układu iteracyjnego wyznaczającego poszczególne składniki pokazanych sum. Układ taki jest pokazany na rysunku 3.41. W serii układów TTL produkowany jest koder priorytetowy SN 74147 pokazany na rysunku 3.42.



Rysunek 3.42. Dekoder priorytetowy SN 74147

Omówione układy dekoderów i koderów nazywane są także **konwerterami kodów**. Innym przykładem konwertera kodu może być konwerter kodu NKB na kod Graya (i odwrotnie konwerter kodu Graya na kod NKB). Poszczególne bity kodu Graya g_i można wyznaczyć



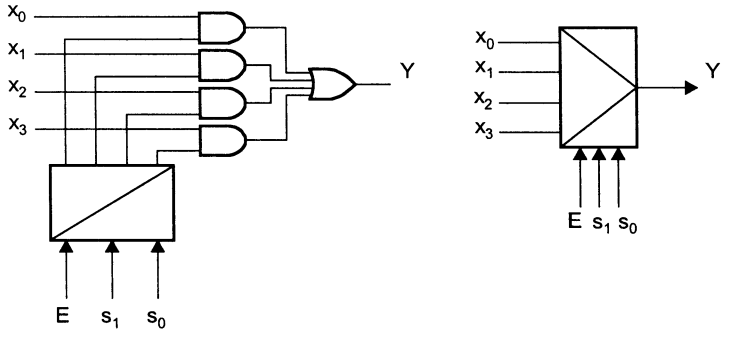
Rysunek 3.43. Konwerter kodu NKB na kod Graya

na podstawie bitów kodu NKB b_i biorąc: $g_i = b_i \oplus b_{i+1}$. Na rysunku 3.43 pokazano układ działający według tej zależności. Podobnie można opisać konwerter kodu Graya na kod NKB. Czytelnikowi zaleca się zaprojektowanie takiego układu.

W serii układów TTL produkowane są także inne konwertery kodów, jak na przykład: 74184 — konwerter kodu BCD na kod binarny, 74185 — konwerter kodu binarnego na kod BCD.

3.3.2. Multipleksery i demultipleksery

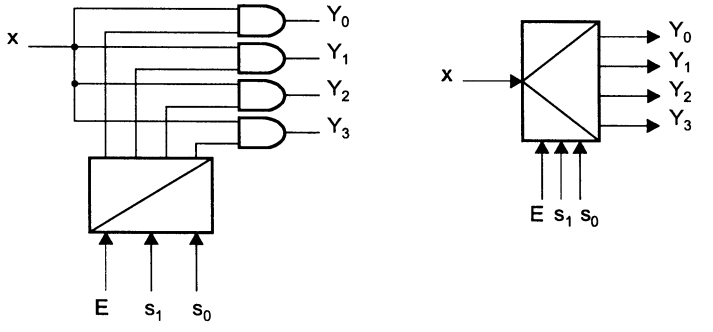
Multipleksery i demultipleksery są układami realizującymi funkcję wybierania (selekcji) sygnałów. Układy te są wyposażone w dwie grupy wejść: wejścia informacyjne i wejścia sterujące. W układzie multipleksera podając odpowiednie słowo na wejścia sterujące wybiera



Rysunek 3.44. Układ i symbol graficzny multipleksera 4×1

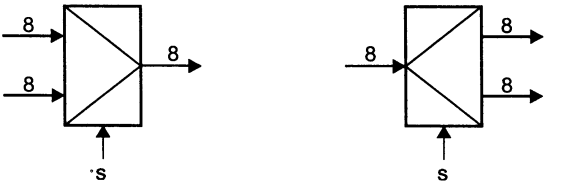
się jedno z wejść informacyjnych, co oznacza, że informacja z tego wejścia jest podawana na wyjście. Przekazywanie informacji z wejścia na wyjście jest rozumiane w ten sposób, że podanie zera na wejście powoduje pojawienie się zera na wyjściu, a podanie jedynki na wejście powoduje pojawienie się jedynki na wyjściu. W układzie demultipleksera słowo na wejściach sterujących wybiera jedno z wielu wyjść i na to wyjście kierowana jest informacja z wejścia. Na rysunku 3.44 pokazano układ 4-wejściowego multipleksera (4×1) oraz jego

oznaczenie, a na rysunku 3.45 pokazano strukturę 4-wyjściowego demultipleksera (1×4) i jego oznaczenie. Wejście E jest wejściem zezwalającym. W obu układach są także dwa wejścia sterujące $s_1 s_0$. Słowo podane na wejścia sterujące określa numer wybranego wejścia multipleksera lub wyjścia demultipleksera. W układach tych słowo sterujące podawane jest na wejścia dekodera (w pokazanych na rysunku 3.44 i 3.45 przykładowych układach jest to dekoder 2-wejściowy), który steruje bramkami iloczynów.



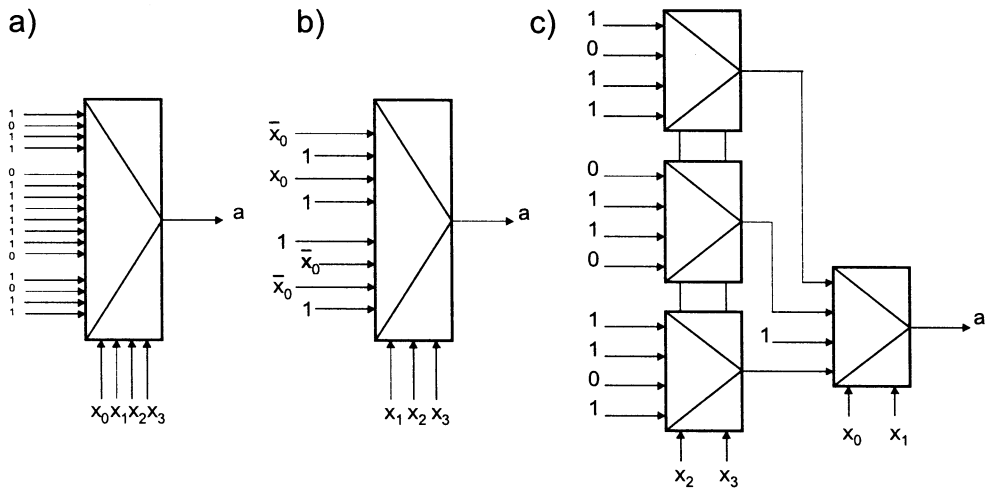
Rysunek 3.45. Układ i symbol graficzny demultipleksera 1×4

Przedstawione na rysunkach 3.44 i 3.45 układy są układami jednobitowymi, tj. wejścia informacyjne w multiplekserach, jak i wyjścia informacyjne w demultiplekserach są jednobitowe. W technice komputerowej stosuje się często układy wielobitowe, w których wybierane są grupy linii. Na rysunku 3.46 pokazano schematycznie układy zwane multiplekserami (demultiplekserami) grupowymi, w których wybierana jest jedna z dwóch grup składająca się z 8 sygnałów. Oczywiście dla takiej selekcji wystarczy tylko jedno wejście sterujące.



Rysunek 3.46. Multiplekser i demultiplekser grup linii

Multipleksery można także stosować do realizacji funkcji logicznych. Na rysunku 3.47a pokazano jak można zrealizować funkcję wyjściową *a* dekodera kodu BCD na kod siedmiosegmentowy za pomocą multipleksera o czterech wejściach sterujących (czyli o szesnastu wejściach informacyjnych). Na rysunku 3.47b pokazano jak tę samą funkcję można zrealizować za pomocą multipleksera o trzech wejściach sterujących, a na rysunku 3.44c — za pomocą multiplekserów o dwóch wejściach sterujących. W tym ostatnim przypadku szczególną rolę odgrywa wybór zmiennych wejściowych, które podaje się na wejścia sterujące multipleksera drugiego poziomu. Poleca się Czytelnikowi znalezienie rozwiązania



Rysunek 3.47. Realizacja funkcji boolowskich za pomoca multiplekserów: a) z 4 wejściami sterującymi, b) z 3 wejściami sterującymi, c) z 2 wejściami sterującymi

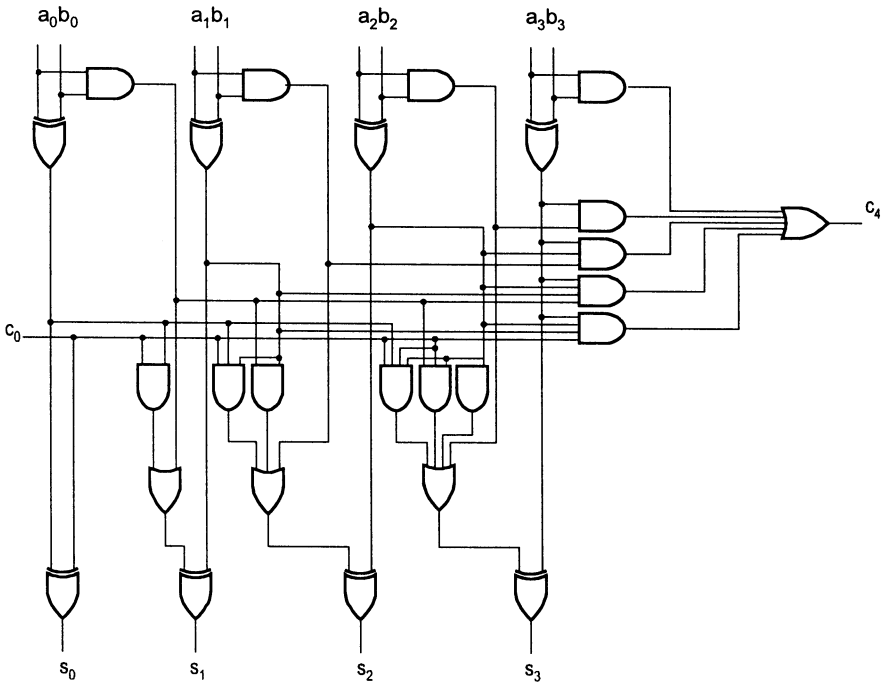
w przypadku, gdy zamieniono sygnały sterujące x_1 i x_2, tj. multipleksery pierwszego stopnia będą sterowane parą sygnałów x_1 i x_3, a multipleksery drugiego stopnia będą sterowane parą sygnałów x_0 i x_2.

3.3.3. Sumatory

Sumatorem nazywa się układ, który dodaje arytmetycznie dwie liczby dwójkowe. Rozpatrzymy tutaj układ dodawania w kodzie NKB.

Układ realizujący operację jednobitowego sumowania, analizowany w rozdziale 3.2.2, był dwuwyjściowym układem kombinacyjnym (wyjście sumy i wyjście przeniesienia na bardziej znaczącą pozycję) trzech zmiennych (dwa bity składników sumy i bit przeniesienia z poprzedniej pozycji). Aby zrealizować operację sumowania dwóch n-bitowych liczb można połączyć ze sobą n jednopozycyjnych sumatorów tak jak na rysunku 3.17, tj. w układzie iteracyjnym. Należy zwrócić uwagę na kierunek propagacji przeniesienia. W pokazanym układzie sumatora przeniesienie propaguje od najmniej znaczącej pozycji do najbardziej znaczącej. Zaletą takiego układu jest to, że może być w razie potrzeby łatwo rozbudowywany, a jego przejrzysta struktura ułatwia testowanie. Wadą układów iteracyjnych jest to, że przeniesienie może propagować przez wszystkie bloki, co powoduje zmniejszenie szybkości działania układu. Ponieważ szybkość działania jest bardzo istotnym czynnikiem, to poszukuje się takich struktur układów, które pozwalają na zwiększanie szybkości działania. W przypadku sumatorów jednym z rozwiązań może być zastosowanie układów przyspieszających sumowanie.

W celu zaprojektowania układu przyspieszającego wykorzystuje się sposób obliczenia tzw. **przeniesienia grupowego**, tj. przeniesienia wyjściowego z najbardziej znaczącej pozycji pewnej liczby (grupy) sumatorów jednobitowych. Przeniesienie to wyznacza się na podstawie znajomości wartości wszystkich bitów obu słów wejściowych oraz bitu przeniesienia wejściowego. Przykładowo dla grupy czterech najmniej znaczących bitów sumatora wyznacza się funkcję boolowską 9 zmiennych. Niech 4-bitowy sumator ma wejścia a_3, b_3,

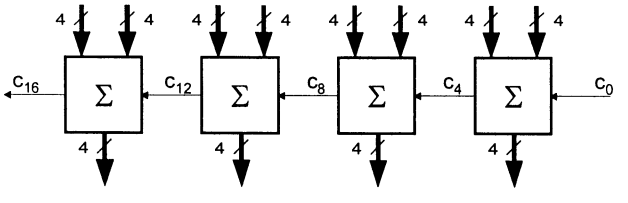


Rysunek 3.48. Czterobitowy sumator z układem generacji przeniesienia grupowego

a_2, b_2, a_1, b_1 i a_0, b_0 oraz przeniesienie wejściowe c_0. Funkcje przeniesień z poszczególnych pozycji sumatora można przedstawić w postaci $c_{i+1} = a_i b_i + (a_i + b_i)c_i$. Oznaczając przez g_i iloczyn zmiennych a_i i b_i, a przez p_i ich sumę można otrzymać:

$$c_1 = g_0 + p_0 c_0$$
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

Na podstawie ostatniego równania można zrealizować układ generujący przeniesienie c_4. Na rysunku 3.48 pokazano układ 4-bitowego sumatora wraz z układem obliczającym przeniesienie c_4. W układzie tym obliczenie funkcji y_i oraz przeniesień c_i następuje według powyższych wzorów. Sumowanie realizowane za pomocą tego układu jest szybsze w stosunku do zwykłego sumatora, gdyż przeniesienie propaguje przez mniejszą liczbę bramek. Łącząc ze sobą cztery takie 4-bitowe sumatory można zbudować sumator 16-bitowy

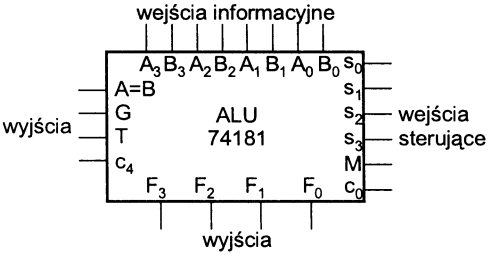


Rysunek 3.49. Sumator 16-bitowy z przeniesieniem grupowym

(rys. 3.49), w którym przeniesienie propagowane jest przez cztery stopnie iteracji, a nie przez szesnaście, jak byłoby w przypadku klasycznego układu iteracyjnego (por. rys. 3.17).

Sumatory wykorzystywane są w tzw. układach **arytmetyczno-logicznych ALU** (ang. *arithmetic-logic unit*). W serii TTL taki układ jest oznaczany numerem SN 74181. ALU jest układem kombinacyjnym, który realizuje różne funkcje w zależności od jego wysterowania pięciobitowym słowem na wejściach $s_3 - s_0$ i M. Argumentami tych operacji są dwa 4-bitowe słowa binarne. Układ może wykonywać operacje arytmetyczne (np. dodawanie, odejmowanie) lub logiczne (np. sumowanie, iloczynowanie, sumowanie modulo 2). ALU wykonuje także operacje jednoargumentowe, jak np. negowanie, inkrementowanie, dekrementowanie lub operacje przesuwania.

Wśród sygnałów wejściowych ALU można wyróżnić dwie grupy: grupę wejść informacyjnych i grupę wejść sterujących. Na wejścia informacyjne podawane są dwa słowa, które są argumentami wykonywanej operacji. Na wejścia sterujące podawany jest 5-bitowy kod operacji, która ma być wykonana.



Rysunek 3.50. Końcówki 4-bitowego układu ALU

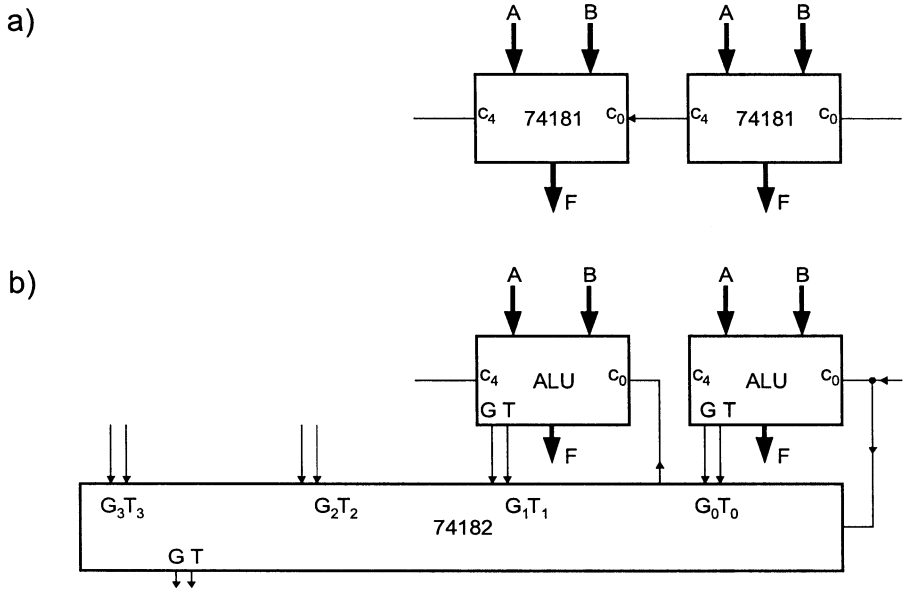
Na rysunku 3.50 pokazano oznaczenia końcówek układu 74181. Wejścia sterujące $s_3 - s_0$ wraz z wejściem ustawiającym tryb pracy M służą do programowania funkcji wykonywanych przez układ. W zależności od stanu wejścia M układ wykonuje operacje logiczne albo arytmetyczne. W tablicy opisu układu 74181 (por. tablica 3.11) zestawiono zbiór wszystkich funkcji wykonywanych przez ALU wraz z odpowiednimi wartościami sygnału M i $s_3 - s_0$. Argumenty operacji oznaczono przez A i B, a wynik operacji przez F. Zarówno argumenty jak i wynik są 4-bitowe. Nie wszystkie operacje zawarte w kolumnie „Funkcje arytmetyczne" ($M = 0$) mają sens, tzn. da się znaleźć ich sensowne zastosowanie. Operacje te znajdują się w wierszach 2, 3, 5, 6, 8, 9, 11, 12, 14 i 15 tablicy.

Wyjaśnienia wymagają jeszcze cztery wyjścia: c_4, G, T i $A = B$. Na wyjściu c_4 pojawia się sygnał przeniesienia wyjściowego powstającego w ostatnim stopniu 4-bitowego sumatora. Sygnał ten jest wykorzystywany do łączenia ze sobą kilku takich samych układów (por. rys. 3.51a).

Na wyjściach G i T pojawiają się sygnały przeniesień: przeniesienia generowanego G i przeniesienia transmitowanego T. Są to sygnały informujące o konfiguracjach argumentów i mogą być wykorzystane do generowania przeniesień. Sygnał G oznacza, że sygnały A i B mają takie kombinacje, że przeniesienie wychodzące zostanie wygenerowane bez względu na to jakie jest przeniesienie przychodzące. Sygnał T jest sygnałem warunkującym transmisję przeniesienia wchodzącego. Sygnały te służą do równoległego wyliczenia przeniesień wszystkich stopni sumatora zbudowanego z wielu 4-bitowych układów (podobnie jak

Tablica 3.11. Tablica funkcji wykonywanych przez układ SN 74181

	funkcje		
	logiczne	arytmetyczne	
wybór $s_3s_2s_1s_0$	$M = 1$	$M = 0$ $c_n = 1$	$M = 0$ $c_n = 0$
0000	F = $\overline{A}$	F = A	F = A plus 1
0001	F = $\overline{A+B}$	F = $A+B$	F = $(A+B)$ plus 1
0010	F = $\overline{A}B$	F = $A+\overline{B}$	F = $(A+\overline{B})$ plus 1
0011	F = 0	F = minus 1 (U2)	F = 0
0100	F = $\overline{AB}$	F = A plus $A\overline{B}$	F = A plus $A\overline{B}$ plus 1
0101	F = $\overline{B}$	F = $(A+B)$ plus $A\overline{B}$	F = $(A+B)$ plus $A\overline{B}$ plus 1
0110	F = $A \oplus B$	F = A minus B minus 1	F = A minus B
0111	F = $A\overline{B}$	F = $A\overline{B}$ minus 1	F = $A\overline{B}$
1000	F = $\overline{A}+B$	F = A plus AB	F = A plus AB plus 1
1001	F = $\overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
1010	F = B	F = $(A+\overline{B})$ plus AB	F = $(A+\overline{B})$ plus AB plus 1
1011	F = AB	F = AB minus 1	F = AB
1100	F = 1	F = A plus A*	F = A plus A plus 1
1101	F = $A+\overline{B}$	F = $(A+B)$ plus A	F = $(A+B)$ plus A plus 1
1110	F = $A+B$	F = $(A+\overline{B})$ plus A	F = $(A+\overline{B})$ plus A plus 1
1100	F = A	F = A minus 1	F = A



Rysunek 3.51. Połączenie układów 181 i 182

przeniesienia g_i i p_i w układzie sumatora jednobitowego). Na wyjściu $A = B$ pojawia się sygnał 1, gdy argumenty operacji są takie same.

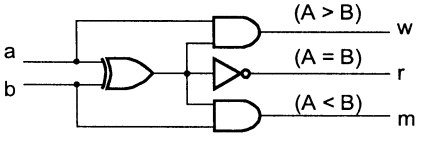
Na rysunku 3.51 pokazano dwie konfiguracje łączenia układów ALU: jedną z przeniesieniem generowanym kaskadowo i drugą z przeniesieniem równoległym. W pierwszym przypadku przeniesienie wyjściowe z 8-bitowego sumatora dwóch słów wejściowych A_i i B_i jest generowane kaskadowo i powstaje na wyjściu c_4 układu sumującego bardziej znaczące pozycje. Natomiast w drugim przypadku celem przyspieszenia pracy sumatora połączono dwa sumatory 4-bitowe z układem wyliczającym przeniesienie w sposób równoległy (układ SN 74182). Układ SN 74182 pozwala łączyć 4 układy SN 74181.

3.3.4. Komparatory

Komparatorem nazywać będziemy układ służący do porównania dwóch liczb, w szczególności zapisanych w kodzie NKB. Najczęściej dokonuje się porównywania liczb reprezentowanych słowami o tej samej długości. Wynik porównania może być trojaki:

1) liczba A jest większa od liczby B,
2) liczba A jest mniejsza od liczby B,
3) liczba A jest równa liczbie B.

Taki wynik można zakodować na dwóch bitach, ale ze względu na wygodę posługiwania się układem komparatora częściej koduje się wynik za pomocą trzech sygnałów. Przyjmijmy tutaj, że będą to sygnały w, m i r. Sygnał $w = 1$, gdy $A > B$, sygnał $m = 1$, gdy $A < B$ i sygnał $r = 1$, gdy $A = B$. W pozostałych przypadkach sygnały te przyjmują wartość 0.



Rysunek 3.52. Komparator jednobitowy

Rozważmy najprostszy komparator, tj. komparator liczb jednobitowych. Na rysunku 3.52 pokazano układ komparatora jednobitowego. Można zauważyć, że na wyjściu bramki EXOR jest 1, gdy bity wejściowe są różne. Wtedy na wejściu jednego z iloczynów są dwie jedynki, a zatem na jego wyjściu jest jedynka. Gdy bity są równe, to na wyjściu r jest 1.

Projektując układ komparatora dwubitowego można, dla dwóch dwubitowych słów wejściowych A_1A_0 i B_1B_0, utworzyć tablicę prawdy jak pokazano w tablicy 3.12 i na tej podstawie zaprojektować układ. Można także zaprojektować układ iteracyjny. Pojedynczy blok takiego układu ma 5 wejść (A_i, B_i, w_i, m_i i r_i) i 3 wyjścia (w_{i+1}, m_{i+1} i r_{i+1}). W tablicy 3.13 pokazano tabele prawdy funkcji w_{i+1}, m_{i+1} i r_{i+1} zależnych od zmiennych A_i, B_i oraz w_i, m_i i r_i. Na podstawie tych zależności można wyznaczyć funkcje wyjściowe jako:

$$w_{i+1} = A_i\overline{B_i} + (A_i + \overline{B_i})w_i$$
$$m_{i+1} = \overline{A_i}B_i + (\overline{A_i} + B_i)m_i$$
$$r_{i+1} = (A_iB_i + \overline{A_i}\,\overline{B_i})r_i$$

Jak powiedziano wcześniej, istnieje zależność pomiędzy sygnałami wyjściowymi. Jedną z nich jest, że $r_i = w_i m_i$. Ponieważ zależność ta jest prawdziwa na każdym stopniu iteracji,

Tablica 3.12. Tablica prawdy komparatora dwubitowego

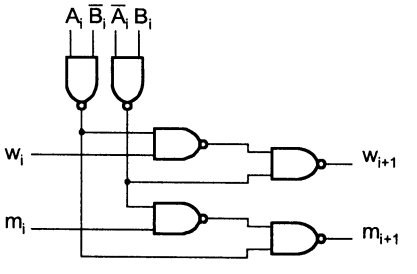
A_0	A_1	B_0	B_1	w A > B	r A = B	m A < B
0	0	0	0	0	1	0
1	0	0	0	1	0	0
0	1	0	0	1	0	0
1	1	0	0	1	0	0
0	0	1	0	0	0	1
1	0	1	0	0	1	0
0	1	1	0	1	0	0
1	1	1	0	1	0	0
0	0	0	1	0	0	1
1	0	0	1	0	0	1
0	1	0	1	0	1	0
1	1	0	1	1	0	0
0	0	1	1	0	0	1
1	0	1	1	0	0	1
0	1	1	1	0	0	1
1	1	1	1	0	1	0

Tablica 3.13. Tablica prawdy komparatora iteracyjnego

w_i A > B	r_i A = B	m_i A < B	A_i	B_i	w_{i+1} A > B	r_{i+1} A = B	m_{i+1} A < B
1	0	0	0	0	1	0	0
1	0	0	1	0	1	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	1	0	0
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	0	0	1	0	0	1
0	1	0	1	1	0	1	0
0	0	1	0	0	0	0	1
0	0	1	1	0	1	0	0
0	0	1	0	1	0	0	1
0	0	1	1	1	0	0	1

to sygnały przeniesień pomiędzy stopniami mogą być tylko dwa. Wtedy otrzymuje się układ iteracyjny, którego pojedynczy blok pokazano na rysunku 3.53.

Powracając do komparatora dwubitowego można otrzymać go biorąc dwa bloki układu iteracyjnego i przyjmując ponadto, że $r_0 = 1$ oraz $w_0 = 0$ i $m_0 = 0$. Nietrudno wykazać, że układ ten będzie prostszy od układu realizującego tablicę 3.12.



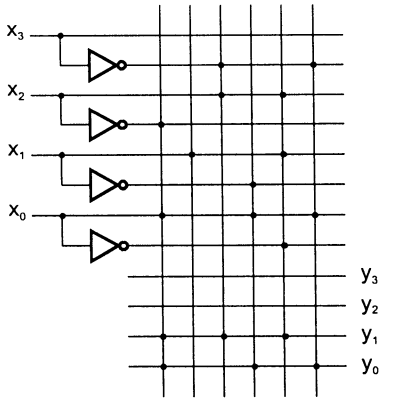
Rysunek 3.53. Układ realizujący jeden blok komparatora iteracyjnego

W serii układów TTL produkowane są 4-bitowe komparatory SN 7485, które można łączyć ze sobą kaskadowo. Komparatory te mają 8 wejść bitowych A_3, A_2, A_1, A_0, B_3, B_2, B_1 i B_0 i 3 wejścia w, m i r oraz 3 wyjścia w, m i r. Komparatory można łączyć kaskadowo w celu uzyskania układu komparacji większej liczby bitów.

3.4. Projektowanie układów cyfrowych za pomocą matrycowych układów programowalnych

Pokazana do tej pory metodyka projektowania układów cyfrowych prowadziła do budowania układów za pomocą jak najmniejszej liczby bramek, a w ogólnym przypadku do jak najmniejszej liczby układów scalonych. Współczesna technologia pozwala na zbudowanie większości układów cyfrowych w jednym układzie scalonym. Jeśli użytkownik projektuje układ, który będzie powielany w długich seriach, to warto rozważyć scalenie tego układu jako specjalizowanego układu scalonego. Są to tzw. układy ASIC (ang. *application specific integrated circuits*). Układy logiczne, które nie będą powielane w długich seriach, mogą być realizowane za pomocą programowanych uniwersalnych układów logicznych, tzw. układów PLD (ang. *programmable logic devices*). Producenci układów scalonych wprowadzili na rynek ten typ układów celem umożliwienia realizacji, stosunkowo tanio, niewielkich serii urządzeń cyfrowych. Jest wiele różnych układów PLD, ale wszystkie one budowane są jako układy matrycowe, w których można programować funkcje realizowane przez elementy w matrycy oraz połączenia pomiędzy tymi elementami. Producenci tych układów wskazują użytkownikom sposób programowania, aby łatwo mogli zaprogramować zaprojektowany przez siebie układ. Na rysunku 3.54 pokazano najprostszą strukturę programowaną tzw. PLA (ang. *programmable logic array*).

Układ PLA składa się z dwóch matryc połączeń. Załóżmy, że kropki widoczne na rysunku realizują dołączenie danej linii do wejść bramek. Kropki na poziomych liniach pierwszej (na rysunku górnej) matrycy realizują dołączenie wejść matrycy do bramek iloczynów logicznych, a kropki na poziomych liniach drugiej matrycy (na rysunku dolnej) realizują dołączenie pionowych linii do bramek sumy logicznej. W takim przypadku



Rysunek 3.54. Układ PLA

mówimy, że są to matryce AND-OR. Taka struktura pozwala na zrealizowanie sumacyjnej postaci funkcji. Jak widać na rysunku 3.54 zaprogramowanie układu pozwoliło na realizację funkcji:

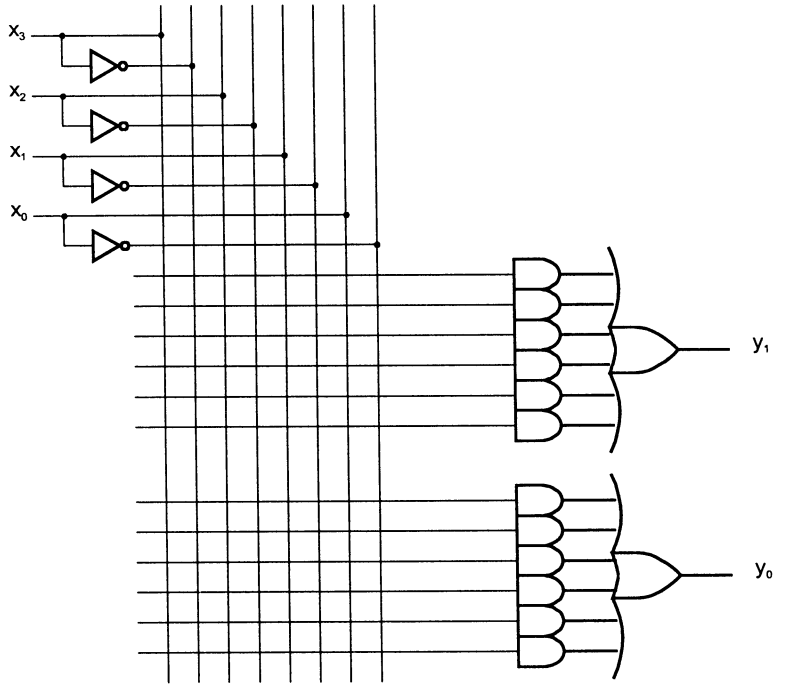
$$y_0 = x_0\bar{x}_2 + x_0\bar{x}_1 + x_0\bar{x}_3$$
$$y_1 = x_0\bar{x}_2 + x_2\bar{x}_3 + \bar{x}_0x_1x_2$$

Zatem zaprojektowanie odpowiedniej funkcji sprowadza się do tego, aby zaprogramować odpowiednie połączenia i wykonać je w strukturze PLD. Dokonuje się tego w specjalnych urządzeniach zwanych programatorami. Programatory przyjmują z komputera sterującego informację, gdzie ma zostać dokonane połączenie a gdzie nie. W tym celu przesyła się do programatora plik zawierający zera i jedynki (połączenia i brak połączeń) w standardowych formatach. Jednym z nich jest format JEDEC (z rozszerzeniem *.jed*). Plik taki przygotowuje się za pomocą specjalnych programów, o których będzie jeszcze dalej mowa.

Budowanie układów PLA napotyka na pewne ograniczenia. Matryce pokazane na rysunku 3.54 mogą służyć do realizacji 4 funkcji 4 zmiennych, przy czym minimalne postacie sumacyjne tych funkcji nie mogą zawierać więcej niż 6 iloczynów, gdyż tyle wynosi liczba pionowych linii łączących matryce. Zwykle w rzeczywistych układach matrycowych tych linii jest więcej. Typowym układem jest układ o 8 wejściach i 8 wyjściach. Typową liczbą linii łączących jest 24, choć dla 8 zmiennych możliwych iloczynów elementarnych może być 128 (w przypadku gdy liczba jedynek funkcji jest równa liczbie zer). Jeżeli trzeba zrealizować funkcję o większej liczbie jedynek, to należy wziąć postać iloczynową oraz układ PLA o zamienionych matrycach: pierwsza realizująca sumy a druga realizująca iloczyny. Jest to tzw. matryca OR-AND. Spotyka się także matryce NAND-NAND oraz NOR-NOR. Jeśli realizowane zadanie przekracza możliwości założonego układu PLA, to projektant może dobrać inny, większy układ. Jednak rozszerzanie dwóch matryc staje się trudne technologicznie i dlatego producenci układów scalonych zaproponowali inne rozwiązanie, zwane układem PAL.

W układach PAL wyeliminowano jedną z matryc i zastąpiono ją bramkami wyjściowymi z odpowiednią liczbą wejść. Przykładowo może to być matryca iloczynów oraz bramki sumy logicznej celem realizacji postaci sumacyjnej. Na rysunku 3.55 pokazano 4-wejściowy i 2-wyjściowy układ PAL wyposażony w 6-wejściowe bramki sumy. W takim układzie programuje się połączenia w matrycy 8×12 (8 pionowych linii zmiennych i ich negacji i 12 linii do dwóch 6-wejściowych bramek sumy).

Jednym z najbardziej popularnych układów PAL jest układ o symbolu PAL 16L8. Układ pokazano na rysunku 3.56. Z rysunku widać, że układ jest wyposażony w 10 wejść (końcówki 1 – 9 i 11) oraz 8 wyjść z 8-wejściowych bramek OR. Należy zauważyć, że programowana matryca iloczynów ma 64 linie poziome (8 bramek sum po 8 wejść) oraz 32 linie pionowe (20 linii zmiennych i ich negacji oraz 12 linii z wyjść 6 bramek sum i ich negacji). W takim układzie można realizować sprzężenie zwrotne z wyjścia bramek na wejście układu, co jest wykorzystywane do budowania asynchronicznych układów sekwencyjnych omówionych w dalszej części książki. Do wyjść bramek sumy dołączono tzw. trójstanowe bramki inwerterów. Są to bramki inwerterów sterowanych dodatkowym sygnałem (wejście od góry bramki), który jeśli jest 1, to bramka jest zwykłym inwerterem, a jeśli jest 0, to bramka jest odłączona od linii wyjściowej. Takie rozwiązanie stosuje się, aby dana linia mogła być albo linią wyjściową, albo wejściową (rozwiązanie stosowane w komputerach do budowy magistral — ang. *bus*). Dlatego w pokazanym układzie linie 13 do 18 mogą być także wejściami projektowanego układu. Bramki trójstanowe są sterowane przez jedną z linii matrycy AND.



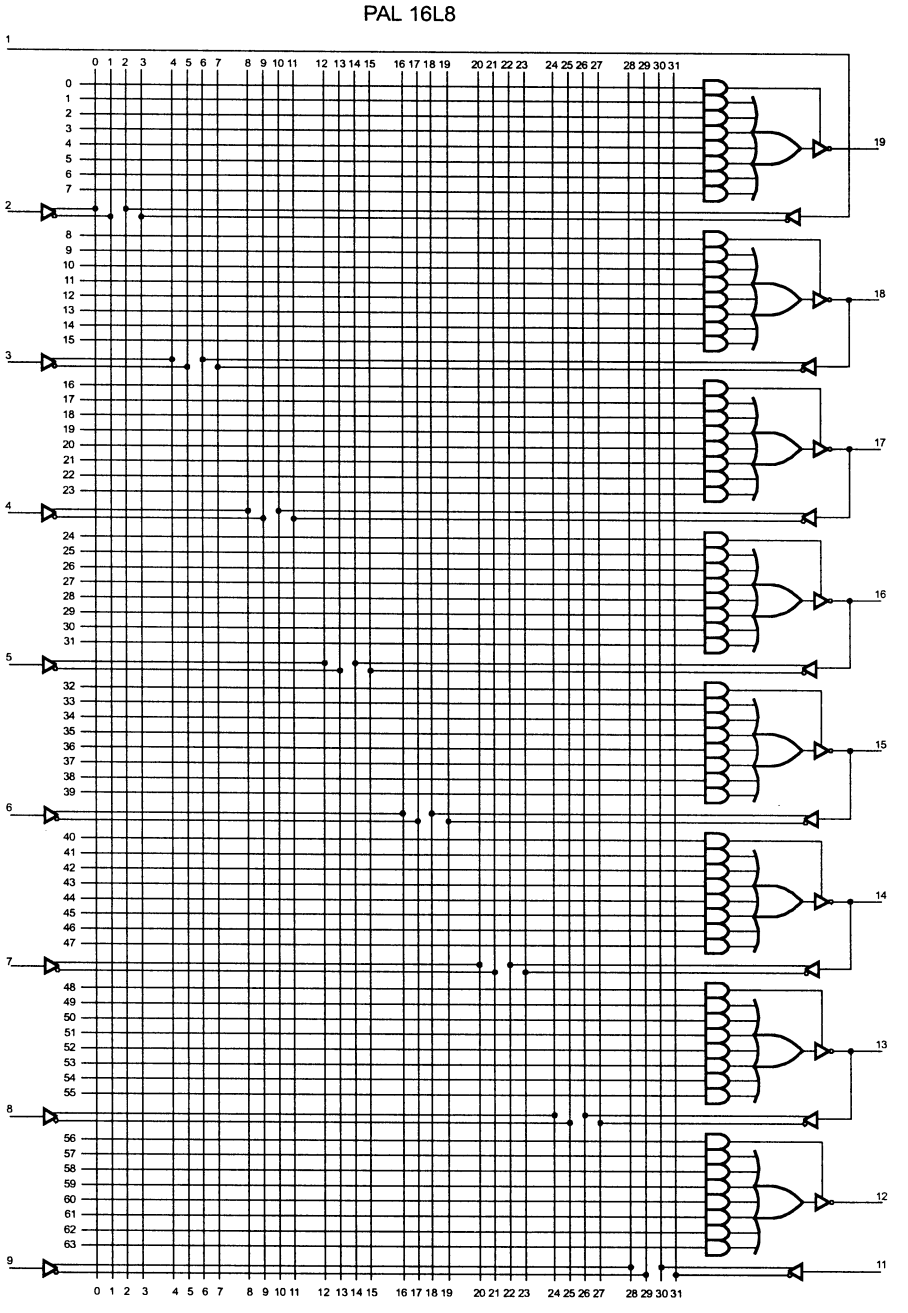
Rysunek 3.55. Układ PAL

W układzie można zmieścić 8 funkcji 10 zmiennych lub 7 funkcji 11 zmiennych, lub 6 funkcji 12 zmiennych lub itd. aż do 2 funkcji 16 zmiennych. Jak wynika z pokazanego rysunku wyjście każdej z bramek OR może odpowiadać jednej z projektowanych funkcji pod warunkiem, że jej minimalna postać sumacyjna nie zawiera więcej niż 7 iloczynów. Jeżeli ten ostatni warunek nie jest spełniony, to projektant może wykorzystać dwie bramki OR jako sumę iloczynów, ale wtedy zmniejsza się liczba wyjść układu.

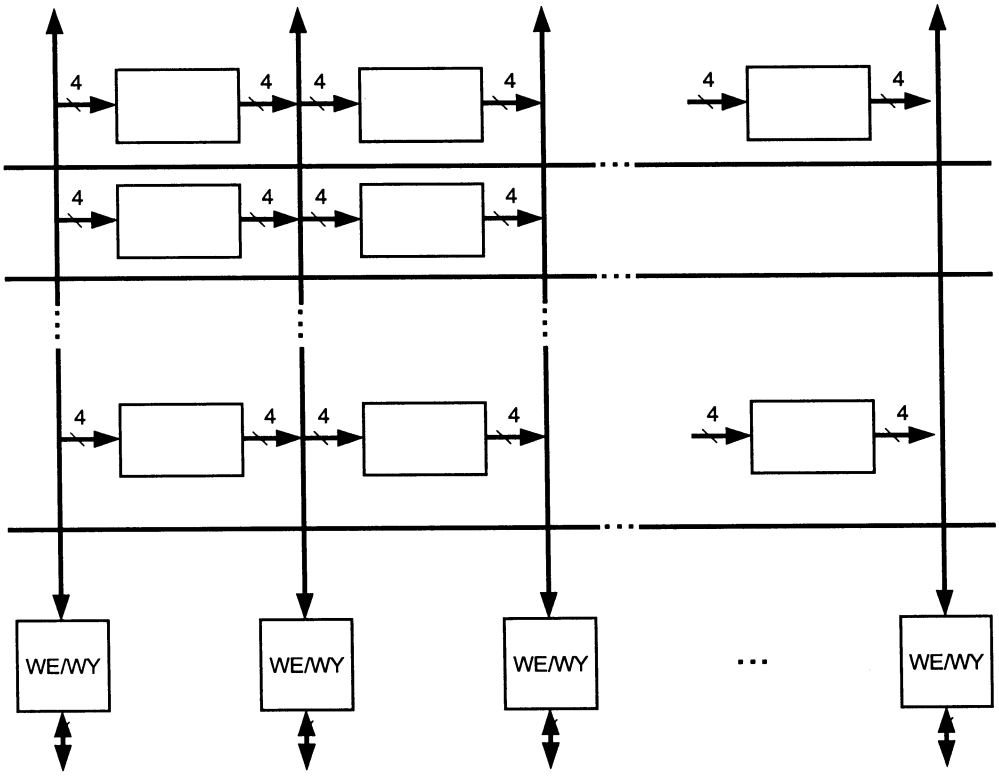
Producenci układów scalonych oferują szeroką gamę różnych innych układów programowalnych. Mogą to być układy z programowaną matrycą sumy i bramkami iloczynu (dla postaci iloczynowej funkcji), a mogą to być układy wyposażone w dodatkowe układy cyfrowe tzw. przerzutniki do projektowania synchronicznych układów sekwencyjnych. W zależności od zastosowania projektant może wybrać sobie odpowiedni układ, a podstawowym kryterium wyboru jest cena.

Innym stosowanym układem jest matryca komórek, w którym programuje się funkcje komórek i połączenia pomiędzy nimi. Układ taki nosi nazwę FPGA (ang. *field programmable gate array* — FPGA) i jest pokazany na rysunku 3.57.

Każda komórka składa się z uniwersalnych układów logicznych (dekodery, multipleksery), a komórki te można łączyć poprzez pokazane na rysunku magistrale połączeń. Uniwersalny układ logiczny można zaprogramować, aby wykonywał odpowiednią funkcję, a magistrale służą do realizacji połączeń pomiędzy komórkami. Przykładowa matryca z rysunku 3.57 ma komórki o 4 wejściach i 4 wyjściach. Taką komórkę mogą tworzyć np. cztery multipleksery o 4 wejściach sterujących. Każdy z nich realizuje funkcję 4 zmiennych, zależną od podanych zer i jedynek na wejścia danego multipleksera. Układ FPGA jest wyposażony także w specjalne bufory WE/WY, które umożliwiają różne rodzaje wejść



Rysunek 3.56. Układ PAL 16L8



Rysunek 3.57. Układ FPGA

i wyjść (np. wyjścia trójstanowe). Układ FPGA programuje się zwykle za pomocą specjalnych programów, które wyznaczają funkcje danej komórki i połączenia.

Projektowanie układów logicznych za pomocą układów programowanych odbywa się przy wykorzystaniu komputerowych systemów wspomagania projektowania CAD (ang. *computer aided design*). Zwykle pomagają one projektantowi przy minimalizacji funkcji, symulacji działania układu oraz przygotowaniu odpowiedniego pliku dla programatora. Dla zobrazowania Czytelnikowi tej procedury pokazany będzie przykładowy system wspomagania projektowania układów logicznych za pomocą układów LSI, tzw. system ABEL. System zostanie zaprezentowany na przykładzie zadania polegającego na zaprojektowaniu układu multipleksera o 4 wejściach i jednym wyjściu. To proste zadanie zostanie wykonane w dwóch wariantach, aby pokazać, że stosując ten system można dojść do wyniku wieloma drogami. Zapis w języku ABEL opatrzono komentarzem, aby łatwiej można było prześledzić tok projektowania.

Projektowany układ opisuje się w języku ABEL tworząc tzw. plik źródłowy. Składa się on z nagłówka, części deklaratywnej, części opisującej układ oraz z utworzonych przez projektanta testów projektowanego układu. Nagłówek zawiera nazwę układu. Jest to zwykle jedno słowo wpisane po instrukcji **module**. W następnej linii można użyć instrukcji **options** i wpisać za nią ciąg znaków w pojedynczym apostrofie. Instrukcja ta służy do wygodnego posługiwania się plikiem źródłowym w dalszym procesie przetwarzania. Następna instrukcja nagłówka to **title**. Po niej umieszcza się, w pojedynczym apostrofie, skrócony opis

projektowanego zadania. W części deklaratywnej projektant wybiera programowany układ, który chce wykorzystać (system może wskazać mu inny), przypisuje sygnały wejściowe i wyjściowe odpowiednim końcówkom, definiuje charakter stosowanych sygnałów (buforowany, zanegowany itp.), określa nazwy używanych stałych. W części opisującej układ projektant może opisać funkcje układu na różne sposoby. Może stosować równania boolowskie (instrukcja **equations**) a może wypełnić tablice prawdy. Część testująca zaczyna się instrukcją **test_vectors** i opisuje wzbudzenia układu i odpowiadające im spodziewane sygnały wyjściowe. Dalej przedstawiono dwa pliki źródłowe opisujące ten sam przykładowy układ, którym jest jednowyjściowy układ multipleksera 4-wejściowego.

```
module  MUX
title 'multiplekser 4 na 1
projektowal Andrzej Skorupski Warszawa luty 2000'
"linie rozpoczynajace sie podwojnym apostrofem sa pomijane
"brak deklaracji options
   mux          device 'PAL16L8';       "deklaracja ukladu

   x3,x2,x1,x0    pin    1,2,3,4;       "deklaracja wejsc
   s1,s0          pin    5,6;           "deklaracja wejsc sterujacych
   y              pin    14;            "deklaracja wyjscia
   X = .x.;                             "oznaczenie nieokreslonosci
   H = 1;                               "oznaczenie stanu 1 przez H
   L = 0;                               "oznaczenie stanu 0 przez L

equations
y = x0&!s0&!s1 # x1&s0&!s1 # x2&!s0&s1 # x3&s0&s1; "rownanie
                                        "boolowskie funkcji y
test_vectors  ( [s1, s0, x0, x1, x2, x3] -> y)
                [L,  L,  H,  X,  X,  X]  -> H; "testy sprawdzajace
                [L,  L,  L,  X,  X,  X]  -> L; "poprawnosc ukladu
                [L,  H,  X,  H,  X,  X]  -> H;
                [L,  H,  X,  L,  X,  X]  -> L;
                [H,  L,  X,  X,  H,  X]  -> H;
                [H,  L,  X,  X,  L,  X]  -> L;
                [H,  H,  X,  X,  X,  H]  -> H;
                [H,  H,  X,  X,  X,  L]  -> L;
   end
```

W części deklaratywnej podano nazwę modułu (w naszych przykładach jest to MUX) oraz jego charakterystykę (w naszych przykładach jest to multiplekser 4 na 1). Zadeklarowano układ matrycowy (w naszych przykładach jest to układ PAL16L8), a następnie przypisano jego końcówkom sygnały wejściowe i wyjściowe (w tym miejscu projektant musi posłużyć się katalogiem, aby podać numery końcówek). W części opisowej podano zależności pomiędzy sygnałami wejściowymi i wyjściowymi za pomocą równania boolowskiego.

W języku ABEL stosuje się następujące oznaczenia funktorów logicznych:

! — negacja
& — iloczyn
# — suma
$ — suma mod 2

Na końcu pliku podano tzw. wektory testowe. Projektant wybiera pewne wzbudzenia układu (wektory wejściowe) i określa odpowiadające im wektory wyjściowe. System sprawdza poprawność testów i sygnalizuje projektantowi ewentualne błędy.

System ABEL wyposażony jest w edytor tekstowy umożliwiający poprawianie pliku źródłowego, w symulator sprawdzający poprawność opisu układu za pomocą wektorów testowych, program minimalizujący wyrażenia boolowskie, kompilator sprawdzający poprawność pliku źródłowego, program przygotowujący dane dla programatora i dokumentację projektu oraz program wspomagający ewentualny wybór układu programowanego.

Innym możliwym opisem tego samego układu może być następujący plik źródłowy.

```
module MUX
title 'miltiplekser 4 na 1
projektowal Andrzej Skorupski Warszawa luty 2000'

    MUX     device 'PAL16L8';

    x3,x2,x1,x0     pin     1,2,3,4;
    s1,s0           pin     5,6;
    y               pin     14;
    X = .x.;
    s = [s1,s0];                    "deklaracja dwubitowego wektora

equations
    when (s == 0) then y = x0;      "wskazanie wartosci y dla
    when (s == 1) then y = x1;    "przypadku gdy s=0
    when (s == 2) then y = x2;
    when (s == 3) then y = x3;

test_vectors  ( [s1, s0,  x0, x1, x2, x3] -> y)
                [0,  0,  1,  X,  X,  X] -> 1;
                [0,  0,  0,  X,  X,  X] -> 0;
                [0,  1,  X,  1,  X,  X] -> 1;
                [0,  1,  X,  0,  X,  X] -> 0;
                [1,  0,  X,  X,  1,  X] -> 1;
                [1,  0,  X,  X,  0,  X] -> 0;
                [1,  1,  X,  X,  X,  1] -> 1;
                [1,  1,  X,  X,  X,  0] -> 0;
end
```

Plik ten zawiera wyrażenie typu **when... then** określające zachowanie układu, czyli wartość funkcji *y* dla różnych przypadków. Można zaobserwować także, że wyrażenie to posługuje się wektorem dwubitowym. Domniemane wartości tego wektora są wartościami dziesiętnymi.

W rozdziale 4 pokazane będą przedstawione inne przykłady zastosowania języka ABEL.

Układy sekwencyjne 4

4.1. Wstęp

Projektowane poprzednio układy cyfrowe opisywane były wyrażeniami boolowskimi ponieważ każdemu stanowi wejściowemu odpowiadał jednoznacznie stan wyjść. Istnieją układy cyfrowe, zwane układami sekwencyjnymi (albo automatami), które na identyczne pobudzenie, ale przychodzące w różnym czasie, odpowiadają w różny sposób, tj. dla identycznych wektorów wejściowych mogą pojawić się różne wektory wyjściowe. Wynika to stąd, że w różnych chwilach układ znajduje się w różnych stanach. Pod wpływem sygnałów wejściowych układ przechodzi od stanu do stanu, a sygnały wyjściowe zależą od stanu w jakim układ się znajduje, albo i od stanu i od sygnałów wejściowych, w zależności od rodzaju automatu. Wyróżnia się dwa rodzaje automatów:

— automaty, w których stany wyjściowe zmieniają się w czasie zmiany stanu automatu, a więc zależą tylko od stanu automatu, nazywane są automatami Moore'a,
— automaty, w których stany wyjściowe zmieniają się także w czasie zmiany stanu sygnałów wejściowych, a więc zależą i od stanu automatu i od stanu sygnałów wejściowych, nazywane są automatami Mealy'ego.

Z danego stanu, pod wpływem danego wektora wejściowego, układ zawsze przechodzi do ściśle określonego stanu. Dlatego, dla każdego stanu, podając na wejście układu określoną sekwencję wektorów wejściowych otrzymamy odpowiednią sekwencję wektorów wyjściowych. Aby to było możliwe układ musi rozpoznawać stan w jakim się znajduje.

Ze względu na konieczność uwzględnienia stanu, układy sekwencyjne opisuje się inaczej niż układy kombinacyjne. Dla każdego stanu i dla każdej kombinacji sygnałów wejściowych trzeba określić do jakiego stanu układ przechodzi. W ten sposób można utworzyć **tablicę przejść**, która w pełni opisuje wszystkie możliwe przejścia od stanu do stanu. W tablicy 4.1 przedstawiono tablicę przejść, która opisuje możliwe przejścia ze stanu do stanu, przykładowego układu sekwencyjnego, który ma cztery stany $s_1 - s_4$ i dwa wejścia x_1 i x_0.

W tablicy 4.1 jest opis zachowania układu we wszystkich możliwych przypadkach. W pierwszym wierszu tablicy widać do jakich stanów układ przechodzi będąc w stanie s_1. Jeśli wzbudzenie układu jest 00, to automat przechodzi do stanu s_2, jeśli wzbudzenie układu jest 01, to automat przechodzi do stanu s_3, jeśli wzbudzenie układu jest 11, to automat przechodzi do stanu s_1 i jeśli wzbudzenie układu jest 10, to automat przechodzi do stanu s_4. Podobnie należy odczytywać pozostałe wiersze. Takie przedstawienie „zachowania się"

Tablica 4.1. Przykładowa tablica 4-stanowego automatu

$s\backslash x_1x_0$	00	01	11	10
s_1	s_2	s_3	s_1	s_4
s_2	s_1	s_4	s_1	s_3
s_3	s_4	s_1	s_2	s_1
s_4	s_1	s_3	s_2	s_2

Tablica 4.2. Przykładowe tablice wyjść automatu z rysunku 4.1:a) automatu Mealy'ego, b) automatu Moore'a

a)

$s\backslash x_1x_0$	00	01	11	10
s_1	0	1	1	0
s_2	1	1	1	1
s_3	0	0	0	0
s_4	0	1	1	0

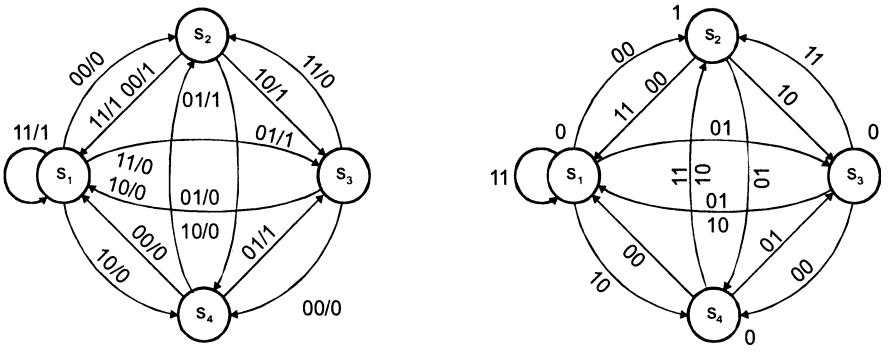
b)

s	y
s_1	0
s_2	1
s_3	0
s_4	0

automatu opisuje jedynie jego wewnętrzne działanie, tj. przejścia od stanu do stanu. Brak natomiast opisu zachowania się jego wyjść. Stany wyjściowe są opisane w drugiej tablicy, tzw. tablicy wyjść.

Postać tablicy wyjść zależy od rodzaju automatu. W tablicy 4.2 pokazano przykładowe tablice wyjść automatu Mealy'ego i automatu Moore'a dla 4-stanowego automatu o dwóch wejściach (może to być automat działający wg tablicy przejść z tablicy 4.1). Tablice wyjść automatu pokazane w tablicy 4.2 utworzono przy założeniu, że jest tylko jeden sygnał wyjściowy y. W innym przypadku w każdym miejscu tablicy należałoby umieścić ciąg bitów o długości równej liczbie sygnałów wyjściowych.

Podsumowując rozważania można stwierdzić, że układy sekwencyjne opisuje się za pomocą dwóch tablic: tablicą przejść i tablicą wyjść. Istnieją jednak jeszcze inne opisy automatów. W tym miejscu podano opis, tzw. graf automatu, a dalej (por. rozdz. 4.3) inny opis, tzw. wykres czasowy.



Rysunek 4.1. Grafy przykładowego automatu: a) automat Mealy'ego, b) automat Moore'a

Grafem automatu nazywać będziemy graf składający się z węzłów i strzałek. Liczba węzłów jest równa liczbie stanów, a strzałki pomiędzy węzłami obrazują odpowiednie przejścia pomiędzy stanami. W przypadku automatu Moore'a obok węzłów grafu są podawane odpowiednie stany sygnałów wyjściowych odpowiadające danemu stanowi. W przypadku automatu Mealy'ego sygnały wyjściowe podawane są obok strzałek, gdyż w ten sposób obrazowana jest ich zależność od sygnałów wejściowych. Na rysunku 4.1 pokazano grafy odpowiadające przykładowemu automatowi Moore'a i Mealy'ego.

Zarówno dla automatu Moore'a jak i automatu Mealy'ego wyróżnia się dwa możliwe rozwiązania. Rozwiązanie pierwsze polega na tym, że zmiana stanu automatu następuje tylko w pewnych chwilach określonych przez zewnętrzny generator (zegar) i drugie, gdzie zmiana stanu następuje w chwili zmiany stanu sygnałów wejściowych. Pierwsze rozwiązanie nazywane jest automatem synchronicznym, a drugie asynchronicznym.

Tablica 4.3. Tablica przejść i wyjść wymuszeń przerzutników typu *D*

a)

Q\D	0	1
0	0	1
1	0	1

b)

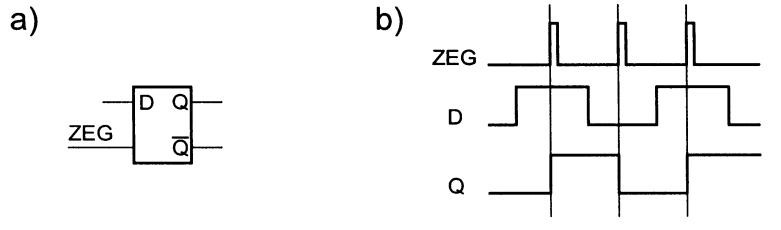
$Q^t \rightarrow Q^{t+1}$	D
0 → 0	0
0 → 1	1
1 → 0	0
1 → 1	1

Jak zostało wcześniej powiedziane sekwencyjne układy cyfrowe muszą pamiętać swój stan. W tym celu wykorzystuje się tzw. **przerzutniki** (ang. *flip-flop*). Są to elementarne dwustanowe układy sekwencyjne typu Moore'a. Za ich pomocą oraz za pomocą bramek można budować złożone automaty, posługując się metodami opisanymi w dalszej części książki. Dla ułatwienia projektowania trzeba rozwiązanie zakwalifikować do jednej z dwóch grup możliwych rozwiązań: układu synchronicznego lub asynchronicznego. W układach synchronicznych stosuje się przerzutniki synchronizowane dodatkowym sygnałem, tzw. sygnałem zegarowym. W takim przypadku stan przerzutników zmienia się tylko w chwilach wyznaczonych przez sygnał zegarowy. W układach asynchronicznych stosuje się przerzutniki zmieniające swój stan w czasie zmiany stanu sygnałów wejściowych. Proces projektowania takich automatów jest trudniejszy i jak pokazano dalej istnieją zjawiska, które nie występują w automatach synchronicznych. Dlatego najpierw omówiono proces projektowania automatów synchronicznych, a potem asynchronicznych.

4.2. Synchroniczne układy sekwencyjne

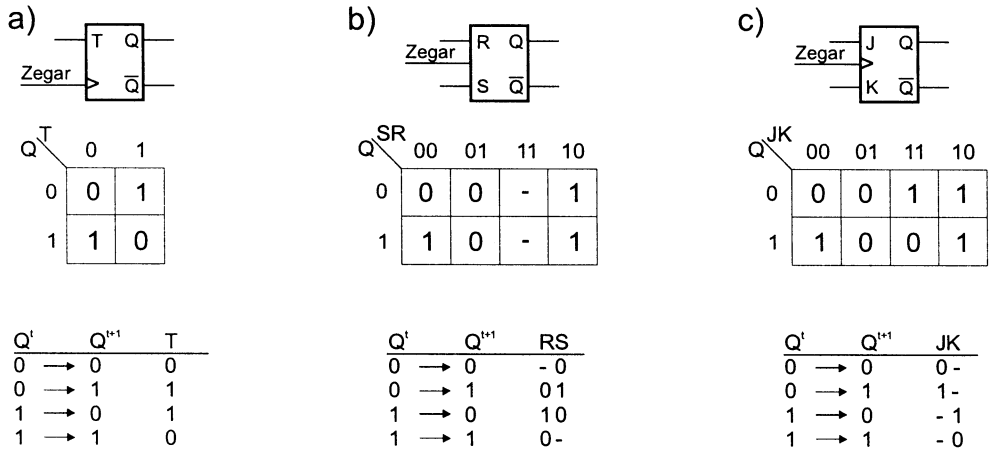
4.2.1. Przerzutniki synchroniczne

Na rysunku 4.2 przedstawiono symbol i wykres czasowy wyidealizowanego przerzutnika synchronicznego zwanego przerzutnikiem typu ***D*** od słowa dana (ang. *data*). Przerzutnik ten działa w taki sposób, że w momencie pojawienia się impulsu zegarowego przerzutnik



Rysunek 4.2. Symbol i wykres czasowy przerzutnika typu D

zapamiętuje stan jaki w tym czasie był na jego wejściu *D*. Mówimy, że zostało do niego wpisane 0 lub 1. Na rysunku 4.2b pokazano tzw. wykres czasowy, który przedstawia przebiegi sygnałów w czasie. Założono, że na początku przerzutnik był wyzerowany ($Q = 0$) i w pewnym momencie pojawiła się jedynka na jego wejściu. Układ nie zmienia swojego stanu aż do momentu pojawienia się impulsu zegarowego. Wtedy na wyjściu Q pojawia się jedynka. Podobnie jest w dalszej części wykresu, gdzie zero przychodzące na wejście *D* przerzutnika nie zmienia jego stanu, aż do momentu pojawienia się impulsu zegarowego.



Q \ T	0	1
0	0	1
1	1	0

Q \ SR	00	01	11	10
0	0	0	-	1
1	1	0	-	1

Q \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Q^t	Q^{t+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Q^t	Q^{t+1}	RS
0	0	- 0
0	1	0 1
1	0	1 0
1	1	0 -

Q^t	Q^{t+1}	JK
0	0	0 -
0	1	1 -
1	0	- 1
1	1	- 0

Rysunek 4.3. Oznaczenia przerzutników, ich tablice przejść oraz tablice wymuszeń: a) typu *T*, b) typu *RS*, c) typu JK

Przedstawiony przerzutnik jest dwustanowym automatem z jednym sygnałem wejściowym, a więc jego tablica przejść, pokazana jako tablica 4.3a, jest dwuwierszowa i dwukolumnowa. Bez względu na pierwotny stan przerzutnika jego następny stan zależy od wartości sygnału wejściowego. Jeśli na wejściu jest 0, to następny stan jest 0 (kolumna pierwsza), a jeśli na wejściu jest 1, to następny stan jest 1 (kolumna druga). W tablicy 4.3b przedstawiono tzw. tablicę wymuszeń, w której pokazano jakie wartości sygnałów wejściowych należy podać na wejście *D*, aby w przerzutniku zaszła pożądana zmiana. Takich zmian może być 4. Symbol Q^t oznacza stan pierwotny przerzutnika, a symbol Q^{t+1} stan następny, tj. po przyjściu impulsu zegarowego. W drugiej kolumnie tablicy wymuszeń podaje się wartości sygnału wejściowego powodujące odpowiednią zmianę. Tablice wymuszeń są przydatne w czasie projektowania automatów.

Oprócz przerzutników typu *D*, do projektowania układów sekwencyjnych używa się trzech innych rodzajów przerzutników, których symbole graficzne, tablice przejść i tablice wymuszeń pokazano na rysunku 4.3. Są to przerzutniki typu *T*, *RS* i *JK*.

Jednowejściowy przerzutnik typu ***T*** (ang. *trigger*) zmienia swój stan, w momencie pojawienia się impulsu zegarowego, jeżeli $T = 1$. Jeśli $T = 0$, to przerzutnik pozostaje w stanie pierwotnym. Z pierwszej kolumny tablicy przejść ($T = 0$) widać, że przerzutnik pozostaje w stanie, w którym był, a z drugiej ($T = 1$), że przerzutnik zmienia swój stan.

Dwuwejściowy przerzutnik typu ***RS*** (ang. *reset* i *set*) działa w taki sposób, że podanie jedynki na wejście *S* ustawia stan 1 na wyjściu *Q*, a podanie 1 na wejście *R* ustawia stan 0. Nie dopuszcza się przypadku aby $R = S = 1$ (jest to kombinacja zabroniona), co uwidocznione jest w kolumnie trzeciej tablicy przejść.

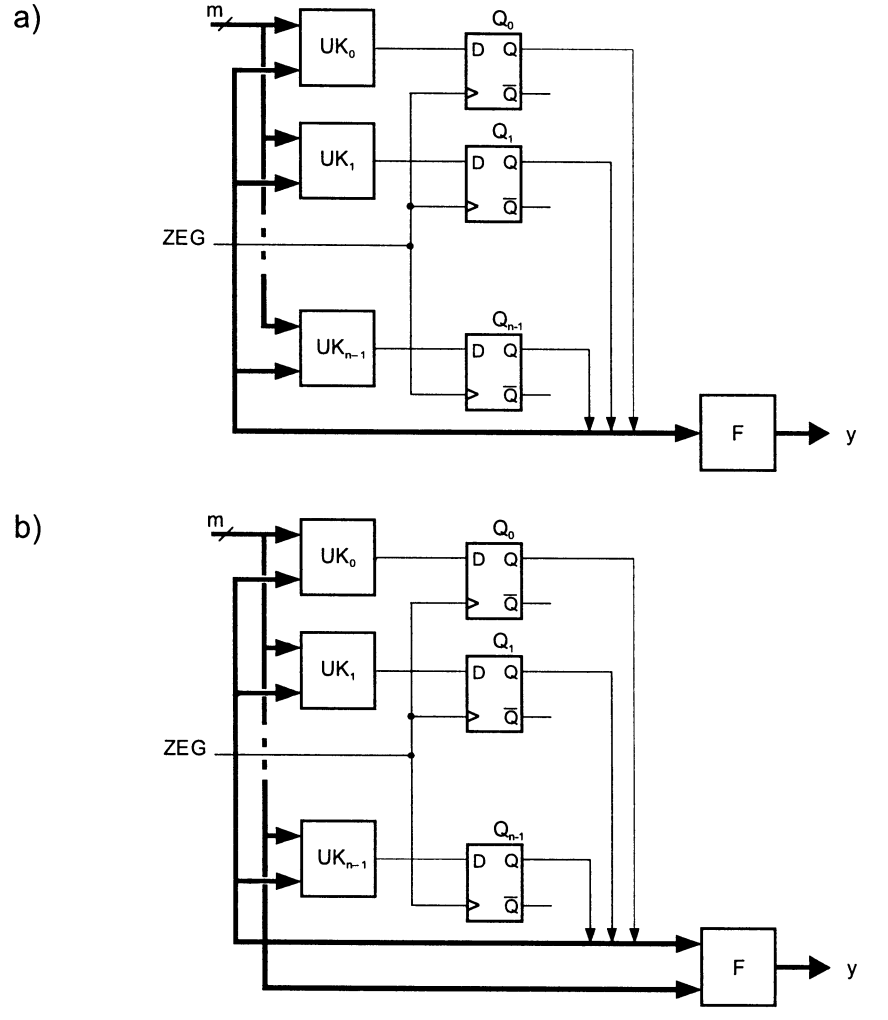
Nazwa dwuwejściowego przerzutnik typu ***JK*** pochodzi od imienia i nazwiska jego konstruktora (John Kilby), a działa on podobnie jak przerzutnik *RS*, z tą różnicą, że gdy $J = 1$ i $K = 1$, to zmienia się stan przerzutnika. W innych przypadkach wejście *J* działa jak wejście *S*, a wejście *K* jak wejście *R*.

Przerzutniki nazywane są czasem automatami elementarnymi. Złożone automaty budowane są z bramek i przerzutników. Najczęściej do budowy automatów synchronicznych używane są przerzutniki typu *D* i *JK*.

4.2.2. Struktura automatu synchronicznego

Załóżmy, że należy zrealizować k-stanowy automat synchroniczny. Zadaniem projektanta jest określenie wymaganej liczby przerzutników (n). Ponieważ przerzutnik jest automatem dwustanowym, to biorąc dwa przerzutniki Q_1 i Q_0 można zbudować automat czterostanowy. Jego stany będą określone przez stany przerzutników: 00, 01, 10 i 11. Liczba przerzutników k-stanowego automatu ma spełniać warunek, że 2^n musi być większe lub równe k. Przykładowo do realizacji 17-stanowego automatu potrzeba 5 przerzutników.

Po wybraniu liczby przerzutników należy wybrać jego typ. Jak zostało powiedziane najczęściej będą to przerzutniki typu D lub JK. Zwykle do budowy automatu używa się jednego typu przerzutnika, choć automat można zrealizować używając różnych typów prze-



Rysunek 4.4. Struktura automatu na przerzutnikach typu D: a) Moore'a , b) Mealy'ego

rzutników. W niniejszej książce przykłady będą rozwiązywane z użyciem jednego, wybranego typu przerzutnika.

Po wybraniu typu przerzutnika należy, na podstawie tablicy przejść, wyznaczyć ich tzw. **funkcje wzbudzeń**. Jeśli będą wybrane przerzutniki typu *T* lub *D*, to trzeba wyznaczyć *n* takich funkcji, po jednej dla każdego przerzutnika. Jeśli zaś będą wybrane przerzutniki typu *RS* lub *JK*, to trzeba wyznaczyć $2n$ takich funkcji (po dwie dla każdego przerzutnika). Natomiast na podstawie tablicy wyjść automatu należy wyznaczyć funkcje wyjść. Na rysunku 4.4 pokazano strukturę automatów synchronicznych Moore'a i Mealy'ego. Pogrubione linie oznaczają grupę sygnałów. Na przykład założono, że jest to automat *m*-wejściowy i grupę wszystkich linii wejściowych oznaczono jedną pogrubioną linią z zaznaczeniem ukośną kreską liczności grupy linii.

Jak zostało wcześniej powiedziane struktury automatu Moore'a i Mealy'ego różnią się realizacją funkcji wyjść *F*. W pierwszym przypadku funkcja ta zależy tylko od stanów przerzutników (rys. 4.4a) i jest *n*-argumentowa, a w drugim przypadku także od stanów na wejściach automatu (rys.4.4b) i jest $(n + m)$-argumentowa. W tym drugim przypadku każda zmiana sygnałów wejściowych, nawet bez zmiany stanów przerzutników, może spowodować zmianę sygnałów wyjściowych. Na rysunku pokazano strukturę automatu złożoną z *n* przerzutników typu *D*. Dlatego automat składa się z *n* układów kombinacyjnych oznaczonych na rysunku symbolami UK_i. Zadaniem projektanta jest, oprócz wyboru liczby i typu przerzutników, wyznaczenie tych funkcji oraz funkcji *F*. Metodologię postępowania przedstawiono w następnym punkcie.

4.2.3. Proces projektowania automatów synchronicznych

Proces projektowania automatu rozpoczyna się zwykle od analizy jego opisu słownego. Z opisu tego powinna wynikać liczba sygnałów wejściowych, wyjściowych oraz liczba stanów, a także możliwość określenia funkcji przejść oraz funkcji wyjść. Narzędziem pomocniczym do analizy opisu może być graf automatu. Na jego podstawie niekiedy łatwiej jest wyznaczyć tablicę przejść i wyjść. Po wyznaczeniu tych tablic, proces projektowania może być sformalizowany. Algorytm postępowania jest następujący:

1. Minimalizacja liczby stanów automatu.
2. Zakodowanie tablicy przejść i wyjść.
3. Wybór typu przerzutników.
4. Znalezienie funkcji wzbudzeń przerzutników.
5. Znalezienie funkcji wyjść.

Ad 1. Otrzymane, na podstawie analizy opisu słownego, tablice przejść i wyjść automatu (zwane dalej tablicami pierwotnymi) zawierają zwykle więcej stanów niż jest to konieczne. Wynika to z faktu, że tworząc graf automatu zakładamy pewne stany, które są nadmiarowe. Minimalizacja liczby stanów polega na tym, że sprawdza się czy nie istnieje grupa stanów, którą można zastąpić jednym stanem. Wyszukiwanie tych grup i zastępowanie ich jednym stanem, to proces minimalizacji, w wyniku którego otrzymuje się minimalną tablicę przejść (z minimalną liczbą stanów).

Ad 2. Otrzymana w poprzednim kroku minimalna tablica przejść musi zostać zakodowana, tzn. każdemu stanowi należy przypisać ciąg binarny. Ciąg binarny odpowiadający danemu stanowi określa stany przerzutników dla tego stanu. Liczba przerzutników konieczna do zrealizowania danego automatu jest równa długości tego ciągu. Wynikiem tego kroku jest zakodowana tablica przejść.

Ad 3. W procesie projektowania automatu należy zdecydować się na typ stosowanego przerzutnika. Kryterium wyboru typu przerzutników powinna być prostota układu, tj. minimalna liczba elementów potrzebna do realizacji funkcji wzbudzeń i funkcji wyjść. Spełnienie tego kryterium jest trudne, gdyż ze względu na brak metod formalnych, osiągnąć je można tylko przez porównanie różnych rozwiązań. W niniejszej książce wybór typu przerzutnika będzie dokonywany arbitralnie.

Ad 4. Po przyjęciu typu przerzutników należy znaleźć ich funkcje wzbudzeń. Na podstawie tablicy przejść wyznacza się funkcje, które tak wysterowywują dany przerzutnik, że jego stany zmieniają się zgodnie z zakodowaną tablicą stanów. Funkcje te są funkcjami boolowskimi, których argumentami są stany przerzutników i zmienne wejściowe. Technikę poszukiwania funkcji wzbudzeń pokazano w dalszym ciągu niniejszego rozdziału. Będzie ona zilustrowana przykładami.

Ad 5. Znalezienie funkcji wyjść odbywa się na podstawie tablicy wyjść. Ponieważ tablice wyjść automatu Moore'a i automatu Mealy'ego są różne, to i funkcje wyjść też są różne. Funkcja wyjść automatu Moore'a ma jako zmienne niezależne jedynie stany przerzutników, natomiast funkcja wyjść automatu Mealy'ego ma jako zmienne niezależne zarówno stany przerzutników, jak i zmienne wejściowe.

Przykład 4.1. Zaprojektować komparator dwóch liczb w kodzie NKB pojawiających się szeregowo na jego wejściach. Komparator ma w każdej chwili (dla każdej pary bitów) wskazywać równość liczb lub w przypadku gdy są one różne, ma wskazywać która z nich jest większa.

Interpretacja zadania. Szeregowe podawanie słów binarnych na wejścia układu oznacza, że kolejne bity słowa pojawiają się pojedynczo na wejściu układu, zgodnie z impulsami zegarowymi. Ponieważ porównanie dotyczy dwóch liczb, to projektowany układ jest dwuwejściowym automatem synchronicznym (nie licząc wejścia zegarowego). Wejścia te oznaczymy przez a_i i b_i, tak jak bity porównywanych słów A i B. Ważną rzeczą jest ustalenie kolejności pojawiania się bitów na tych wejściach. Gdyby sygnały wejściowe pojawiały się począwszy od najbardziej znaczącego bitu, to wynik porównania można by uzyskać w momencie pojawienia się pierwszych różnych bitów i rozwiązanie zadania jest wtedy bardzo proste. Natomiast na wejścia projektowanego automatu kolejne bity słów wejściowych pojawiają od najmniej znaczącego do najbardziej znaczącego. Wynik porównania pojawia się wówczas na końcu cyklu, gdy pojawią się najbardziej znaczące bity. Rozwiązując zadanie będziemy abstrahować od sygnałów sterujących rozpoczynających i kończących porównywanie. Oznacza to, że zakładamy ciągłość pracy automatu.

Analiza opisu zadania. Automat ma dwa sygnały wejściowe. Liczba sygnałów wyjściowych może być różna. W tym miejscu zrobimy takie samo założenie jak w rozdziale 3 dla komparatora kombinacyjnego i założymy wyjściowy kod „1 z 3" i wtedy mamy trzy wyjścia automatu. Oznaczymy je przez w ($A > B$), m ($A < B$) i r ($A = B$). Założymy, że projektowany automat jest typu Moore'a, a więc każdemu stanowi przypisuje się trzybitowe słowo wyjściowe w, m i r.

W każdej chwili automat powinien być w jednym z trzech stanów: s_1 — wskazującym na równość liczb, s_2 — wskazującym na to, że liczba A jest większa od B i s_3 — wskazującym na to, że liczba A jest mniejsza od B. Zatem dla każdego stanu trzeba wyznaczyć po cztery przejścia, tj. po jednym dla 4 kombinacji sygnałów wejściowych.

Gdy automat jest w stanie s_1, to pod wpływem sygnałów wejściowych $a_i = 0$ i $b_i = 0$ pozostaje on w tym stanie, gdyż stan s_1 wskazuje na równość liczb. Podobnie jest dla

Tablica 4.4. Tablica przejść komparatora szeregowego

S\aibi	00	01	11	10	w	m	r
s_1	s_1	s_3	s_1	s_2	0	0	1
s_2	s_2	s_3	s_2	s_2	1	0	0
s_3	s_3	s_3	s_3	s_2	0	1	0

sygnałów wejściowych $a_i = 1$ i $b_i = 1$. Sygnały wejściowe $a_i = 1$ i $b_i = 0$, w przypadku gdy automat jest w stanie s_1 świadczą o tym, że liczba A jest większa od B i dlatego automat przechodzi do stanu s_2. Podobnie jest dla sygnałów wejściowych $a_i = 0$ i $b_i = 1$. W tym przypadku (automat jest w stanie s_1) oznacza to, że liczba A jest mniejsza od B i dlatego automat przechodzi do stanu s_3. Przeprowadzając podobne rozumowanie dla pozostałych stanów otrzymamy tablicę przejść pokazaną w tablicy 4.4. Jest to tablica przejść o trzech wierszach (liczba stanów) i czterech kolumnach (liczba kombinacji sygnałów wejściowych) pokazana wraz z tablicą wyjść zgodną z wcześniejszym opisem.

Po określeniu tablicy przejść i tablicy wyjść należy wykonać sprawdzenie czy tablica nie zawiera za dużo stanów, tzn. czy nie istnieją grupy stanów, które można zastąpić jednym stanem. Jest to pierwszy krok projektowania automatu, czyli minimalizacja stanów. Wynikiem minimalizacji jest tablica przejść o najmniejszej liczbie stanów, która opisuje automat działający identycznie jak automat opisany słownie. Algorytm minimalizacji omówiono w następnym punkcie.

Zminimalizowana tablica przejść i wyjść stanowi punkt wyjścia do drugiego kroku projektowania automatu jakim jest zakodowanie jego stanów. Przez zakodowanie rozumiemy przypisanie każdemu stanowi różnych ciągów (słów) binarnych. Minimalna długość ciągu zależy od liczby stanów automatu. Jeżeli stanów jest nie więcej jak cztery, to wystarczy przyjąć ciąg dwubitowy. Jeżeli stanów jest nie więcej jak osiem, to wystarczy przyjąć ciąg trzybitowy. Jeśli jest k stanów, to długość ciągu n powinna spełniać nierówność $n > \log_2 k$.

W celu zakodowania trzech stanów automatu wystarczy przyjąć dwubitowe słowo. Istnieją 24 różne warianty zakodowania (pierwszy stan można zakodować na 4 sposoby i wówczas drugi stan na trzy sposoby a trzeci na dwa — co daje 24 różne zakodowania). Problem wyboru jednego z tych wariantów związany jest z poszukiwaniem rozwiązania optymalnego. Optymalność może mieć różny sens i w niniejszym tekście problem ten nie będzie dyskutowany. Tu będziemy przyjmować arbitralne zakodowanie i dla przykładowego automatu zostanie przyjęte, że stanowi s_1 zostanie przypisany ciąg 00, stanowi s_2 ciąg 01, stanowi s_3 ciąg 10.

Trzecim krokiem projektowania automatu jest wybór typu przerzutników. Ich liczba jest równa liczbie bitów słowa binarnego kodującego stany automatu. Inaczej mówiąc każdemu bitowi tego ciągu przypisuje się jeden przerzutnik. Tutaj przyjmiemy, że przykładowy komparator realizowany będzie za pomocą przerzutników typu D.

Czwarty krok projektowania automatu, to wyznaczanie funkcji wzbudzeń przerzutników, czyli poszukiwanie funkcji boolowskich, które sterując wejściami przerzutników będą powodowały, że ich stany będą zmieniać się tak jak wskazuje zakodowana tablica przejść.

Tablica 4.5. Zakodowana tablica przejść komparatora

Q_1Q_0 \ai bi	00	01	11	10
00	00	10	00	01
01	01	10	01	01
10	10	10	10	01

Tablica 4.6. Rozszerzona do 4 wierszy tablica przejść komparatora

$Q_1Q_0 \backslash a_i\, b_i$	00	01	11	10
00	00	10	00	01
01	01	10	01	01
11	--	--	--	--
10	10	10	10	01

Przykładowy automat jest trzystanowy, a więc do jego realizacji trzeba co najmniej dwóch przerzutników. Przyjmując najmniejszą liczbę przerzutników należy wyznaczyć dwie funkcje wzbudzeń, które oznaczymy przez D_0 i D_1. Funkcje te wyznacza się z zakodowanej tablicy przejść, która zgodnie z przyjętym założeniem zakodowania stanów pokazana została w tablicy 4.5. W zakodowanej tablicy przejść (tabl. 4.5) należy przypisać odpowiednim bitom ciągu kodującego odpowiednie przerzutniki. W naszym przykładzie bardziej znaczącemu bitowi (pierwszy z lewej) przypisano przerzutnik Q_1, a mniej znaczącemu bitowi przerzutnik Q_0. W celu znalezienia funkcji wzbudzeń tych przerzutników należy utworzyć mapy Karnaugha. Aby łatwiej było wyznaczyć te mapy warto rozszerzyć tablicę przejść o dodatkowe wiersze (w naszym przykładzie jeden) jak pokazano w tablicy 4.6.

Tablica 4.7. Tablice obrazujące zmiany stanów: a) przerzutnika Q_0, b) przerzutnika Q_1

a)

$Q_1\, Q_0 \backslash a_i\, b_i$	00	01	11	10
00	0	0	0	1
01	1	0	1	1
11	-	-	-	-
10	0	0	0	1

b)

$Q_1\, Q_0 \backslash a_i\, b_i$	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	-	-	-
10	1	1	1	0

Na podstawie tablicy przejść automatu tworzymy tablice przejść osobno dla każdego przerzutnika. W tablicach 4.7a i 4.7b pokazano dwie tablice przejść: jedną (tabl. 4.7a) dla przerzutnika Q_1 i drugą (tabl. 4.7b) dla przerzutnika Q_0.

Jak wynika z rozważań z punktu 4.2.1, funkcje wzbudzeń dla przerzutników typu D stosunkowo łatwo wyznaczyć bezpośrednio z tablicy przejść (por. rys.4.2). Mapy Karnaugha tych funkcji (por. rys. 4.5) są po prostu mapami powstałymi z odpowiednich bitów tablicy przejść przerzutnika (dla Q_1 jest to lewy bit z tablicy, a dla Q_0 jest to prawy bit). Poszukiwane funkcje wzbudzeń to:

$$D_0 = a_i\bar{b}_i + Q_0 a_i + Q_0\bar{b}_i$$
$$D_1 = \bar{a}_i b_i + Q_1\bar{a}_i + Q_1 b_i$$

a)

$Q_1\, Q_0 \backslash a_i\, b_i$	00	01	11	10
00	0	0	0	1
01	1	0	1	1
11	-	-	-	-
10	0	0	0	1

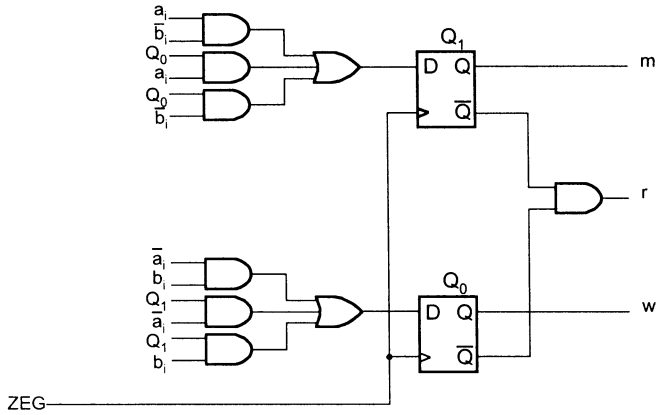
D_0

b)

$Q_1\, Q_0 \backslash a_i\, b_i$	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	-	-	-
10	1	1	1	0

D_1

Rysunek 4.5. Kompletne mapy Karnaugha funkcji wzbudzeń przerzutników typu D



Rysunek 4.6. Rozwiązanie zadania z przykładu 4.1

Każdą z tych funkcji można zrealizować stosując bramki sumy i iloczynów (pomijając bramki negacji) na czterech bramkach (jedna bramka sumy — trzywejściowa i trzy bramki dwuwejściowych iloczynów). Zatem do realizacji funkcji wzbudzeń potrzeba 8 bramek.

Następnie na podstawie tablicy wyjść trzeba wyznaczyć 3 funkcje wyjść. W naszym przykładzie poszukiwany automat jest automatem typu Moore'a, a więc sygnały wyjściowe w, m i r zależą tylko od stanów przerzutników Q_0 i Q_1. Funkcje wyjść to: $w = Q_0$, $m = Q_1$, $r = \overline{Q}_0\overline{Q}_1$.

Na rysunku 4.6 pokazano cały układ, który stanowi rozwiązanie przykładu 4.1 zrealizowane na przerzutnikach typu D. ⊠

W przedstawionym przykładzie posłużono się przerzutnikami typu D, dla których najłatwiej wyznaczyć funkcje wzbudzeń. Jak powiedziano już wcześniej drugim przerzutnikiem stosowanym do projektowania automatów synchronicznych jest przerzutnik typu JK i dalej pokazano sposób wyznaczania funkcji wzbudzeń dla tego typu przerzutników. Następnie, choć przerzutniki typu T stosowane są rzadko, to dla zachowania systematyczności opisu zostaną one także wykorzystane. Przerzutniki typu RS nie są używane do projektowania automatów synchronicznych (są wykorzystywane do projektowania automatów asynchronicznych) i tu zostaną pominięte.

Dla każdego przerzutnika typu JK trzeba wyznaczyć dwie funkcje wzbudzeń: jedną dla wejścia J i drugą dla wejścia K. Aby wyznaczyć te funkcje należy posłużyć się jednym z opisów funkcji boolowskich przedstawionych w rozdziale 3. Opis wybiera się w zależności od liczby zmiennych, a więc od wielkości tablicy przejść. Jeśli automat ma wiele stanów i wiele sygnałów wejściowych, to stosuje się komputerowe wspomaganie projektowania (por. rozdz. 4.4). Natomiast dla automatów o niewielkiej liczbie stanów i niewielkiej liczbie wejść można posłużyć się mapą Karnaugha.

Procedura wyznaczania map Karnaugha funkcji wzbudzeń przerzutników JK automatu o czterech stanach i dwóch sygnałach wejściowych jest następująca:

1. Ułożyć wiersze zakodowanej tablicy przejść według kodu Graya. Jeśli stanów jest mniej niż 4, to w nieistniejącym wierszu wpisać wartości nieokreślone.
2. Ułożyć kolumny tablicy przejść według kodu Graya.

a)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	0	0	0	1
01	-	-	-	-
11	-	-	-	-
10	0	0	0	1

J_0

b)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	-	-	-	-
01	0	1	0	0
11	-	-	-	-
10	-	-	-	-

K_0

c)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	-	-	-
10	-	-	-	-

J_1

d)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	-	-	-	-
01	-	-	-	-
11	-	-	-	-
10	0	0	0	1

K_1

Rysunek 4.7. Mapy Karnaugha funkcji wzbudzeń przerzutników *JK*

3. Narysować 4 mapy Karnaugha dla funkcji wzbudzeń obu przerzutników J_1, K_1, J_0 i K_0. Wierszom mapy Karnaugha przypisać zmienne Q_1 i Q_0, a kolumnom zmienne wejściowe, w taki sposób, aby ułożenie pól map Karnaugha odpowiadało ułożeniu pól tablicy przejść. Oznacza to, że oznaczenie wierszy i kolumn map Karnaugha i tablicy przejść ma być takie samo.
4. Wypełnić mapy Karnaugha posługując się odpowiednią tablicą wymuszeń z rysunku 4.3 (w tym przypadku dla przerzutnika *JK*).

Na rysunku 4.7a, b, c i d pokazano wynik zastosowania omawianej procedury dla zakodowanej tablicy przejść automatu z przykładu 4.1.

Z wyznaczonych map Karnaugha wynika, że poszukiwane funkcje to:

$$J_0 = a_i \overline{b_i}$$
$$K_0 = \overline{a_i} b_i$$
$$J_1 = \overline{a_i} b_i$$
$$K_1 = a_i \overline{b_i}$$

Rozwiązanie to wymaga dwóch bramek (dwie dwuwejściowe bramki iloczynu). Porównując to rozwiązanie z otrzymanym rozwiązaniem na przerzutnikach typu *D*, gdzie funkcje wzbudzeń były zrealizowane na ośmiu bramkach, widać, że korzystniejsze jest rozwiązanie na przerzutnikach typu *JK* niż na przerzutnikach typu *D*. Stąd można wysnuć wniosek, że wybierając typ przerzutnika nie należy kierować się liczbą poszukiwanych funkcji wzbudzeń (wtedy lepsze są przerzutniki typu *D*), ale złożonością funkcji wzbudzeń. Zwykle funkcje wzbudzeń dla przerzutników typu *JK* mają miejsca nieokreślone i dlatego funkcje te mogą być prostsze niż dla przerzutników typu *D*.

a)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	-	-	-	-
10	0	0	0	1

Q_1

b)

$Q_1 Q_0$ \ $a_i b_i$	00	01	11	10
00	0	0	0	1
01	0	1	0	0
11	-	-	-	-
10	0	0	0	1

Q_0

Rysunek 4.8. Mapy Karnaugha funkcji wzbudzeń przerzutników typu T dla automatu z przykładu 4.1: a) dla przerzutnika Q_1, b) dla przerzutnika Q_0

Rozpatrzmy teraz rozwiązanie na przerzutnikach typu T. Stosując analogiczną procedurę wyznaczania funkcji wzbudzeń jak dla przerzutników typu JK można otrzymać wynik w postaci map Karnaugha pokazanych na rysunku 4.8a i b.

Z map Karnaugha można znaleźć funkcje wzbudzeń:

$$T_0 = \overline{a_i} b_i Q_0 + a_i \overline{b_i}\,\overline{Q_0}$$

$$T_1 = \overline{a_i} b_i \overline{Q_1} + a_i \overline{b_i} Q_1$$

Każdą z tych funkcji można zrealizować na trzech bramkach (jedna bramka sumy dwuwejściowa i dwie bramki iloczynów trzywejściowych) i dlatego do realizacji obu funkcji potrzeba jedynie 6 bramek. Daje to, w tym konkretnym przypadku, lepsze rozwiązanie niż na przerzutnikach typu D. Złożoność funkcji wzbudzeń otrzymywanych dla danego automatu silnie zależy od sposobu zakodowania stanów automatu. Dla innego zakodowania wynik porównania rozważanych rozwiązań mógłby być inny.

Na koniec tych rozważań warto zwrócić uwagę na różnice jakie występują przy projektowaniu automatów Moore'a i Mealy'ego. Jest prawdziwe twierdzenie, że każdą tablicę przejść i wyjść automatu Moore'a można zamienić na tablicę przejść i wyjść automatu Mealy'ego. Prawdziwe jest też twierdzenie odwrotne. Zamieńmy teraz tablicę przejść i wyjść automatu Moore'a z przykładu 4.1 na tablicę przejść i wyjść automatu Mealy'ego.

Tablica przejść automatu Mealy'ego jest taka sama jak równoważnego mu automatowi Moore'a. Natomiast tablicę wyjść automatu Mealy'ego tworzy się w taki sposób, że jej wymiar ma być taki sam jak tablicy przejść, a w odpowiednie jej pola wpisuje się sygnały wyjściowe takie jakie są przypisane stanom automatu Moore'a w tych polach. Zilustrowano to w tablicy 4.8. W tablicy 4.8a przedstawiono tablicę przejść, a w tablicy 4.8b tablicę wyjść.

Sygnały wyjściowe automatu Moore'a zmieniają się w innych chwilach niż sygnały wyjściowe automatu Mealy'ego. Sygnały wyjściowe automatu Moore'a zmieniają się w mo-

Tablica 4.8. Tablice przejść i wyjść automatu Mealy'ego równoważne tablicom automatu Moore'a z tablicy 4.4

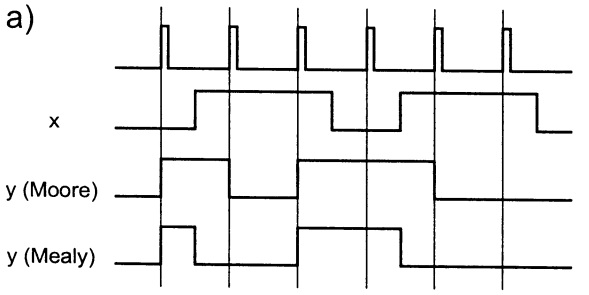
a)

S \ $a_i b_i$	00	01	11	10
s_1	s_1	s_3	s_1	s_2
s_2	s_2	s_3	s_2	s_2
s_3	s_3	s_3	s_3	s_2

b)

S \ $a_i b_i$	00	01	11	10
s_1	001	010	001	100
s_2	100	010	100	100
s_3	010	010	010	100

a)



b)

S \ x	0	1	x
1	3	2	0
2	3	1	0
3	3	1	1

c)

S \ x_1x_0	0	1
1	3/1	2/0
2	3/1	1/0
3	3/1	1/0

Rysunek 4.9. Zmiany na wyjściach automatów Moore'a i Mealy'ego: a) wykres czasowy, b) tablica przejść i wyjść automatu Moore'a, c) tablica przejść i wyjść automatu Mealy'ego

mentach zmiany stanu sygnału zegarowego (albo narastającego zbocza, albo opadającego zbocza impulsu zegarowego, w zależności od rodzaju przerzutnika), natomiast sygnały wyjściowe automatu Mealy'ego zmieniają się albo w momencie zmiany stanu sygnału zegarowego (zmiany stanu automatu), albo w momencie zmiany stanów na wejściach automatu. Przypadki te pokazane są na rysunku 4.9. Z rysunku 4.9a widać, że pierwsza i trzecia zmiana sygnału wyjściowego automatu Mealy'ego jest wywołana przez zmianę sygnału zegarowego, a druga i czwarta zmiana — przez zmianę sygnału wejściowego. Interpretacja tego zjawiska jest taka, że równoważne automaty Moore'a i Mealy'ego nie działają identycznie, gdyż momenty zmiany sygnałów wyjściowych są różne. Projektanci automatów muszą ten fakt uwzględniać. W tablicy przejść i wyjść opisującej automat Mealy'ego zastosowano konwencję umieszczenia w jednej tablicy: zarówno tablicy przejść jak i tablicy wyjść.

Zamianę tablicy przejść i wyjść automatu Mealy'ego na tablice automatu Moore'a wykonuje się w taki sposób, że każdej parze stan — wyjście automatu Mealy'ego przypisuje się stan automatu Moore'a. Dla tablicy przejść i wyjść automatu Mealy'ego pokazanej w tablicy 4.9a jest 5 takich par, którym przypisano stany automatu Moore'a w następujący sposób:

$$s_1/0 \quad — \quad A_1$$
$$s_1/1 \quad — \quad A_2$$
$$s_2/0 \quad — \quad A_3$$
$$s_3/0 \quad — \quad A_4$$
$$s_3/1 \quad — \quad A_5$$

Tablica 4.9. Tablice przejść i wyjść automatu Moore'a i Mealy'ego

a)

		0	1	
	s_1	$s_2/0$	$s_3/1$	
	s_2	$s_1/0$	$s_3/0$	
	s_3	$s_1/1$	$s_2/0$	

b)

		0	1	y	
	A_1	A_3	A_5	0	
	A_2	A_3	A_5	1	
	A_3	A_1	A_4	0	
	A_4	A_2	A_3	0	
	A_5	A_2	A_3	1	

Wtedy można utworzyć 5-stanową tablicę przejść i wyjść automatu Moore'a w taki sposób, że stan następny automatu automatu Moore'a jest stanem następnym względem stanu s_i z danej pary dla automatu Mealy'ego, a stan wyjścia jest określony przez drugi element pary. W tablicy 4.9b pokazano tablicę przejść i wyjść automatu Moore'a.

Ponieważ występują wyżej opisane różnice w działaniu obu typów automatów, to projektanci analizując dane zadanie muszą zastanowić się czy warunki zadania nie narzucają typu automatu.

4.2.4. Minimalizacja liczby stanów automatów synchronicznych

Otrzymana podczas procesu projektowania tablica przejść i wyjść automatu może zawierać dwa stany (lub więcej), które można zastąpić jednym. Mówimy wtedy, że istnieje równoważny danemu automat o mniejszej liczbie stanów. Znalezienie takich stanów prowadzi w wielu przypadkach do znacznego uproszczenia projektowanego automatu, a w szczególności do zmniejszenia liczby przerzutników. Procedura minimalizacji liczby stanów stanowi integralną część procesu projektowania każdego automatu.

Tablica 4.10. Tablice przejść automatu: a) pierwotna, b) zastępcza, c) minimalna

a)

s \ x_1x_0	00	01	11	10	y
1	1	1	2	3	0
2	1	2	3	2	1
3	3	1	2	1	0

b)

s \ x_1x_0	00	01	11	10	
a	a	a	2	a	0
2	a	2	a	2	1
a	a	a	2	a	0

c)

s \ x_1x_0	00	01	11	10	
a	a	a	2	a	0
2	a	2	a	2	1

Niech będzie dana tablica przejść i wyjść trzystanowego automatu Moore'a jak pokazano w tablicy 4.10a. Można zauważyć, że gdy oznaczenia stanów 1 i 3 zastąpi się literą *a*, to otrzyma się tablicę pokazaną w tablicy 4.10b, w której dwa wiersze (pierwsze i trzeci) są takie same. W takim przypadku mówimy, że stany 1 i 3 są stanami równoważnymi i można je zastąpić jednym stanem. Powstanie wtedy zminimalizowana tablica dwustanowa pokazana w tablicy 4.10c. Z opisu słownego automatu często wynika, że automat może znajdować się w stanie, w którym nie pojawiają się pewne wymuszenia. Przykładowo w automacie sterującym telewizorem, gdy jest włączona telegazeta, to nie reaguje on na zmiany numeru kanału. W takich przypadkach mamy do czynienia z automatami niezupełnymi (nie w pełni określonymi). W tablicy 4.11a pokazano tablicę przejść i wyjść takiego automatu. Przyglądając się tej tablicy można zauważyć, że stany *a* i *c* można zastąpić jednym, choć w pierwszej i czwartej kolumnie kreski wskazujące na nieokreśloność przejścia należy zastąpić odpowiednio literami *b* i *c*. Takie stany są nazywane zgodnymi (w odróżnieniu od równoważnych). W przykładowej tablicy można także zauważyć, że stany *b* i *d* można by zastąpić jednym pod warunkiem zastąpienia stanów *a* i *c* jednym stanem (patrz pierwsza kolumna tablicy). Zgodność stanów *b* i *d* jest zgodnością warunkową, a warunkiem jest zgodność stanów *a* i *c*. Zminimalizowana dwustanowa tablica przejść i wyjść tego automatu jest pokazana w tablicy 4.11b.

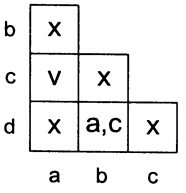
Tablica 4.11. Tablice przejść automatu niezupełnego: a) pierwotna, b) minimalna

a)

s \ x_1x_0	00	01	11	10	
a	b	d	c	-	0
b	a	d	b	-	1
c	-	d	a	c	0
d	c	d	d	a	1

b)

s \ x_1x_0	00	01	11	10	
x(a,c)	y	y	x	x	0
y(b,d)	x	y	y	x	1



Rysunek 4.10. Trójkątna tablica skracania

Proces minimalizacji liczby stanów automatów polega na wyszukiwaniu takich grup stanów, które można zastąpić jednym i które powodują, że tablica przejść i wyjść automatu składa się z najmniejszej możliwej liczby stanów. Aby spełnić te warunki, proces minimalizacji trzeba przeprowadzić systematycznie. W tym celu tworzy się tzw. trójkątną tablicę skracania. Na rysunku 4.10 pokazano taką tablicę dla automatu opisanego tablicą 4.11a. Tablica skracania ma tyle kratek ile par stanów należy porównywać (w przypadku automatów czterostanowych kratek jest sześć). Dla n stanów kratek jest $n(n-1)/2$. W odpowiednie kratki tablicy skracania wpisuje się jeden z trzech znaków:

1) zgodność stanów (w tablicy zaznaczono ten fakt znakiem v),
2) niezgodność stanów (w tablicy zaznaczono ten fakt znakiem x),
3) zgodność warunkową (w tablicy zaznaczono ten fakt wpisując pary stanów, których zgodności są warunkami zgodności danej pary).

Po takim zapisaniu tablicy należy iteracyjnie sprawdzać warunki zgodności i w przypadku ich niespełnienia (para stanów stanowiąca warunek jest niezgodna) zgodność warunkową należy zamienić na niezgodność (daną parę stanów zaznaczyć znakiem x).

Konieczność iteracyjnego procesu sprawdzania warunków wynika z tego, że podczas kolejnego sprawdzania kratek z warunkami trójkątnej tablicy skracania stwierdzenie niezgodności jakiejś pary może nastąpić później niż sprawdzenie kratki zawierającej tę parę jako warunek. Dlatego proces iteracyjnego sprawdzania należy zakończyć, gdy w danym cyklu nie zostanie stwierdzona niezgodność żadnej pary. Przykłady podane dalej zilustrują powyższe uzasadnienie.

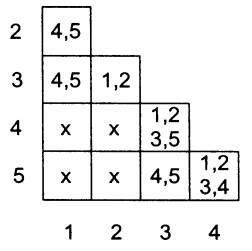
Następny krok procesu minimalizacji to znalezienie grup stanów zgodnych o jak największej liczności zwanych maksymalnymi grupami (zbiorami) stanów zgodnych. Jeżeli liczność grupy wynosi 3, to muszą istnieć 3 pary stanów zgodnych wchodzących do tej grupy. Jeżeli liczność grupy wynosi 5, to musi istnieć 10 par stanów zgodnych wchodzących do tej grupy. W ogólnym przypadku dla liczności n musi istnieć $n(n-1)/2$ par stanów zgodnych wchodzących do tej grupy. Algorytm poszukiwania maksymalnych grup stanów zgodnych będzie przedstawiony w przykładzie 4.3.

Ponieważ każdy stan może być elementem wielu grup, to liczba grup stanów zgodnych może być większa niż liczba stanów automatu równoważnego danemu o minimalnej liczbie stanów. Dlatego każdemu stanowi, który jest elementem więcej niż jednej grupy należy wybrać grupę, której elementem pozostanie. Kryterium wyboru stanowią najczęściej warunki zgodności stanów. Dany stan przypisuje się do tej grupy, w której, wraz z innymi stanami tworzy pary, które stanowią warunki zgodności par przypisanych do jednej grupy. Aby lepiej zrozumieć ten problem, proces ten będzie zilustrowany przykładami.

a)

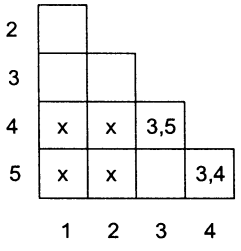
	0	1	y
1	2	4	0
2	1	5	0
3	2	5	-
4	1	3	1
5	2	4	1

b)



Rysunek 4.11. Tablica przejść automatu i trójkątna tablica skracania

Przykład 4.2. Dana jest tablica przejść i wyjść automatu Moore'a pokazana na rysunku 4.11a. Trójkątna tablica skracania jest pokazana na rysunku 4.11b. Poszukując maksymalnych grup stanów zgodnych otrzymamy dwie grupy: 1, 2, 3 i 3, 4, 5. Należy wybrać grupę, do której zostanie przypisany stan 3. W tym przypadku występuje pełna zgodność stanów 1 i 2 oraz 4 i 5. Zatem z trójkątnej tablicy skracania można usunąć na pewno spełnione te warunki. Pozostaną warunki, które stanowią kryterium wyboru grupy dla stanu 3. Taka uproszczona tablica trójkątna jest przedstawiona na rysunku 4.12.



Rysunek 4.12. Uproszczona tablica skracania z wykreślonymi warunkami spełnionymi

Jak widać z rysunku 4.12 pozostały dwa warunki zgodności: 3, 4 i 3, 5. Aby warunki te były spełnione, stan 3 musi zostać przypisany do grupy 3, 4, 5. Dlatego rozwiązaniem przykładowego zadania jest dwustanowy automat o stanach odpowiadających grupom: 1, 2 i 3, 4, 5. ⊠

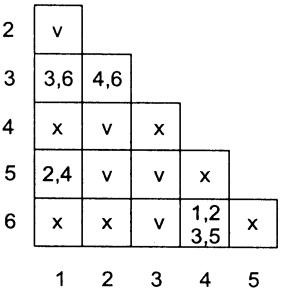
Przykład 4.3. Niech będzie dany 6-stanowy automat Mealy'ego określony tablicą przejść i wyjść pokazaną na rysunku 4.13a. Jego trójkątna tablica skracania jest pokazana na rysunku 4.13b. Z tablicy skracania można znaleźć maksymalne grupy stanów zgodnych. Są to: $\Phi_{max} = \{1, 2, 3, 5; 2, 4; 3, 6; 4, 6\}$. Stąd wynika, że należy rozwiązać problem przypisania stanu 2 (do pierwszej lub do drugiej grupy), stanu 3 (do pierwszej lub do trzeciej grupy), stanu 4 (do drugiej lub do czwartej grupy) i stanu 6 (do trzeciej lub do czwartej grupy). W tym celu utwórzmy hipotetyczną tablicę przejść dla grup stanów maksymalnie zgodnych pokazaną w tablicy 4.12a.

Zacznijmy od rozważenia do jakiej grupy przypisać stan 2. Z tablicy 4.12a można utworzyć dwie: jedną jeśli stan 2 będzie w grupie 1, 2, 3, 5 (tablica 4.12b) i drugą jeśli

a)

a b	S 00	01	11	10	Y 00	01	11	10
1	-	3	4	2	-	1	1	1
2	4	-	-	-	0	-	-	-
3	6	6	-	-	0	1	-	-
4	-	6	1	5	-	0	0	1
5	-	-	2	-	-	-	1	-
6	3	-	2	3	0	-	0	1

b)



Rysunek 4.13. Tablica przejść automatu i trójkątna tablica skracania

Tablica 4.12. Ilustracja rozważań przypisania stanu 2 do grupy

a)

S \ a b	00	01	11	10				
1,2,3,5	4,6	3,6	2,4	2	0	1	1	1
2,4	4	6	1	5	0	0	0	1
3,6	3,6	6	2	3	0	1	0	1
4,6	3	6	1,2	3,5	0	0	0	1

b)

S \ a b	00	01	11	10				
1,2,3,5	4,6	3,6	2,4	2	0	1	1	1
4	-	6	1	5	-	0	0	1
3,6	3,6	6	2	3	0	1	0	1
4,6	3	6	1,2	3,5	0	0	0	1

c)

S \ a b	00	01	11	10				
1,3,5	6	3,6	2,4	2	0	1	1	1
2,4	4	6	1	5	0	0	0	1
3,6	3,6	6	2	3	0	1	0	1
4,6	3	6	1,2	3,5	0	0	0	1

Tablica 4.13. Ilustracja rozważań przypisania stanu 3 do grupy

a)

S \ a b	00	01	11	10				
1,3,5	6	3,6	2,4	2	0	1	1	1
2,4	4	6	1	5	0	0	0	1
6	3	-	2	3	0	-	0	1

b)

S \ a b	00	01	11	10				
1,5	-	3	2,4	2	-	1	1	1
2,4	4	6	1	5	0	0	0	1
3,6	3,6	6	2	3	0	1	0	1

2 będzie w grupie 2, 4 (tablica 4.12c). Z tablic tych widać, że zgodność grupy stanów 1, 2, 3, 5 (a dokładnie stanów 1 i 5) jest warunkowana zgodnością stanów 2 i 4. Dlatego stan 2 należy połączyć ze stanem 4 i zostawić grupy 1, 3, 5 i 2, 4. Wtedy jednak z dalszych

rozważań należy wyeliminować grupę 4, 6, gdyż zgodność stanów 4 i 6 jest warunkowana zgodnością stanów 1 i 2, który to warunek nie jest teraz spełniony.

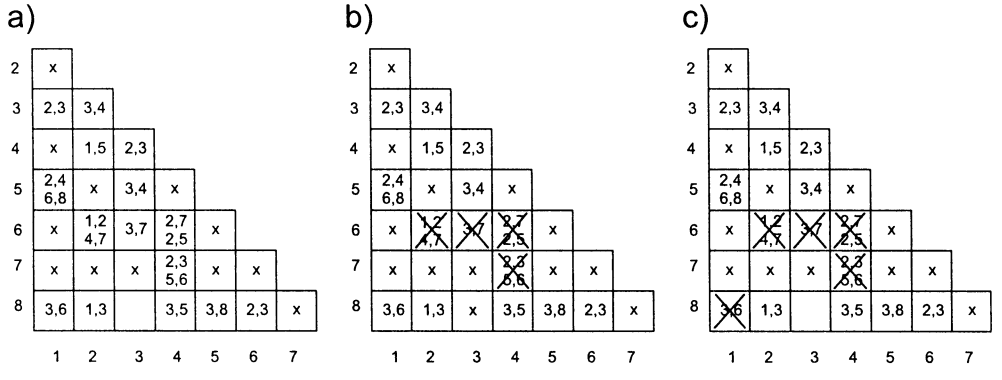
Rozważmy możliwość przypisania stanu 3 albo do grupy 1, 3, 5 albo do grupy 3, 6. Utwórzmy analogicznie jak poprzednio dwie tablice pokazane w tablicy 4.13, gdzie stan 3 przypisano do grupy 1, 3, 5 (tabl. 4.13a) i do grupy 3, 6 (tabl. 4.13b). Ponieważ zgodność grupy 1, 3, 5 (a dokładnie stanów 1 i 3) jest warunkowana zgodnością stanów 3 i 6, to należy pozostawić grupę 3, 6 i wówczas przyjąć jako rozwiązanie automat trzystanowy o stanach 1, 5; 2, 4 i 3, 6. ⊠

Przykład 4.4. Niech będzie dany automat Mealy'ego określony tablicą przejść i wyjść pokazaną w tablicy 4.14.

Tablica 4.14. Pierwotna tablica przejść i wyjść przykładowego automatu

s \ $a_i b_i$	0	1	0	1
1	2	6	0	0
2	4	1	1	0
3	3	-	-	0
4	2	5	1	-
5	4	8	0	-
6	7	2	1	0
7	3	6	1	1
8	-	3	-	0

Zbudujmy trójkątną tablicę skracania i dokonajmy wpisów do niej według zasady, że pary zgodnych stanów oznaczymy pustym miejscem, pary stanów zgodnych warunkowo oznaczymy poprzez wpisanie warunków, a pary stanów sprzecznych oznaczymy krzyżykiem. Po dokonaniu tych wpisów otrzymamy trójkątną tablicę skracania przedstawioną na rysunku 4.14a. Ponieważ w takiej tablicy mogą znaleźć się warunki, które nie są spełnione, gdyż pary stanów stanowiących warunek są parami sprzecznymi, to proces wykreślania par przeprowadza się iteracyjnie dotąd, aż w pewnym kroku nie będzie już par sprzecznych. Tablice trójkątne dla naszego przykładu w każdym kroku iteracji przedstawiono na rysunkach 4.14b i 4.14c. W przykładzie tym w drugim kroku wykreślono pary stanów 2, 6; 3, 6; 4, 6 i 4, 7 (rys.4.14b).



Rysunek 4.14. Trójkątne tabelice skracania w kolejnych krokach skracania

a)

1 | 1,3; 1,5;
2 | 2,3; 2,4; 2,8
3 | 3,4; 3,5; 3,8

b)

1,2,3 | 1,3,5; 2,3,4; 2,3,8

c)

1,2,3 | 1,3,5; 2,3,4; 2,3,8
4 | 4,8

Rysunek 4.15. Kolejne kroki procedury poszukiwania grup stanów maksymalnie zgodnych

W trzecim kroku należy wykreślić te pary stanów, które jako warunek swej zgodności mają wykreślone w poprzednim kroku stany. W naszym przykładzie jest jedna taka para 1, 8, która jako warunek zgodności miała parę stanów 3, 6. Ponieważ para stanów 1, 8 nie stanowi warunku zgodności żadnych innych par, to proces budowania trójkątnej tablicy skracania kończy się na tym kroku, a wynikowa trójkątna tablica skracania jest pokazana w tablicy 4.14c.

Procedura minimalizacji wymaga teraz znalezienia grup stanów maksymalnie zgodnych. W tym celu należy wyszukać pary tworzące „trójki" (trzy pary), „czwórki" (sześć par) itd. Algorytm poszukiwania maksymalnych grup stanów zgodnych rozpocząć można od dowolnego stanu. Zacznijmy od stanu 1. Tworzy on pary 1, 3 i 1, 5. Na rysunku 4.15a utworzono pierwszy wiersz. Biorąc kolejno stan 2 uzupełnia się listę o następne pary (druga kolumna trójkątnej tabeli skracania), które w tym przypadku są: 2, 3; 2, 4; i 2, 8 (drugi wiersz na rysunku 4.15a). Biorąc stan 3 (trzeci wiersz na rysunku 4.15a) uzupełnia się listę o pary z trzeciej kolumny trójkątnej tablicy skracania: 3, 4; 3, 5 i 3, 8. W tym momencie procedury należy zauważyć, że powstały trójki: 1, 3, 5; 2, 3, 4 i 2, 3, 8. Skutkuje to tym, że z tych trójek tworzy się jeden wiersz (rys. 4.15b), a ponieważ zawierają one wszystkie pary, to pary te można wykreślić.

Dalej dopisując następny stan 4 (rys. 4.15c) uzupełnia się listę o pary z czwartej kolumny trójkątnej tablicy skracania. W naszym przykładzie jest tylko jedna para: 4,8 (drugi wiersz z rys. 4.15c). Para ta powoduje, że można zapisać na liście „czwórkę" 2, 3, 4, 8 pokazaną na rysunku 4.16.

1,2,3,4 | 1,3,5; 2,3,4,8
5 | 5,8
6 | 6,8
7 | 7

Rysunek 4.16. Kolejne kroki procedury poszukiwania grup stanow maksymalnie zgodnych

Jako następny stan bierzemy stan 5 i dopisujemy jedynie parę 5, 8, a następnie biorąc stan 6 dopisujemy parę 6, 8. Jako ostatni stan bierzemy stan 7, który nie tworzy żadnej pary i dlatego on sam tworzy grupę stanów. W wyniku otrzymano grupy stanów pokazane na rysunku 4.16.

Otrzymany zbiór grup stanów maksymalnie zgodnych dla naszego przykładu to: {1, 3, 5; 2, 3, 4, 8; 5, 8; 6, 8;7}. W tym momencie stajemy przed problemem wyboru w jakiej grupie pozostawić stany powtarzające się, a mianowicie: 3, 5 i 8. Rozpatrzmy zatem do jakiej grupy zaliczyć stan 3. Na rysunku 4.17a pokazano tablicę, której wiersze odpowiadają każdej grupie maksymalnie zgodnej a kolumny sygnałom wejściowym. W tablicę wpisano warunki zgodności. Na rysunku 4.17b pokazano tak zmienioną tablicę, że stan 3 wyeliminowano z grupy 1, 3, 5, a na rysunku 4.17c z grupy 2, 3, 4, 8. Zmieniono także odpowiednio wpisane warunki zgodności.

a)

G \ X	0	1
1,3,5	2,3	6,8
	2,4	
	3,4	
2,3,4,8	2,3	1,3
	3,4	1,5
		3,5
5,8		3,8
6,8		2,3
7		

b)

G \ X	0	1
1, 5	2,4	6,8
2,3,4,8	2,3	1,3
	3,4	1,5
		3,5
5,8		3,8
6,8		2,3
7		

c)

G \ X	0	1
1,3,5	2,3	6,8
	2,4	
	3,4	
2,4,8		1,3
		1,5
		3,5
5,8		3,8
6,8		2,3
7		

Rysunek 4.17. Sprawdzanie warunków zgodności grup stanów maksymalnie zgodnych

Z tablicy pokazanej na rysunku 4.17b można zauważyć, że po usunięciu stanu 3 z grupy 1, 3, 5 nie będą spełnione warunki zgodności grupy 2, 3, 4, 8 z drugiej kolumny tablicy (dla $x = 1$). Oznacza to, że z grupy tej należy wyłączyć stan 8 (warunki zgodności stanów 2 i 8 oraz 4 i 8). Z tablicy pokazanej na rysunku 4.17c można zauważyć, że nie będzie spełniony warunek 2, 3 (zgodność stanów 1 i 3 oraz 6 i 8), warunek 3,4 (zgodność stanów 3 i 5) i warunek 3, 8 (zgodność stanów 5 i 8). Zatem usunięcie stanu 3 z grupy 2, 3, 4, 8 powoduje „rozbicie" grup 1, 3, 5; 5, 8 i 6, 8. Dlatego stan 3 pozostawia się w grupie 2, 3, 4, 8, choć usuwa się z niej stan 8.

a)

G \ X	0	1
1, 5		6,8
2,3,4,8		1,5
5,8		3,8
6,8		2,3
7		

b)

G \ X	0	1
1, 5		
2,3,4,8		1,5
5,8		3,8
6,8		2,3
7		

c)

G \ X	0	1
1, 5		6,8
2,3,4,8		1,5
5,8		
6,8		2,3
7		

Rysunek 4.18. Analiza przynależności stanu 5 do grupy stanów maksymalnie zgodnych

Na rysunku 4.18a pokazano grupy stanów pozostałe po pierwszym kroku. Teraz zostanie rozważona przynależność stanu 5 albo do grupy 1, 5, albo do grupy 5, 8. W tablicy z rysunku 4.18b widać, że usunięcie stanu 5 z grupy 1, 5 powoduje, że nie spełniony jest warunek zgodności grupy 2, 3, 4 (warunek zgodności stanów 2 i 4). Dlatego pozostawia się grupę 1, 5.

Tablica 4.15. Pierwotna tablica przejść i wyjść przykładowego automatu

S \ X	0	1	0	1
s_1	s_2	s_3	0	0
s_2	s_2	s_1	1	0
s_3	s_4	s_2	1	0
s_4	s_2	s_3	1	1

W ostatnim kroku analizuje się przynależność stanu 8. Ponieważ występuje on albo pojedynczo, albo w grupie ze stanem 6, to oczywiście wybiera się grupę. Po tych rozważaniach otrzymuje się 4 grupy stanów: 1, 5; 2, 3, 4; 6, 8; 7. Oznaczając je odpowiednio przez s_1, s_2, s_3 i s_4 otrzymuje się wynikową tablicę przejść i wyjść pokazaną w tablicy 4.15 i jest to jedyne rozwiązanie naszego przykładu. ⊠

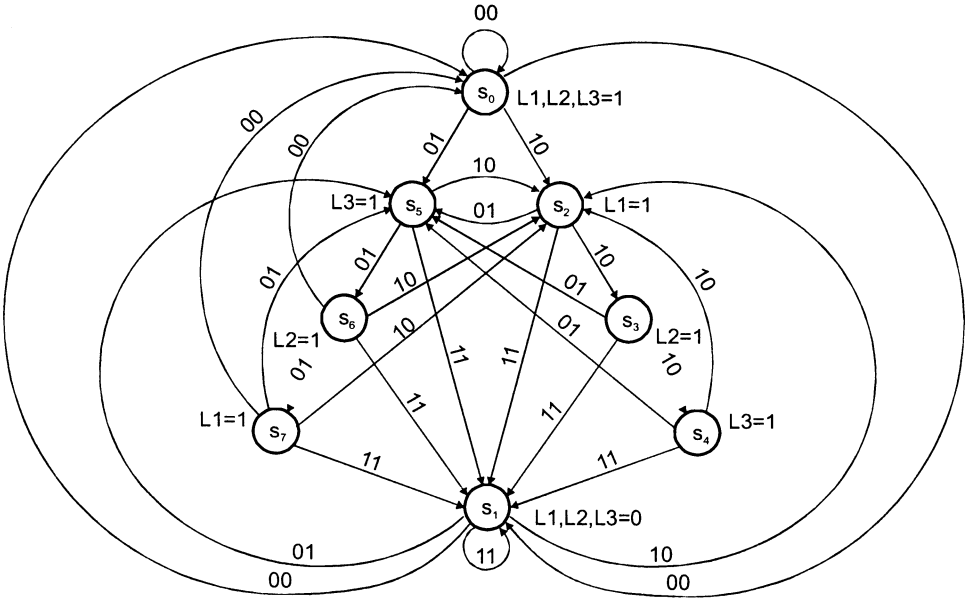
4.2.5. Przykładowe automaty synchroniczne

Przykład 4.5. Zaprojektować automat, który ma dwa sygnały wejściowe ustawiające pracę układu sterującego wyświetlaniem trzech lampek. Inicjalizacja pracy wyświetlania następuje, gdy na wejściach są sygnały 00. Wówczas palą się wszystkie lampki. Pojawienie się na wejściach sygnałów 10 powoduje, że lampki palą się przemiennie w kolejności *L1*, *L2* i *L3*. Pojawienie się na wejściach sygnałów 01 powoduje, że lampki palą się przemiennie w kolejności *L3*, *L2* i *L1*. Pojawienie się na wejściach sygnałów 11 powoduje zatrzymanie pracy układu i wszystkie lampki gasną.

Rozwiązanie. Analizę zadania tego typu wygodnie jest przeprowadzić za pomocą grafu. Na rysunku 4.19 pokazano graf przykładowego automatu. Założono automat Moore'a ze stanami:

s_0 — zapalone wszystkie lampki,
s_1 — zgaszone wszystkie lampki,
s_2, s_3, s_4 — zapalają się kolejno lampki *L1*, *L2* i *L3*,
s_5, s_6, s_7 — zapalają się kolejno lampki *L3*, *L2* i *L1*.

Na podstawie grafu można znaleźć tablicę przejść i wyjść automatu pokazaną w tablicy 4.16. Przyjmując sposób zakodowania korespondujący (prawie zgodny) z sygnałami wyjścio-



Rysunek 4.19. Graf automatu z przykładu 4.5

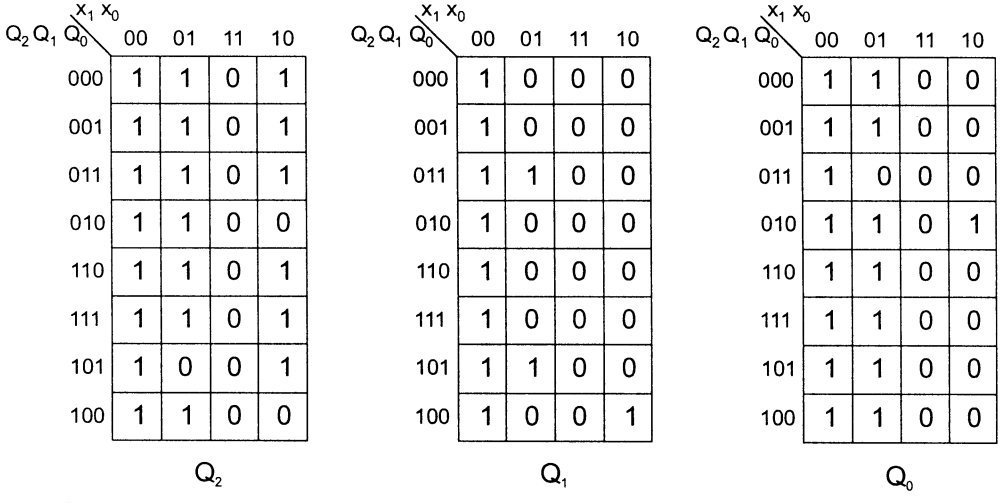
Tablica 4.16. Tablice przejść i wyjść automatu z przykładu 4.5

S \ $x_1 x_0$	00	01	11	10	L1	L2	L3
s_0	s_0	s_5	s_1	s_2	1	1	1
s_1	s_0	s_5	s_1	s_2	0	0	0
s_2	s_0	s_5	s_1	s_3	1	0	0
s_3	s_0	s_5	s_1	s_4	0	1	0
s_4	s_0	s_5	s_1	s_2	0	0	1
s_5	s_0	s_6	s_1	s_2	0	0	1
s_6	s_0	s_7	s_1	s_2	0	1	0
s_7	s_0	s_5	s_1	s_2	1	0	0

Tablica 4.17. Zakodowane tablice przejść i wyjść automatu z przykładu 4.5

$Q_2 Q_1 Q_0 \setminus x_1 x_0$	00	01	11	10	L1	L2	L3
111	111	101	000	100	1	1	1
000	111	101	000	100	0	0	0
100	111	101	000	010	1	0	0
010	111	101	000	001	0	1	0
001	111	101	000	100	0	0	1
101	111	011	000	100	0	0	1
011	111	110	000	100	0	1	0
110	111	101	000	100	1	0	0

$Q_2 Q_1 Q_0 \setminus x_1 x_0$	00	01	11	10	L1	L2	L3
000	111	101	000	100	0	0	0
001	111	101	000	100	0	0	1
011	111	110	000	100	0	1	0
010	111	101	000	001	0	1	0
110	111	101	000	100	1	0	0
111	111	101	000	100	1	1	1
101	111	011	000	100	0	0	1
100	111	101	000	010	1	0	0



Rysunek 4.20. Mapy Karnaugha funkcji wzbudzeń przerzutników typu *D* automatu z przykładu 4.5

wymi tak, aby maksymalnie uprościć realizację funkcji wyjść, otrzymamy zakodowaną tablicę przejść i wyjść pokazaną w tablicy 4.17a. W tablicy 4.17b pokazano uporządkowaną według kodu Graya tablicę przejść i wyjść, a na rysunku 4.20 przedstawiono mapy Karnaugha wzbudzeń trzech przerzutników. Na podstawie tych map można wyznaczyć funkcje wzbudzeń:

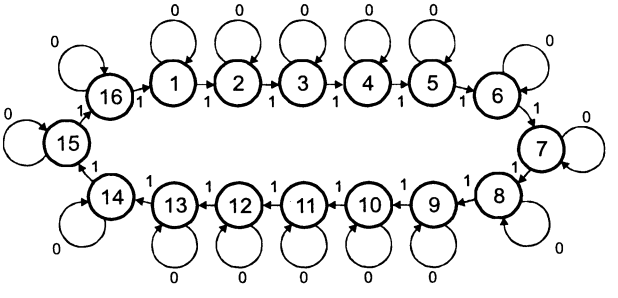
$$D_0 = \overline{x_1}\,\overline{x_0} + \overline{x_1}Q_2 + \overline{x_1}\,\overline{Q_1} + \overline{x_1}\,\overline{Q_0}Q_1 + \overline{x_0}\,\overline{Q_0}Q_1\overline{Q_2}$$
$$D_1 = \overline{x_1}\,\overline{x_0} + \overline{x_1}\,\overline{Q_2}Q_1Q_0 + \overline{x_1}Q_2\overline{Q_1}Q_0 + \overline{x_0}Q_2\overline{Q_1}\,\overline{Q_0}$$
$$D_2 = (\overline{x_1} + \overline{x_0})(Q_2 + \overline{Q_1} + Q_0 + \overline{x_1})(\overline{Q_2} + Q_1 + \overline{Q_0} + \overline{x_0})(\overline{Q_2} + Q_1 + Q_0 + \overline{x_1})$$

Z tablicy wyjść można wyznaczyć funkcje wyjść:

$$L_1 = Q_2Q_1 + Q_2\overline{Q_0}$$
$$L_2 = \overline{Q_2}Q_1 + Q_1Q_0$$
$$L_3 = Q_2Q_0 + \overline{Q_1}Q_0 \quad \boxtimes$$

Przykład 4.6. Zaprojektować automat zmieniający cyklicznie 2^n stanów w czasie, gdy na jego jedynym wejściu jest 1. Gdy na jego wejściu jest 0, to automat pozostaje w stanie, w którym był.

Rozwiązanie. Projektowany automat będzie składał się z n przerzutników, co zapewnia żądaną liczbę stanów. Cykliczność oznacza, że graf tego automatu tworzy okrąg składający się z 2^n węzłów. Taki automat nazywa się **licznikiem**. W tym przypadku jest to licznik impulsów taktujących (zegarowych). Licznik liczy modulo 2^n w czasie, gdy na jego wejściu jest 1. Zatem jest to wejście typu E (ang. *enable*).



Rysunek 4.21. Graf automatu z przykładu 4.6

Załóżmy rozwiązanie, dla którego $2^n = 16$. Rozwiązanie takie można zrealizować za pomocą 4 przerzutników. Graf takiego automatu pokazano na rysunku 4.21. Tablice przejść są pokazane w tablicy 4.18: pierwotna — 4.18a i zakodowana 4.18b. Funkcję wyjść takiego automatu można zaprojektować różnie, ale często przyjmuje się, że są to po prostu wyjścia przerzutników. Załóżmy rozwiązanie na przerzutnikach typu T. Mapy Karnaugha funkcji wzbudzeń czterech przerzutników pokazano na rysunku 4.22, a na rysunku 4.23 cały układ.

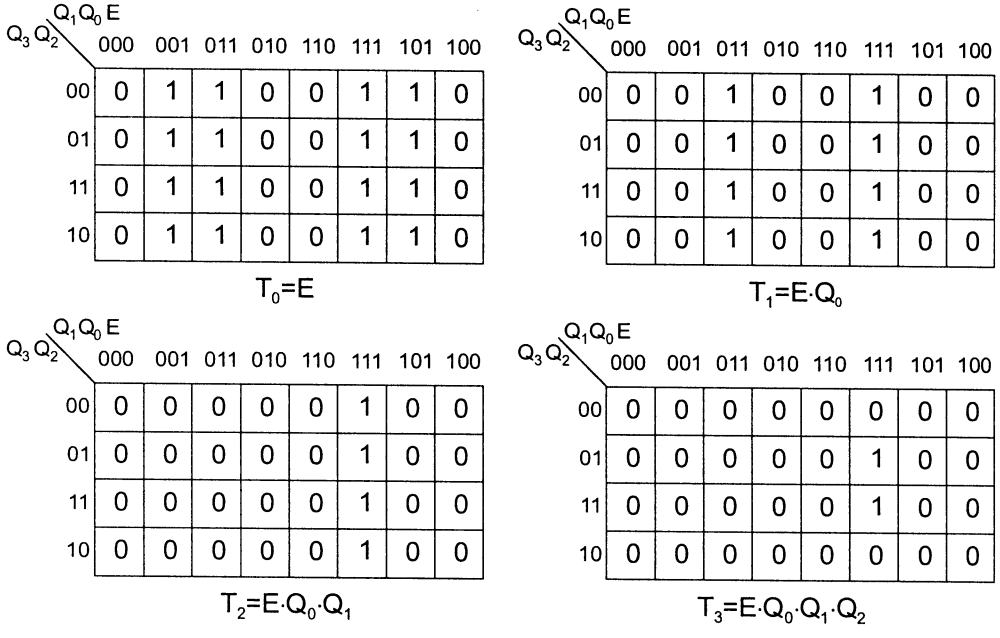
Tablica 4.18. Pierwotna i zakodowana tablica przejść licznika 16-stanowego

a)

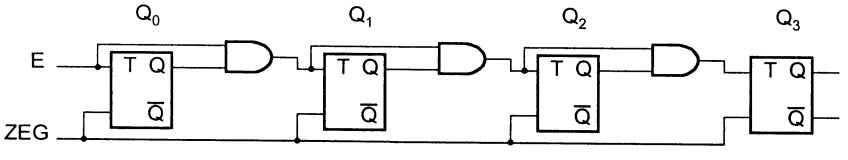
S \ E	0	1
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11
11	11	12
12	12	13
13	13	14
14	14	15
15	15	16
16	16	1

b)

$Q_3Q_2Q_1Q_0 \setminus E$	0	1
0000	0000	0001
0001	0001	0010
0010	0010	0011
0011	0011	0100
0100	0100	0101
0101	0101	0110
0110	0110	0111
0111	0111	1000
1000	1000	1001
1001	1001	1010
1010	1010	1011
1011	1011	1100
1100	1100	1101
1101	1101	1110
1110	1110	1111
1111	1111	0000



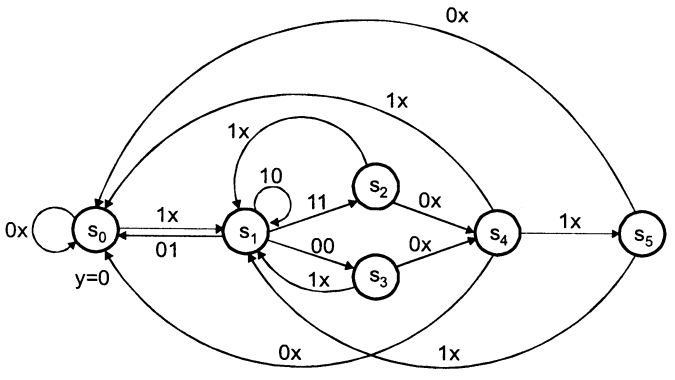
Rysunek 4.22. Mapy Karnaugha funkcji wzbudzeń przerzeutników typu *T* licznika z przykładu 4.6



Rysunek 4.23. Licznik 16-stanowy zrealizowany ma przerzutnikach typu *T* ⊠

Przykład 4.7. Zaprojektować automat wykrywający na jego wejściu D sekwencję stanów $1c01$, gdzie c jest wartością sygnału na wejściu c. Wykrycie sekwencji jest sygnalizowane na wyjściu automatu sygnałem 1.

Rozwiązanie. Zadanie nie precyzuje w pełni zachowania automatu we wszystkich możliwych przypadkach. Projektant musi w takim przypadku dokonać odpowiednich dodatkowych założeń. W przykładowym zadaniu chodzi o zmiany sygnału na wejściu c. Zakładamy, że jeśli nastąpi zmiana tego sygnału po wykryciu pierwszej jedynki, to zostanie ona zignorowana. Oznacza to, że sygnał c powinien być stały przez czas 4 taktów zegarowych począwszy od pierwszej jedynki.



Rysunek 4.24. Graf automatu z przykładu 4.7

Graf automatu pokazany jest na rysunku 4.24 i na jego podstawie sporządzono tablice przejść i wyjść automatu Moore'a pokazane w tablicy 4.19a. Minimalizując automat otrzymamy tablice przejść i wyjść pokazane w tablicy 4.19b (stany s_2 i s_3 połączono) oraz zakodowane (przypadkowo) tablice w tablicy 4.19c. W wyniku takiego zakodowania

Tablica 4.19. Tablica przejść i wyjść automatu z przykładu 4.7: a) pierwotna, b) zminimalizowana, c) zakodowana

a)

S \ Dc	00	01	11	10	y
s_0	s_0	s_0	s_1	s_1	0
s_1	s_3	s_0	s_2	s_1	0
s_2	s_4	s_4	s_1	s_1	0
s_3	s_4	s_4	s_1	s_1	0
s_4	s_0	s_0	s_5	s_5	0
s_5	s_0	s_0	s_1	s_1	1

b)

S \ Dc	00	01	11	10	y
s_0	s_0	s_0	s_1	s_1	0
s_1	$s_{2,3}$	s_0	$s_{2,3}$	s_1	0
$s_{2,3}$	s_4	s_4	s_1	s_1	0
s_4	s_0	s_0	s_5	s_5	0
s_5	s_0	s_0	s_1	s_1	1

c)

S \ Dc	00	01	11	10	y
000	000	000	001	001	0
001	011	000	011	001	0
011	010	010	001	001	0
010	000	000	110	110	0
110	000	000	001	001	1

a)

$Q_2Q_1Q_0$ \ Dc	00	01	11	10
000	0	0	1	1
001	1	0	1	1
011	0	0	1	1
010	0	0	0	0
110	0	0	1	1
111	-	-	-	-
101	-	-	-	-
100	-	-	-	-

D_0

$D_0 = \overline{Q}_1D + Q_0D + Q_2D + \overline{Q}_1Q_0\overline{c}$

b)

$Q_2Q_1Q_0$ \ Dc	00	01	11	10
000	0	0	0	0
001	1	0	1	0
011	1	1	0	0
010	0	0	1	1
110	0	0	0	0
111	-	-	-	-
101	-	-	-	-
100	-	-	-	-

D_1

$D_1 = Q_0\overline{Dc} + Q_1Q_0\overline{D} + \overline{Q}_2Q_1\overline{Q}_0D + \overline{Q}_1Q_0Dc$

c)

$Q_2Q_1Q_0$ \ Dc	00	01	11	10
000	0	0	0	0
001	0	0	0	0
011	0	0	0	0
010	0	0	1	1
110	0	0	0	0
111	-	-	-	-
101	-	-	-	-
100	-	-	-	-

D_2

$D_2 = \overline{Q}_2Q_1\overline{Q}_0D$

Rysunek 4.25. Mapa Karnaugha funkcji wzbudzeń przerzutników Q_2 typu D automatu z przykładu 4.7 a) przerzutnika Q_0, b) przerzutnika Q_1, c) przerzutnika Q_2

otrzymano następujące funkcje wzbudzeń przerzutników typu D z map Karnaugha przedstawionych na rysunku 4.25:

$$D_0 = \overline{Q_1}D + Q_0D + Q_2D + Q_0\overline{Q_1}\,\overline{c}$$

$$D_1 = Q_0\overline{D}\,\overline{c} + Q_0Q_1\overline{D} + \overline{Q_0}Q_1\overline{Q_2}D + Q_0\overline{Q_1}Dc$$

$$D_2 = \overline{Q_0}Q_1\overline{Q_2}D$$

⊠

4.3. Automaty asynchroniczne

4.3.1. Wstęp

Układy asynchroniczne stosowane są głównie w urządzeniach automatyki, gdzie zwykle sygnały wejściowe pojawiają się w chwilach przypadkowych. Ponieważ automat musi zareagować na dane zdarzenie (zmiana stanu sygnału wejściowego), to sygnały wyjściowe muszą zmieniać się w momencie zmiany sygnału wejściowego. W technice komputerowej sygnały wejściowe automatu zmieniają się najczęściej w takt impulsów synchronizujących i dlatego stosuje się tu zwykle układy synchroniczne. Są one łatwiejsze w projektowaniu. Natomiast projektowanie automatów asynchronicznych wymaga od projektanta znacznej wprawy i doświadczenia. W niniejszym rozdziale przedstawiono zjawiska występujące w układach asynchronicznych i pokazano metody projektowania automatów uwzględniające te zjawiska.

4.3.2. Tablice przejść i wyjść automatów asynchronicznych

Automaty asynchroniczne mogą być, podobnie jak automaty synchroniczne, automatami typu Moore'a i automatami typu Mealy'ego. Interpretacja zmiany stanów jest jednak inna. W przypadku automatów synchronicznych zmiana stanu następowała w momencie poja-

Tablica 4.20. Pierwotna tablica przejść automatu asynchronicznego

S \ $x_1 x_0$	00	01	11	10	y
s_1	s_2	s_2	s_4	s_1	1
s_2	s_3	s_2	s_2	s_2	0
s_3	s_4	s_4	s_2	s_3	1
s_4	s_1	s_4	s_3	s_4	0

wienia się impulsu synchronizującego i automat pozostawał w nowym stanie aż do momentu przyjścia następnego impulsu synchronizującego. W automatach asynchronicznych zmiany stanu następują w momentach zmiany stanu sygnałów wejściowych. Rozważmy tablicę przejść i wyjść automatu pokazaną w tablicy 4.20.

Działanie automatu można analizować sprawdzając jego zachowanie się dla różnych sygnałów wejściowych. Zacznijmy od prawej kolumny, tj. wzbudzenia 00. Można zauważyć, że dla każdego stanu automatu wzbudzenie 00 powoduje zmianę stanu. Jeśli automat jest w stanie s_1, to przejdzie do stanu s_2. Jeśli jest w stanie s_2, to przejdzie do stanu s_3. Jeśli jest w stanie s_3, to przejdzie do stanu s_4 i wreszcie jeśli jest w stanie s_4, to przejdzie do stanu s_1. W takim przypadku automat „nie zatrzymuje się" w żadnym ze stanów. Mówimy, że w kolumnie 00 nie istnieje **stan stabilny**. Obserwując tablicę wyjść dochodzi się do wniosku, że dla wzbudzenia 00 automat jest generatorem naprzemiennego ciągu zer i jedynek.

Analizując drugą kolumnę tablicy przejść, tj. dla wzbudzenia 01, można zaobserwować, że automat znajdzie się albo w stanie s_2 albo w stanie s_4. Jeśli był w stanie s_1, to przejdzie do stanu s_2 i w nim pozostanie, a jeśli był w stanie s_3, to przejdzie do stanu s_4 i w nim pozostanie. Stany s_2 i s_4 nazywać będziemy stanami stabilnymi.

W trzeciej kolumnie tablicy przejść bez względu na stan w jakim automat znajdował się początkowo, po przyjściu wzbudzenia 11 automat zawsze przejdzie do stanu s_2. Jeśli był w stanie s_1 lub s_4, to do stanu s_2 przejdzie pośrednio. Ze stanu s_1 najpierw przejdzie do stanu s_4, a następnie ze stanu s_4 przejdzie do stanu s_3 i dopiero ze stanu s_3 przejdzie do stanu s_2. Natomiast jeśli był w stanie s_4, to przejdzie do stanu s_3 i ze stanu s_3 przejdzie do stanu s_2. Natomiast ze stanu s_3 automat przejdzie bezpośrednio do stanu s_2. Mówimy wówczas, że w tej kolumnie jest jeden stan stabilny. Wreszcie w czwartej kolumnie są cztery stany stabilne. Bez względu na stan automatu wzbudzenie 10 pozostawia automat w stanie w jakim był przed przyjściem tego wzbudzenia.

Z opisu widać, że zachowanie automatu jest określone przez jego tablicę przejść, ale w przeciwieństwie do automatów synchronicznych tablica może określać sensowne lub bezsensowne jego zachowania. W przykładowym automacie wzbudzenie 00 powoduje generację ciągu zer i jedynek, a wzbudzenie 10 nie zmienia jego stanu. Oba zachowania z punktu widzenia działania automatu sekwencyjnego są „niewytłumaczalne".

4.3.3. Niezawodność działania automatów asynchronicznych

Jednym z problemów występującym podczas projektowania jest zjawisko tzw. **wyścigów** (ang. *race*). Załóżymy, że automat asynchroniczny będzie realizowany na przerzutnikach i jego stany będą kodowane podobnie jak w automatach synchronicznych.

Rozpatrzmy tablicę przejść automatu pokazaną w tablicy 4.21. Automat opisany tablicą 4.20 ma 4 stany, które można zakodować na wiele sposobów. Dla ustalenia uwagi przypisano poszczególnym stanom następujące dwubitowe ciągi: s_1 — 00, s_2 — 01, s_3 — 10 i s_4 — 11.

Przykładową tablicę przejść można zrealizować na dwóch przerzutnikach, jednak ich sposób działania jest asynchroniczny, więc inny niż znany nam do tej pory. Przerzutniki te

Tablica 4.21. Zakodowana tablica przejść automatu określonego

S \ $x_1 x_0$	00	01	11	10	y
00	01	01	11	00	1
01	10	01	01	01	0
10	11	11	01	10	1
11	00	11	10	11	0

przedstawiono dalej, natomiast tutaj zwrócimy uwagę na fakt ich niejednorodności. Zastosowane przerzutniki mogą mieć różną szybkość działania. Projektant automatu nie wie, który z przerzutników zmienia swój stan szybciej. Zadaniem projektanta jest tak zaprojektować automat, aby nie występowało zjawisko wyścigu, tj. aby automat działał prawidłowo bez względu na szybkość zastosowanych przerzutników.

Rozpatrzmy zachowanie automatu na podstawie analizy jego działania w każdej kolumnie tablicy przejść. Najpierw będziemy zakładać, że szybciej zmienia swój stan pierwszy przerzutnik, a potem, że szybciej zmienia swój stan drugi przerzutnik. Założenia te mają znaczenie w tych przypadkach, kiedy zmieniają swoje stany oba przerzutniki. Jeśli jest więcej przerzutników, to rozpatruje się te miejsca w tablicy przejść, gdzie zmienia swój stan więcej niż jeden przerzutnik.

W pierwszej kolumnie zmieniają swoje stany oba przerzutniki w drugim i czwartym wierszu. Jeśli szybciej zmienia swój stan pierwszy przerzutnik, to wychodząc ze stanu 01 (drugi wiersz) automat przejdzie do stanu 11 (a nie 10). W czwartym wierszu wychodząc ze stanu 11 automat przechodzi do stanu 01 (a nie 00). W kolumnie tej następowała cykliczna zmiana stanów (generacja), co oznacza, w świetle podanego wyjaśnienia, że zmienia swój stan tylko pierwszy przerzutnik. Zmiany następują w pętli $11 \rightarrow 01 \rightarrow 11$. Jeśli szybciej zmienia swój stan drugi przerzutnik, to wychodząc ze stanu 01 (drugi wiersz) automat przejdzie do stanu 00 (a nie 10). W czwartym wierszu wychodząc ze stanu 11 automat przejdzie do stanu 10 (a nie 00). W tym przypadku generacja następować będzie przez zmianę drugiego przerzutnika w pętli $00 \rightarrow 00 \rightarrow 01$ lub w pętli $11 \rightarrow 10 \rightarrow 11$.

W drugiej kolumnie nie ma przypadku, aby zmieniały się dwa przerzutniki, a więc nie występuje zjawisko wyścigu. Podobnie jest w czwartej kolumnie. Natomiast w trzeciej kolumnie dwa przerzutniki zmieniają swoje stany w pierwszym i trzecim wierszu tablicy przejść. Jeśli szybciej zmienia swój stan pierwszy przerzutnik, to wychodząc ze stanu 00 (pierwszy wiersz) automat przejdzie do stanu 10 (a nie 11). A ze stanu 10 (czwarty wiersz) automat przejdzie do stanu 00 (a nie do stanu 01). Nastąpi zatem generacja $00 \rightarrow 10 \rightarrow 00$. Jeśli szybciej zmienia swój stan drugi przerzutnik, to wychodząc ze stanu 00 (pierwszy wiersz) automat przejdzie do stanu 10 (a nie 11). Z tablicy przejść wynika, że stan 01 jest stanem stabilnym i automat w nim pozostanie. W czwartym wierszu wychodząc ze stanu 11 automat przechodzi do stanu 10. Wychodząc ze stanu 10 (trzeci wiersz) automat przejdzie do stanu 11 (a nie 01). Ze stanu 11 automat przejdzie do stanu 10 i powstanie generacja $10 \rightarrow 11 \rightarrow 10$.

Z przedstawionego, w danym przykładzie zjawiska, widać, że w zależności od założonej szybkości działania przerzutników jest różne działanie danego automatu. Ponieważ, jak powiedziano wcześniej, przykładowa tablica przejść nie jest sensowna, to rozpatrzmy inny przykład. Niech będzie dana tablica przejść pokazana w tablicy 4.22.

W tablicy 4.22 kółkami zaznaczono stany stabilne. Można zauważyć, że stany stabilne znajdują się w każdym wierszu i w każdej kolumnie. Jest to przesłanka do stwierdzenia, że automat jest sensowny. W pierwszej kolumnie zjawisko wyścigu można zauważyć w trzecim i czwartym wierszu. Jeśli szybciej zmienia swój stan pierwszy przerzutnik, to wychodząc ze

Tablica 4.22. Przykładowa tablica przejść automatu asynchronicznego

S \ X_1X_0	00	01	11	10
00	01	01	11	(00)
01	(01)	10	(01)	10
10	01	11	(10)	00
11	00	(11)	10	10

stanu 10 (trzeci wiersz) automat przejdzie do stanu 00, a ze stanu 00 przejdzie do stanu 01 i w tym stanie pozostanie, gdyż jest to stan stabilny. W czwartym wierszu wychodząc ze stanu 11 automat przejdzie do stanu 01 i w tym stanie pozostanie. Działanie automatu jest prawidłowe, gdyż zawsze dojdzie do stanu stabilnego 01. Jeśli szybciej zmienia swój stan drugi przerzutnik, to wychodząc ze stanu 10 (trzeci wiersz) automat przejdzie do stanu 11, następnie powróci do stanu 10. Nastąpi generacja, czyli przełączanie się drugiego przerzutnika. W czwartym wierszu wychodząc ze stanu 11 automat przejdzie do stanu 10 i dalej będzie następowało zjawisko generacji drugiego przerzutnika. Zatem w pierwszej kolumnie automat działa nieprawidłowo, gdyż w przypadku gdy pierwszy przerzutnik jest szybszy, następuje zjawisko generacji.

W drugiej kolumnie wyścig występuje w drugim wierszu. Ze stanu 01 automat przechodzi do stanu 11 i w nim pozostaje, jeśli szybciej zmienia się pierwszy przerzutnik lub przechodzi do stanu 00 i dalej ponownie do stanu 01, jeśli szybciej zmienia się drugi przerzutnik. W tym drugim przypadku nastąpi generacja zmian drugiego przerzutnika.

W trzeciej kolumnie wyścig występuje w pierwszym wierszu. Ze stanu 00 automat przejdzie do stanu 10 i w nim pozostanie, jeśli szybciej zmienia swój stan pierwszy przerzutnik. Jeśli natomiast szybciej zmienia swój stan drugi przerzutnik, to automat przejdzie do stanu 01 i w nim pozostanie. W obu przypadkach automat przechodzi do różnych stanów i mówimy, że w tej kolumnie jest **wyścig krytyczny**.

W czwartej kolumnie wyścig występuje w drugim wierszu. Ze stanu 01 automat przechodzi do stanu 00 i w nim pozostaje, jeśli szybciej zmienia swój stan drugi przerzutnik. Jeśli natomiast szybciej zmienia swój stan pierwszy przerzutnik, to ze stanu 01 automat przejdzie do stanu 11, a następnie do stanu 10 i dopiero potem do stanu 00. Zatem w obu przypadkach automat dochodzi do tego samego stanu. Mówimy wówczas, że wystąpił **wyścig niekrytyczny**.

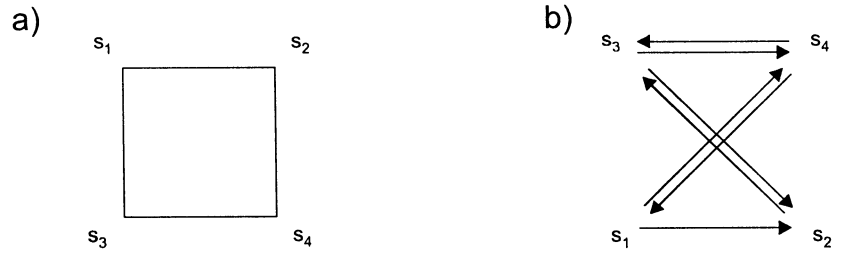
Automat nie działa prawidłowo, jeśli po zakodowaniu jego stanów wystąpi wyścig krytyczny lub generacja. Zjawiska te należy wyeliminować. Eliminacji dokonuje się poszukując takiego sposobu zakodowania stanów aby nie występowały zjawiska szkodliwe. Może się okazać, że nie da się znaleźć takiego sposobu zakodowania i wtedy trzeba zwiększyć liczbę bitów w ciągu kodującym stany. Jeśli stany automatu dało się zakodować najkrótszym k-bitowym ciągiem i nie można uniknąć zmian stanów takich, aby zmieniał się tylko jeden bit, to należy wydłużyć ciąg kodujący do wartości $k+1$ w celu zwiększenia możliwej liczby stanów. Jeśli zmiana stanu automatu ze stanu x do stanu y powoduje zmianę więcej niż jednego przerzutnika, to stan y można zastąpić innym stanem różniącym się od

Tablica 4.23. Pierwotna tablica przejść automatu z rysunku 4.22

S \ X_1X_0	00	01	11	10
S_1	S_2	S_2	S_4	S_1
S_2	S_2	S_3	S_2	S_3
S_3	S_2	S_4	S_3	S_1
S_4	S_1	S_4	S_3	S_3

stanu x tylko na jednej pozycji, i dopiero z tego stanu przejść do stanu y. Zatem nie zmienia się działanie automatu, ale zmienia się tablica przejść. Niektóre zmiany stanów realizuje się poprzez dodatkowo wprowadzone stany. Dodatkowe stany umożliwią przeprowadzenie automatu ze stanu x do stanu y taką drogą, aby nie wystąpił wyścig. Rozważania te zilustrowano przykładem tablicy przejść automatu z tablicy 4.22. Tablica ta odpowiada niezakodowanej tablicy przejść pokazanej w tablicy 4.23.

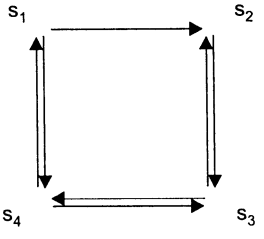
Automat określony tablicą 4.23 można zakodować różnymi ciągami dwubitowymi. Jedno z tych zakodowań daje tablicę 4.22, w której występują zjawiska szkodliwe. Spróbujmy znaleźć inny dwubitowy ciąg, który wyeliminuje te zjawiska. W tym celu wykonuje się graf przejść na sześcianie (ang. *cube*) 2^m-wymiarowym, gdzie sąsiednie wierzchołki sześcianu różnią się tylko jednym bitem (kod Graya). Liczba stanów automatu n ma spełniać nierówność $2^{m-1} < n \leq 2^m$. W naszym przykładzie $n = 4$, więc sześcian będzie 2-wymiarowy (kwadrat), co pokazano na rysunku 4.26a. Graf przejść pokazano na rysunku 4.26b.



Rysunek 4.26. Ilustracja analizy zmian stanów automatu: a) sześcian, b) graf przejść automatu

Z grafu z rysunku 4.26b można zauważyć, że tablica przejść nie zawiera zmian pomiędzy stanami s_1 i s_3. Jeśli zatem dokonamy zamiany miejscami stanów s_3 i s_4 na sześcianie, to strzałki pojawią się tylko pomiędzy sąsiednimi narożnikami. Biorąc kolejne słowa kodu Graya dla kolejnych stanów z narożników sześcianu otrzymamy zakodowanie stanów automatu, które nie powoduje wyścigów (rys. 4.27).

Na podstawie rysunku 4.27 można na kilka różnych sposobów zakodować stany przykładowego automatu tak, aby nie było wyścigu. Biorąc $s_1 = 00$, $s_2 = 01$, $s_3 = 11$ i $s_4 = 10$ otrzymamy jeden sposób zakodowania. Są jeszcze trzy inne sposoby zakodowania: dla $s_1 = 01$, dla $s_1 = 11$ i dla $s_1 = 10$. Można także odstąpić od kodu Graya i przypisać odpowiednim stanom słowa kodowe biorąc: $s_1 = 00$, $s_2 = 10$, $s_3 = 11$ i $s_4 = 01$. Analogicznie można przyjąć trzy inne sposoby: dla $s_1 = 01$, dla $s_1 = 11$ i dla $s_1 = 10$.

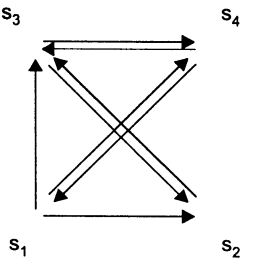


Rysunek 4.27. Przypisanie stanów automatu z rysunku 4.26 narożnikom sześcianu

a)

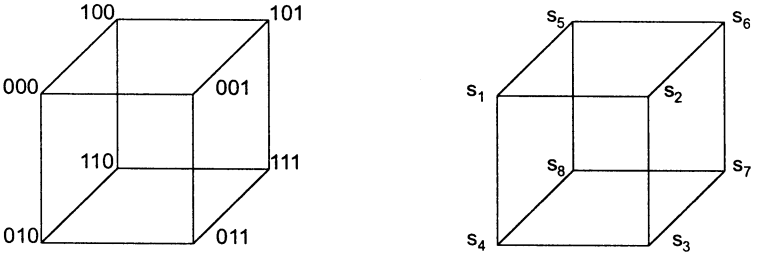
S \ $x_1 x_0$	00	01	11	10
s_1	s_2	s_3	s_4	s_1
s_2	s_2	s_3	s_2	s_3
s_3	s_2	s_4	s_3	s_4
s_4	s_1	s_4	s_3	s_3

b)



Rysunek 4.28. Tablica przejść i odpowiadający jej graf

W przedstawionym przypadku udało się wyeliminować zjawisko wyścigu poprzez zmianę zakodowania. Aby pokazać, że czasami jest to niemożliwe zmodyfikujmy tablicę przejść tak, aby dodać przejście pomiędzy stanami s_1 i s_3. Taką tablicę przejść i odpowiadający jej graf zmian przedstawiono na rysunku 4.28. Z analizy grafu przejść pokazanego na rysunku 4.28b wynika, że nie da się tak przestawić stanów, aby zakodować je zgodnie z kodem Graya i aby nie wystąpiło zjawisko wyścigu. Dlatego należy zwiększyć długość ciągu kodującego do 3 bitów. Wtedy jest 8 możliwych stanów i każdemu stanowi można przypisać jeden z ośmiu narożników sześcianu i jedno z ośmiu słów kodowych. Przypisanie tych słów powinno spełniać warunek eliminacji wyścigu, tj. jednoznacznego przejścia z danego stanu do stanu wskazywanego przez tablicę przejść. Na rysunku 4.29 przedstawiono jedno z możliwych przypisań stanów narożnikom sześcianu.



Rysunek 4.29. Sześcian dla trzech zmiennych i kodowanie stanów przykładowego automatu

W automacie opisanym tablicą z rysunku 4.28 i po zakodowaniu jak pokazano na rysunku 4.29 występuje wyścig w drugiej kolumnie przy przejściu ze stanu s_1 do stanu s_3. Przejście to, mając do dyspozycji 8 stanów, można zrealizować pośrednio przechodząc po narożnikach sześcianu. Na przykład można ze stanu s_1 przejść do stanu s_5, następnie do stanu s_6, dalej do stanu s_7 i dopiero wtedy do stanu s_3. W tablicy 4.24 pokazano ośmiostanową tablicę przejść realizującą wymienione przejścia.

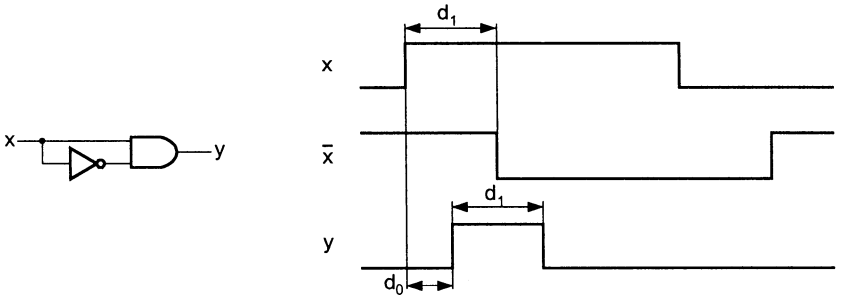
Oczywiście pokazane przejścia pośrednie nie są jedynymi, które eliminują wyścig. W podanym przykładzie innym przejściem ze stanu s_1 do stanu s_3 może być sekwencja: $s_1 \rightarrow s_5 \rightarrow s_8 \rightarrow s_7 \rightarrow s_3$.

Drugim zjawiskiem, które może ujemnie wpłynąć na działanie automatu asynchronicznego jest zjawisko **hazardu** (ang. *risk, hazard*). Zjawisko to powstaje w układach kombinacyjnych, w których ten sam sygnał przechodzi różnymi drogami, a więc z różnymi

Tablica 4.24. Ośmiostanowa tablica przejść z wyeliminowanym wyścigiem

S \ x_1x_0	00	01	11	10
s_1	s_2	s_5	s_4	s_1
s_2	s_2	s_3	s_2	s_3
s_3	s_2	s_4	s_3	s_4
s_4	s_1	s_4	s_3	s_3
s_5	-	s_6	-	-
s_6	-	s_7	-	-
s_7	-	s_3	-	-
s_8	-	-	-	-

opóźnieniami. Istnieje wtedy możliwość powstania krótkotrwałego impulsu sprzecznego z prawami algebry Boole'a. Na przykład może powstać jedynka logiczna na wyjściu iloczynu zmiennej x i jej negacji. Zjawisko to przedstawiono na rysunku 4.30. Czas trwania impulsu



Rysunek 4.30. Ilustracja zjawiska hazardu

jest równy opóźnieniu wnoszonemu przez bramkę inwertera, a czas opóźnienia narastającego zbocza impulsu względem narastającego zbocza sygnału x jest równy opóźnieniu wnoszonemu przez bramkę iloczynu. W układach kombinacyjnych hazard może nie wywoływać szkodliwych zjawisk, ale w układach sekwencyjnych szkodliwy impuls może zostać podtrzymany i w wyniku tego układ może działać wadliwie. Podczas projektowania automatów asynchronicznych trzeba uwzględniać to zjawisko, co pokazano w przykładach.

4.3.4. Struktury automatów asynchronicznych

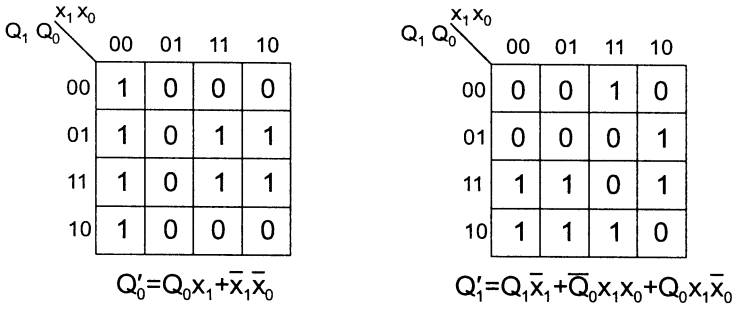
Możliwe są dwie struktury automatu: jedna wykorzystująca wyłącznie bramki (układ kombinacyjny ze sprzężeniem zwrotnym) i druga wykorzystująca bramki i asynchroniczne przerzutniki elementarne typu RS lub $\overline{RS}$.

Rozważmy najpierw rozwiązanie pierwsze na przykładzie automatu zadanego tablicą przejść i wyjść jak w tablicy 4.25. Pierwotną tablicę przejść zakodowano w taki sposób, aby

Tablica 4.25. Pierwotna i zakodowana tablica przejść automatu

S \ x_1x_0	00	01	11	10
s_1	s_2	s_1	s_4	s_1
s_2	s_2	s_1	s_2	s_3
s_3	s_3	s_4	s_2	s_3
s_4	s_3	s_4	s_4	s_1

Q_1Q_0 \ x_1x_0	00	01	11	10
00	01	00	10	00
01	01	00	01	11
11	11	10	01	11
10	11	10	10	00



Rysunek 4.31. Mapy Karnaugha funkcji Q_0' i Q_1'

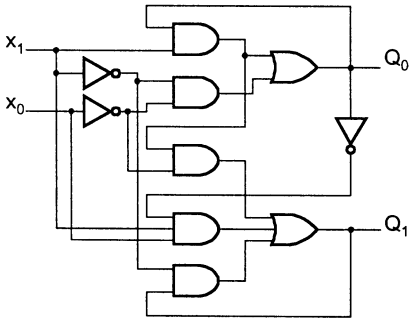
nie powstało zjawisko wyścigu. Ponieważ zakodowana tablica przejść jest czterostanowa, to dało się każdy stan zakodować słowem dwubitowym. Jeśli mniej znaczący bit (bit po prawej stronie słowa) oznaczymy przez Q_0 (podobnie jak to czyniliśmy dla przerzutników) a bit bardziej znaczący przez Q_1, to można utworzyć dwie funkcje $Q_0' = f_0(x_0, x_1, Q_0, Q_1)$ i $Q_1' = f_1(x_0, x_1, Q_0, Q_1)$. Z funkcji tych wynika, że w układzie realizującym te funkcje istnieje sprzężenie zwrotne, gdyż zmienne Q_0 i Q_1 zależą od samych siebie. Oznacza to, że w układzie wykorzystuje się zjawisko opóźnienia i Q_1' oraz Q_0' przyjmują takie same wartości jak Q_1 i Q_0 tylko po pewnym czasie, zwanym czasem opóźnienia lub czasem propagacji. Z zakodowanej tablicy przejść można znaleźć mapy Karnaugha poszukiwanych funkcji, które pokazano na rysunku 4.31. Na podstawie znalezionych map można wyznaczyć funkcje:

$$Q_0' = Q_0 x_1 + \overline{x_1}\,\overline{x_0}$$

$$Q_1' = Q_1 \overline{x_1} + \overline{Q_0} x_1 x_0 + Q_0 x_1 \overline{x_0}$$

Układ realizujący te funkcje pokazano na rysunku 4.32. Jest to układ działający według zadanej tablicy przejść, a więc jest to poszukiwany automat.

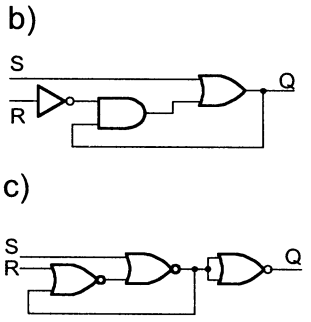
Druga możliwa struktura automatu asynchronicznego wykorzystuje przerzutniki typu RS lub $\overline{R}\,\overline{S}$. Przerzutniki te różnią się od wcześniej omówionych przerzutników stosowanych w automatach synchronicznych. Opierając się na tablicach przejść i wyjść przerzutnika RS, które podano w rozdziale dotyczącym automatów synchronicznych, poszukajmy rozwiązania



Rysunek 4.32. Układ realizujący tablicę przejść z rys. 4.25

a)

Q \ RS	00	01	11	10
0	0	1	-	0
1	1	1	-	0

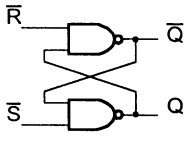


Rysunek 4.33. Przerzutnik asynchroniczny *RS*: a) tablica przejść, b) układ realizujący postać sumacyjną, c) układ na trzech bramkach NOR, d) układ na dwóch bramkach NOR

realizującego te tablice za pomocą bramek, tak jak to opisano. Tablicę przejść pokazano na rysunku 4.33a. Można zauważyć, że $Q' = S + \overline{R}Q$. Stąd układ realizujący postać sumacyjną jest pokazany na rysunku 4.33b. Zamieniając tę postać na postać NOR można otrzymać układ jak na rysunku 4.33c. Nietrudno zauważyć, że pomijając inwerter wyjściowy otrzyma się negację funkcji Q'. Zakładając, że nie zajdzie jednocześnie $S = 1$ i $R = 1$, otrzymamy układ z komplementarnymi wyjściami z dwóch bramek NOR pokazany na rysunku 4.33d.

Przez analogię do układu na bramkach NOR można zbudować układ na bramkach NAND. Układ pokazany na rysunku 4.34a jest przerzutnikiem $\overline{R}\,\overline{S}$. Jego tablica przejść jest pokazana na rysunku 4.34b.

a)



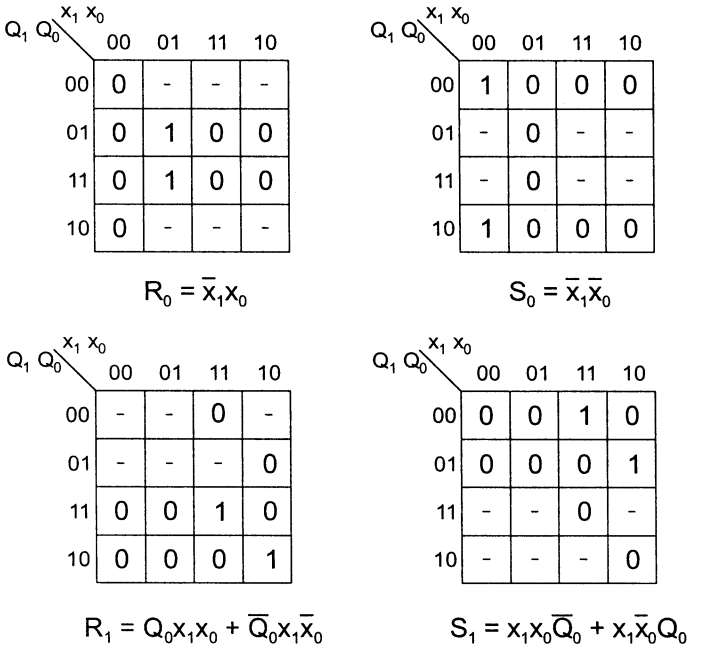
b)

Q \ R̄S̄	00	01	11	10
0	-	0	0	1
1	-	0	1	1

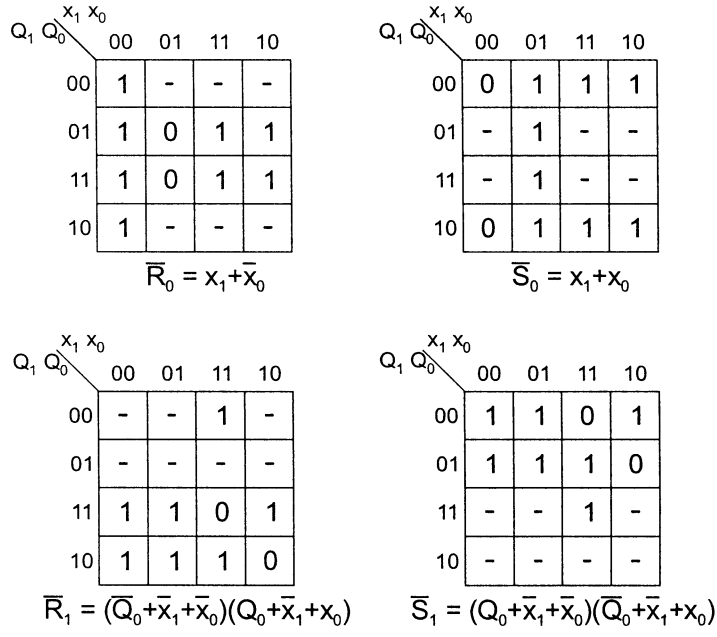
Rysunek 4.34. Stosowany układ przerzutników $\overline{R}\,\overline{S}$: a) układ, b) tablica przejść

Dowolny automat asynchroniczny można zbudować za pomocą przerzutników *RS* lub $\overline{R}\,\overline{S}$. Ich liczbę określa oczywiście liczba stanów automatu i musi być spełnione równanie, że $2^k \geq n$, gdzie k jest liczbą przerzutników, a n jest liczbą stanów automatu. Ponadto, podobnie jak w przypadku automatów synchronicznych, należy na podstawie tablicy przejść wyznaczyć funkcje wzbudzeń tych przerzutników. Procedura wyznaczania funkcji wzbudzeń jest taka sama jak dla automatów synchronicznych i można posłużyć się opisem z rysunku 4.3. Na rysunku 4.35 pokazano mapy Karnaugha funkcji wzbudzeń przerzutników *RS* dla automatu opisanego tablicą przejść 4.25. Podobnie można wyznaczyć funkcje wzbudzeń dla przerzutników $\overline{R}\,\overline{S}$. Mapy Karnaugha tych funkcji są pokazane na rysunku 4.36. Na rysunku 4.37 pokazano oba otrzymane rozwiązania.

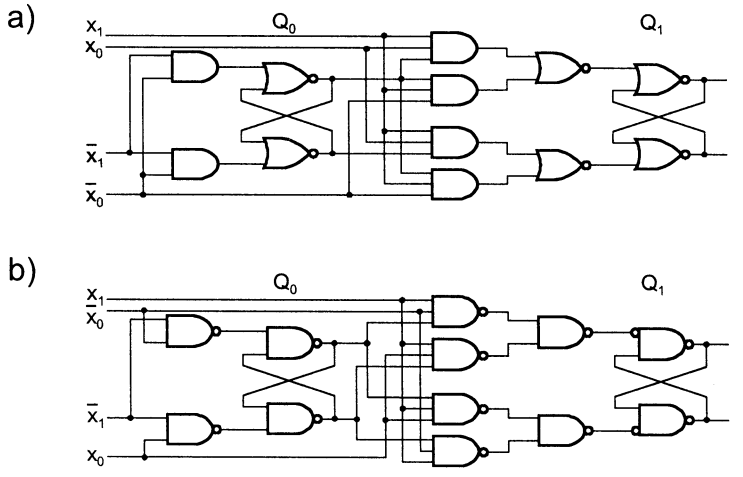
Wyznaczając funkcje wzbudzeń przerzutników należy pamiętać o zjawisku hazardu. W wyżej wyznaczonych funkcjach zjawisko to nie występuje. Natomiast zjawisko hazardu



Rysunek 4.35. Mapy Karnaugha funkcji wzbudzeń przerzutników RS automatu z tablicy 4.25



Rysunek 4.36. Mapy Karnaugha funkcji wzbudzeń przerzutników $\overline{R}\,\overline{S}$ dla automatu z tablicy 4.25



Rysunek 4.37. Układy realizujące przykładowy automat:a) na przerzutnikach *RS*, b) na przerzutnikach $\overline{R}\,\overline{S}$.

może wystąpić w realizacji tego automatu na samych bramkach (rozwiązanie 1). Z map Karnaugha pokazanych na rysunku 4.31 można zauważyć, że jeżeli funkcję Q_0 zrealizuje się jako $Q_0' = Q_0 x_1 + \bar{x}_1 \bar{x}_0$, to gdy $Q_0 = 1$ i $x_0 = 0$ i x_1 zmienia się, to może powstać zjawisko hazardu. Podobnie będzie dla funkcji $Q_1' = Q_1 \overline{x_1} + \overline{Q_0} x_1 x_0 + Q_0 x_1 \overline{x_0}$. Zjawisko hazardu można wyeliminować dodając implikanty funkcji, które nie wchodzą do postaci minimalnej, ale łączą ze sobą inne implikanty:

$$Q_0' = Q_0 x_1 + \overline{x_1}\,\overline{x_0} + Q_0 \overline{x_0}$$
$$Q_1' = Q_1 \overline{x_1} + \overline{Q_0} x_1 x_0 + Q_0 x_1 \overline{x_0} + Q_1 \overline{Q_0} x_0 + Q_1 Q_0 \overline{x_0}$$

Na mapie Karaugha widać to jako dodatkowe pola obejmujące rozdzielne pola postaci minimalnej.

4.3.5. Projektowanie automatów asynchronicznych

Proces projektowania automatów asynchronicznych składa się z 4 etapów:
— wyznaczenie tablic przejść i wyjść,
— minimalizacja tablicy przejść,
— kodowanie tablicy przejść,
— wyznaczenie funkcji realizujących tablice przejść i wyjść.

Wyznaczenie tablic przejść i wyjść automatu

Proces projektowania automatów asynchronicznych rozpoczyna się, tak jak w przypadku automatów synchronicznych, od opisu słownego. Metodą przejścia od opisu słownego do opisu formalnego (tablica przejść i wyjść lub graf), w przypadku automatów asynchronicznych, jest tzw. **wykres czasowy**. Ponieważ zmiany stanów automatu i zmiany sygnałów wyjściowych następują w momencie zmiany sygnałów wejściowych, to jest wygodnie przedstawić te zmiany graficznie w funkcji czasu.

Najpierw na podstawie opisu słownego trzeba wyznaczyć liczbę sygnałów wejściowych i wyjściowych projektowanego automatu. Następnie trzeba przyjąć jakieś zdarzenie przedstawione w opisie słownym za stan początkowy, któremu przypisze się wartości sygnałów wejściowych i wyjściowych. Następnie dokonując zmiany jednego z sygnałów wejściowych otrzymamy jakieś stany sygnałów wejściowych i wyjściowych i przypiszemy im następny stan projektowanego automatu. Następnie dla nowo powstałego stanu ponownie dokonuje się zmiany sygnału wejściowego i na podstawie opisu słownego przypisuje się mu stany sygnałów wyjściowych. Proces ten wykonuje się tak, aby po każdej zmianie stanu sygnału wyjściowego wyznaczany był albo nowy stan, albo aby automat dochodził do stanu, który pojawił się wcześniej. Po takim procesie projektant dojdzie do sytuacji kiedy nie powstają już nowe stany. Za stan nowy uważa się nowo powstałą kombinację sygnałów wejściowych i wyjściowych.

Nie oznacza to, że automat może mieć co najwyżej 2^{n+m} stanów, gdzie n jest liczbą sygnałów wejściowych a m wyjściowych. W ogólnym przypadku zasada ta nie obowiązuje, gdyż opis słowny może zakładać zmiany stanów, które nie powodują zmian sygnałów wyjściowych. Są to przypadki zliczania zdarzeń. Przykładem może być automat, którego sygnał wyjściowy y ma zmienić się po piątym impulsie wejściowym. Wówczas mimo zmian stanu automatu przez 4 impulsy wejściowe — sygnał wyjściowy nie ulega zmianie. Dopiero po piątym impulsie następuje zmiana sygnału wyjściowego. W takim przypadku do projektowania warto wprowadzić pomocnicze sygnały wyjściowe. W opisanym przykładzie przydatne byłyby dwa takie sygnały (dla czterech impulsów wejściowych).

Wykonując wykres czasowy należy pamiętać, że pełni on rolę służebną podczas projektowania automatu i ma prowadzić do zapełnienia tablic przejść i wyjść automatu. Dlatego należy tak konstruować wykres, aby pojawiły się wszystkie możliwe przejścia. Rozważania przedstawiono na przykładzie.

Przykład 4.8. Zaprojektować automat bramkujący impulsy generatora w taki sposób, że na jego wyjściu pojawiają się tylko pełne impulsy z generatora, tj. o czasie trwania równym czasowi trwania impulsów wejściowych.

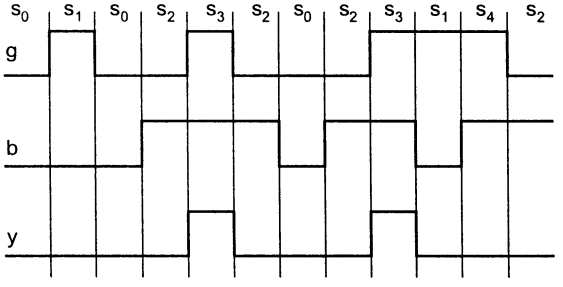
Interpretacja przykładu. Automat ma dwa wejścia: jedno g dla wejściowego generatora i drugie b dla impulsu bramkującego. Automat ma jedno wyjście y, na którym pojawiają się impulsy wyjściowe. Impuls bramkujący może pojawić się w dowolnym momencie, tj. asynchronicznie z impulsami generatora. Załóżmy, że bramkowanie następuje jedynką i przepuszczane są impulsy jedynkowe. Jeśli narastające (dodatnie) zbocze impulsu bramkującego trafi na jedynkę impulsu generatora, to ten impuls nie powinien zostać przepuszczony na wyjście.

Rozwiązanie. Na rysunku 4.38a pokazano wykres czasowy rozpoczynając od stanu gdy na wejściu oznaczonym literą g (generator) jest stan niski (brak impulsów z generatora) i na wejściu oznaczonym literą b (bramka) jest też stan niski. Oznaczono ten stan przez s_0. Po pojawieniu się impulsu z generatora (zmiana sygnału g) sygnał wyjściowy pozostaje w stanie 0, gdyż sygnał $b = 0$ i stan ten oznaczamy przez s_1. Zmiana sygnału g powoduje, że automat przechodzi ponownie do stanu s_0. Z tego stanu zmieniamy teraz drugą zmienną, czyli b. Powstaje nowy stan oznaczony przez s_2. Jeżeli teraz pojawi się impuls z generatora (zmiana sygnału g), to na wyjściu zmieniamy stan na 1 i oznaczamy ten stan automatu przez s_3. Jeśli założymy zmianę na wejściu g (skończony impuls generatora), to automat powraca do stanu s_2. Załóżmy teraz, że zmienimy zmienną b, to okaże się, że automat powróci do stanu pierwotnego s_0. Ponieważ na początku wykresu wychodząc ze stanu s_0 założyliśmy zmianę na wejściu g, to tym razem załóżmy zmianę na wejściu b. Teraz na wykresie

czasowym jest sytuacja, że $g = 0$, $b = 1$ i $y = 0$, a ta sytuacja odpowiada stanowi s_2. Ze stanu s_2 analizowaliśmy już oba przypadki zmian (stan s_2 pojawił się na wykresie czasowym już dwa razy), więc kontynuując wykres lepiej wybrać zmianę na wejściu g, gdyż zmiana na wejściu b doprowadziłaby ponownie do stanu s_0. Zatem drugi raz na wykresie pojawia się stan s_3. Rysując wykres dalej należy wybrać zmianę na wejściu b (gdyż wychodząc ze stanu s_3 zmiana na wejściu g już była uwzględniona), a więc przejście do stanu s_1.

W tym miejscu trzeba zauważyć zrobione założenie, że na wyjściu pojawi się „obcięty" impuls. Nie spełnia to warunków zadania, choć można sądzić, że w tekście zadania chodziło jedynie o obcinanie przedniej części impulsu. Aby nie dopuścić do obcinania również tylnej części impulsu trzeba projektować inny automat. Zostanie on zaprojektowany jako wersja druga tego przykładu.

a)



b)

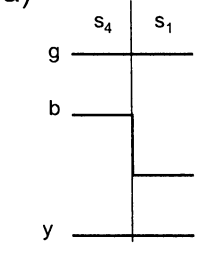
S \ g b	00	01	11	10	y
s_0	(**s_0**)	s_2	-	s_1	0
s_1	s_0	-	s_4	(**s_1**)	0
s_2	s_0	(**s_2**)	s_3	-	0
s_3	-	s_2	(**s_3**)	s_1	1
s_4	-	s_2	(**s_4**)	-	0

Rysunek 4.38. Wykres czasowy i tablice przejść i wyjść automatu z przykładu 4.8

Ze stanu s_1, wraz ze zmianą sygnału wejściowego b na wysoki, automat przejdzie do stanu s_4. Przy zmianie sygnału g na niski automat przejdzie do stanu s_2.

Dalsze działanie projektanta zależy silnie od jego wprawy. Na rysunku 4.38b pokazano tablicę przejść i wyjść projektowanego automatu wyznaczoną na podstawie dotychczasowego wykresu. W tablicy zaznaczono tzw. stany stabilne, tj. stany które występują dla danego wzbudzenia. Widać, że dla wzbudzenia 00 jest stan stabilny s_0, dla wzbudzenia 01 jest stan stabilny s_2, dla wzbudzenia 10 jest stan stabilny s_1, a dla wzbudzenia 11 jest stan stabilny s_3

a)



b)

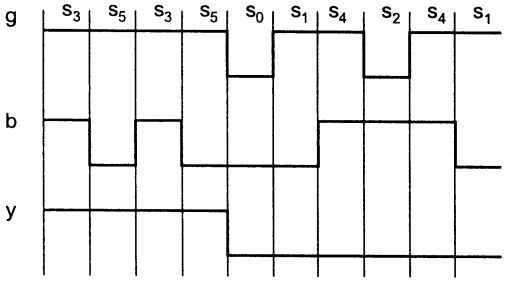
S \ g b	00	01	11	10	y
s_0	(**s_0**)	s_2	-	s_1	0
s_1	s_0	-	s_4	(**s_1**)	0
s_2	s_0	(**s_2**)	s_3	-	0
s_3	-	s_2	(**s_3**)	s_1	1
s_4	-	s_2	(**s_4**)	s_1	0

Rysunek 4.39. Uzupełnienie wykresu czasowego i tablice przejść i wyjść projektowanego automatu

i s_4. Dobrze określony automat powinien mieć stany stabilne w każdej kolumnie (w innym przypadku jest generatorem) i stany stabilne w każdym wierszu (w innym przypadku dany stan występuje tylko w czasie zmian stanu automatu — automat nie zostaje w tym stanie). Z tablicy z rysunku 4.38b widać, że jedno miejsce w tablicy nie zostało wypełnione (przejście ze stanu s_4 pod wpływem wzbudzenia 10). Na rysunku 4.39a pokazano dalszą część wykresu czasowego poczynając od stanu s_4. Po zmianie sygnału b na niski automat przechodzi do stanu s_1. Kończy to proces tworzenia wykresu czasowego i wypełniania tablicy przejść i wyjść pokazanej na rysunku 4.39b.

Wyznaczmy teraz tablicę przejść i wyjść układu bramkowania przy założeniu, że na wyjściu układu mogą pojawić się tylko niezniekształcone impulsy. Wykres czasowy z poprzedniej wersji automatu jest, w tym przypadku poprawny, aż do stanu s_3. Aby na wyjściu

a)



b)

S \ g b	00	01	11	10	y
S0	(S0)	S2	-	S1	0
S1	S0	-	S4	(S1)	0
S2	S0	(S2)	S3	-	0
S3	-	S2	(S3)	S5	1
S4	-	S2	(S4)	S1	0
S5	S0	-	S3	(S5)	1

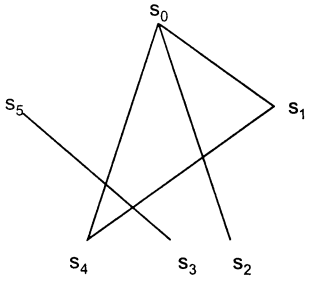
Rysunek 4.40. Zmodyfikowany wykres czasowy i tablica przejść i wyjść automatu z przykładu 4.8

pojawił się pełny impuls, to automat ze stanu s_3 musi przejść do nowego stanu s_5, w którym $g = 1$, $b = 0$ i $y = 1$. Pokazano to na rysunku 4.40a. Jeśli ponownie nastąpi zmiana sygnału na wejściu b, to automat przejdzie do stanu s_3, a po powtórnej zmianie tego sygnału automat powróci do stanu s_5. Jeśli w stanie s_5 zmieni się sygnał g, to automat przejdzie do stanu s_0. Ponieważ nie powstał żaden nowy stan, to kończy się wykres czasowy, a 6-stanowa tablica przejść pokazana jest na rysunku 4.40b. ⊠

Minimalizacja tablicy przejść

Otrzymane, na podstawie wykresu czasowego, tablice przejść i wyjść nie muszą opisywać automatu o minimalnej liczbie stanów realizującego dane zadanie. Proces minimalizacji liczby stanów automatu asynchronicznego jest łatwiejszy niż w przypadku automatu synchronicznego. Wynika to z faktu, że zgodność stanów określa się jedynie dla stanów stabilnych. Warunkami zgodności stanów mogą być tylko stany stabilne, które występują w tej samej kolumnie tablicy przejść. Dlatego minimalizację liczby stanów prowadzi się w dwóch etapach: w pierwszym próbuje się łączyć stany stabilne, które występują w jednej kolumnie oraz w drugim, w którym poszukuje się minimalnej liczby grup stanów zgodnych, które pokrywają zbiór wszystkich stanów. Grupy stanów zgodnych wyszukuje się za pomocą tzw. wykresu zgodności stanów. Tablica przejść i wyjść pokazana na rysunku 4.40b tylko w trzeciej kolumnie (wzbudzenie11) zawiera dwa stany stabilne, ale ponieważ mają one niezgodne wyjścia, to i one nie są zgodne. Wykres zgodności dla tej tablicy przejść pokazano na

a)



b)

S \ g b	00	01	11	10	y
S0,S1,S4	(0)	1	(0)	(0)	0
S2	1	(1)	2	-	0
S3,S5	2	1	(2)	(2)	1

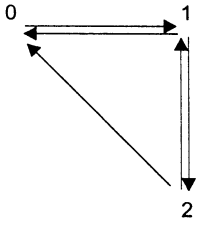
Rysunek 4.41. Ilustracja minimalizacji tablicy przejść i wyjść

rysunku 4.41a. Stany s_0, s_1 i s_4 są wzajemnie zgodne i można zastąpić je jednym stanem, podobnie jak stany s_3 i s_5. Wynikowa tablica przejść i wyjść jest pokazana na rysunku 4.41b.

Kodowanie stanów

Otrzymane w procesie minimalizacji liczby stanów tablice przejść i wyjść trzeba zakodować. W tym celu tworzy się sześcian (dla tablicy z rysunku 4.41 jest to kwadrat), a ponieważ są tylko 3 stany, to graf przejść jest taki jak pokazany na rysunku 4.42a. Z grafu wynika, że nie istnieje takie zakodowanie, aby można było uniknąć zmiany więcej niż jednego bitu podczas

a)



b)

Q_1Q_0 \ g b	00	01	11	10	y
00	00	01	00	00	0
01	00	01	11	-	0
11	10	01	11	11	1
10	00	-	-	-	-

Rysunek 4.42. Kodowanie tablicy przejść i wyjść przykładowego automatu

każdej zmiany stanu. Dlatego przyjmując zakodowanie, że stanowi 0 przypiszemy 00, stanowi 1 przypiszemy 01 i stanowi 2 przypiszemy 11 należy wykorzystać także stan 10 i zmianę stanu z 11 na 00 dokonać poprzez ten stan. Otrzyma się wtedy tablicę przejść pokazaną na rysunku 4.42b.

Realizacja automatu

Zrealizujmy teraz wyznaczoną tablicę przejść i wyjść na bramkach. W tym celu, na podstawie tablicy przejść z rysunku 4.42b należy znaleźć mapy Karnaugha, które pokazano na rysunku 4.43. Na podstawie tych map można znaleźć, że:

$Q_1 Q_0$ \ g b	00	01	11	10
00	0	0	0	0
01	0	0	1	-
11	1	0	1	1
10	0	-	-	-

$Q_1' = Q_0 g + Q_0 Q_1 \bar{b}$

$Q_1 Q_0$ \ g b	00	01	11	10
00	0	1	0	0
01	0	1	1	-
11	0	1	1	1
10	0	-	-	-

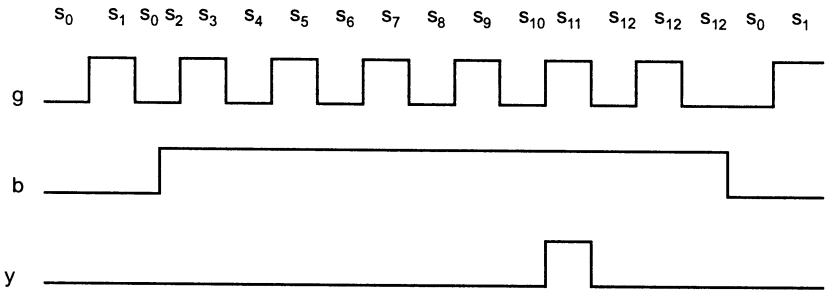
$Q_0' = \bar{g} b + Q_0 g$

Rysunek 4.43. Kodowanie tablicy przejść i wyjść przykładowego automatu

$$Q_1' = Q_0 g + Q_1 Q_0 \bar{b}$$
$$Q_0' = Q_0 g + \bar{g} b$$

Przykład 4.9. Wyznaczyć tablicę przejść i wyjść automatu bramkującego impulsy generatora w taki sposób, że na jego wyjściu pojawia się tylko jeden niezniekształcony impuls, który jest piątym z kolei impulsem po pojawieniu się narastającego zbocza sygnału bramkującego.

a)



b)

S \ g b	00	01	11	10	y
s_0	(s_0)	s_2	-	s_1	0
s_1	s_0	-	s_3	(s_1)	0
s_2	(s_0)	(s_2)	s_3	-	0
s_3	-	s_4	(s_3)	(s_1)	0
s_4	(s_0)	(s_4)	s_5	-	0
s_5	-	s_6	(s_5)	(s_1)	0
s_6	(s_0)	(s_6)	s_7	-	0
s_7	-	s_8	(s_7)	(s_1)	0
s_8	(s_0)	(s_8)	s_9	-	0
s_9	-	s_{10}	(s_9)	(s_1)	0
s_{10}	(s_0)	(s_{10})	s_{11}	-	0
s_{11}	-	s_{12}	(s_{11})	(s_1)	1
s_{12}	(s_0)	(s_{12})	s_{12}	-	0

Rysunek 4.44. Wykres czasowy (a) i tablica przejść i wyjść (b) automatu z przykładu 4.9

Interpretacja przykładu. Automat działa w taki sposób, że po przyjściu narastającego zbocza sygnału *b* odlicza cztery impulsy generatora i dopiero piąty impuls przepuszcza na wyjście. Trzeba jednak przyjąć pewne założenia.

Pierwsze założenie dotyczy momentu pojawienia się dodatniego zbocza sygnału *b*. Może ono pojawić się w czasie gdy sygnał *g* jest równy 0, ale może pojawić się także w momencie, gdy sygnał *g* jest równy 1. W tym drugim przypadku trzeba założyć, czy ten impuls generatora jest liczony jako pierwszy, czy też jest pomijany. Nie odgrywa to szczególnej roli, ponieważ projekt ma uwzględniać liczbę impulsów generatora, a więc może ona być korygowana. Założymy tutaj, że impuls generatora, w czasie którego pojawiło się narastające zbocze sygnału *b*, jest liczony jako pierwszy.

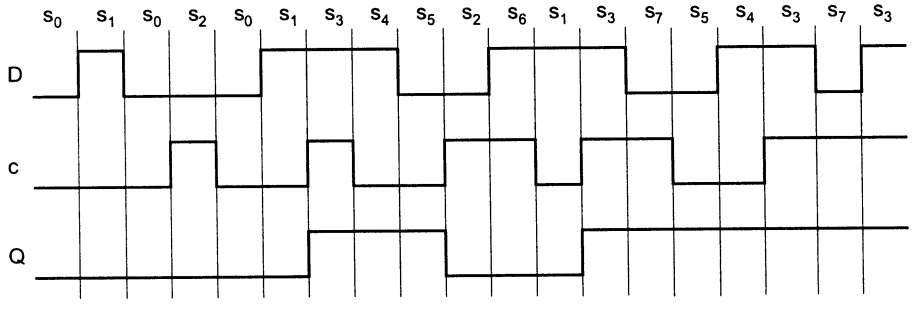
Drugie założenie dotyczy momentu pojawienia się opadającego zbocza sygnału bramkującego. Jeśli pojawi się ono po narastającym zboczu piątego impulsu generatora, to oczywiście przechodzi on na wyjście. Jeśli opadające zbocze sygnału bramkującego pojawi się w czasie trwania piątego impulsu generatora, to także ma on pojawić się na wyjściu. Jeśli jednak opadające zbocze sygnału bramkującego pojawi się przed piątym impulsem generatora, to automat ma rozpocząć odliczanie ponownie po przyjściu narastającego zbocza.

Rozwiązanie. Na rysunku 4.44a pokazano część wykresu czasowego projektowanego automatu, a na rysunku 4.44b tablicę przejść i wyjść. Niektóre przejścia nie wynikają z podanego wykresu, ale można je otrzymać po odpowiednim uzupełnieniu wykresu. Otrzymano automat o 13 stanach. Przeprowadzając minimalizację można otrzymać automat 11-stanowy (trzy pierwsze stany można zastąpić jednym). ⊠

Przykłady 4.10. Zaprojektować przerzutnik synchroniczny typu *D*.

Rozwiązanie. Przerzutnik typu *D* jest dwuwejściowym automatem asynchronicznym, którego wejściami są: wejście *D* oraz wejście zegarowe *c*. Wykres czasowy takiego układu pokazano na rysunku 4.45. Założono, że zmiana stanu przerzutnika będzie następowała po narastającym (dodatnim) zboczu impulsu zegara.

Z wykresu czasowego wynika, że automat jest 8-stanowy. Ponieważ są dwa wejścia, to tablica przejść tego automatu ma 4 kolumny. Z wykresu czasowego trzeba dla każdego stanu odczytać dwie zmiany sygnałów wejściowych, więc każdy stan musi wystąpić na wykresie co najmniej dwa razy. Z rysunku 4.45 można zauważyć, że stan s_6 pojawia się tylko jeden raz. Oznacza to, że jednej ze zmian stanu nie można wyznaczyć. W tablicy przejść pokazanej w tablicy 4.26 wyróżniono to miejsce (druga kolumna). Aby wpisać tam pożądaną wartość należy albo kontynuować wykres czasowy pokazany na rysunku 4.45, albo rozpocząć nowy



Rysunek 4.45. Wykres czasowy dla przerzutnikw typu *D*

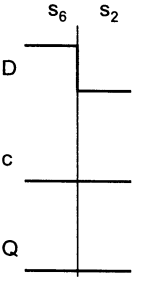
Tablica 4.26. Tablica przejść i wyjść przerzutnika typu D (strzałką zaznaczono miejsce wyznaczone na podstawie rysunku 4.46)

Q \ D c	00	01	11	10	y
s_0	(s_0)	s_2	-	s_1	0
s_1	s_0	-	s_3	(s_1)	0
s_2	s_0	(s_2)	s_6	-	0
s_3	-	s_7	(s_3)	s_4	1
s_4	s_5	-	s_3	(s_4)	1
s_5	(s_5)	s_2	-	s_4	1
s_6	-	s_2	(s_6)	s_1	0
s_7	s_5	(s_7)	s_3	-	1

wykres. W przypadku gdy automat jest złożony, łatwiejszy jest ten drugi sposób. Na rysunku 4.46 pokazano dokończenie wykresu czasowego, tj. tylko jedną brakującą zmianę — jest to zmiana ze stanu s_6 na stan s_2.

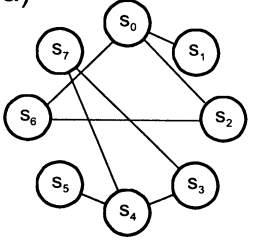
Otrzymana z wykresu czasowego tablica przejść i wyjść nosi nazwę tablicy pierwotnej. Aby dokonać minimalizacji liczby stanów sporządza się graf skracania pokazany na rysunku 4.47a i następnie zminimalizowaną tablicę przejść i wyjść pokazaną na rysunku 4.47b. Otrzymano czterostanową tablicę przejść i wyjść przerzutnika typu D. Wyznaczając graf przejść dla tej tablicy pokazany na rysunku 4.48a, można uzyskać jeden z możliwych sposobów zakodowania minimalnej tablicy przejść i wyjść tak, aby nie występowało zjawisko wyścigu. Zakodowaną tablicę przejść i wyjść pokazano na rysunku 4.48b.

Załóżmy, że układ należy zrealizować na asynchronicznych przerzutnikach $\overline{R}\,\overline{S}$ oraz na bramkach NAND. Na rysunku 4.49 pokazano mapy Karnaugha funkcji wzbudzeń tych



Rysunek 4.46. Dokończenie wykresu czasowego dla przerzutników typu D

a)

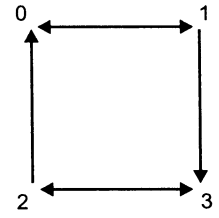


b)

S \ D c	00	01	11	10	y
(s_0,s_2,s_6) 0	(0)	(0)	(0)	1	0
s_1 1	0	-	3	(1)	0
(s_3,s_4,s_7) 3	2	(3)	(3)	(3)	1
s_5 2	(2)	0	-	3	1

Rysunek 4.47. Graf skracania dla tablicy z rysunku 4.26 (a) i zminimalizowana tablica przejść i wyjść przerzutników typu D (b)

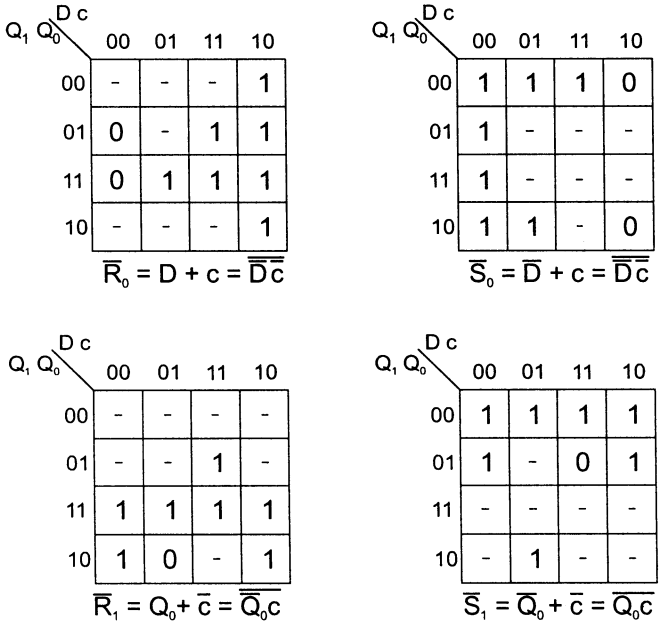
a)



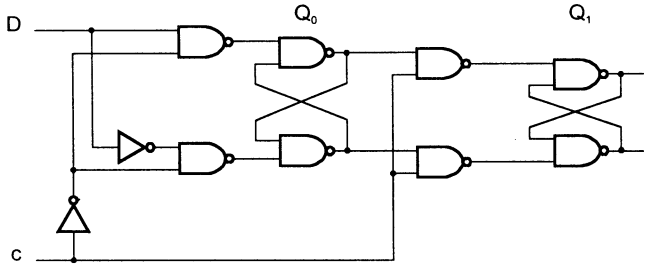
b)

Q_1Q_0 \ D c	00	01	11	10	y
00	00	00	00	01	0
01	00	-	11	01	0
11	10	11	11	11	1
10	10	00	-	11	1

Rysunek 4.48. Graf przejść (a) i zakodowana tablica przejść przerzutników typu *D* (b)



Rysunek 4.49. Mapy Karnaugha funkcji wzbudzeń przerzutników $\overline{R}\,\overline{S}$ dla przerzutnika typu *D*



Rysunek 4.50. Schemat projektowanego przerzutnika typu *D*

przerzutników. Zgodnie z wcześniejszymi uwagami po minimalizacji funkcji wzbudzeń należy sprawdzić czy nie występuje zjawisko hazardu. W podanym przykładzie zjawisko to nie występuje, gdyż w każdej z funkcji występuje tylko jedna grupa zer. Ponadto należy sprawdzić, czy w procesie minimalizacji funkcji nie powstał przypadek, że zarówno wejście $\overline{R}$ jak i $\overline{S}$ będą zerami dla jakiejś kombinacji zmiennych. W omawianym przykładzie przypadek taki nie powstał. Na rysunku 4.50 przedstawiono projektowany układ. ⊠

4.4. Automaty standardowe

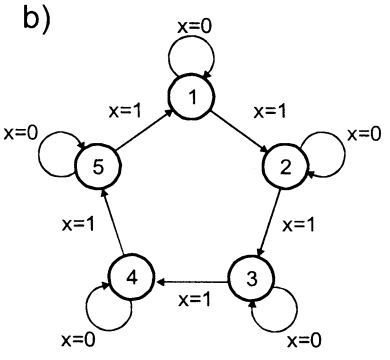
4.4.1. Liczniki

W praktyce często są stosowane automaty standardowe, które projektanci wykorzystują jako bloki wchodzące w skład projektowanych przez nich większych układów. Rozpatrzymy dwie grupy takich układów: liczniki i rejestry.

a)

S \ x	0	1
1	1	2
2	2	3
3	3	4
4	4	5
5	5	1

b)



Rysunek 4.51. Tablica przejść (a) i graf licznika pięciostanowego (b)

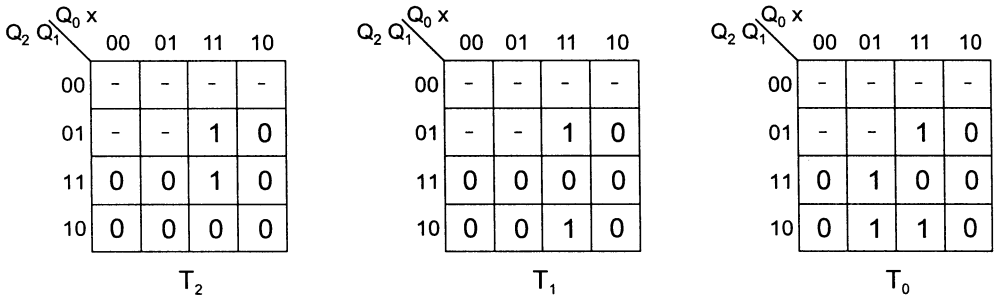
Liczniki są to układy zliczające sygnały wejściowe, a w niektórych przypadkach impulsy zegarowe. Najprostsze liczniki opisane są tablicą przejść pokazaną na rysunku 4.51a i grafem pokazanym na rysunku 4.51b. Jak widać, z takiego opisu nie wynika czy mamy do czynienia z automatem synchronicznym, czy asynchronicznym. Odpowiedź na takie pytanie zależy od realizacji. Tablicę taką można zrealizować jako automat synchroniczny lub asynchroniczny. Zaprojektujmy obie takie realizacje.

Synchroniczny licznik działający wg tablicy przejść i wyjść z rysunku 4.51

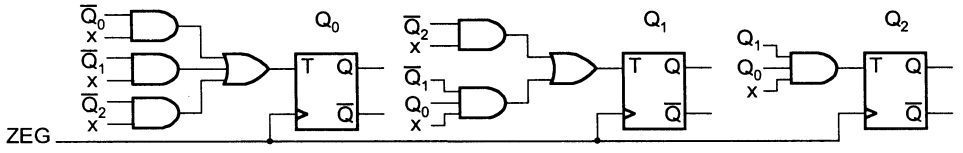
Projektowany licznik ma 5 stanów, które można zakodować na wiele różnych sposobów. Przyjmijmy tutaj, że dokonamy syntezy licznika na trzech przerzutnikach typu T oraz, że

Tablica 4.27. Zakodowana tablica licznika pięciostanowego opisanego tablicą przejść z rys. 4.51a

$Q_2\ Q_1\ Q_0$ \ x	0	1
011	011	100
100	100	101
101	101	110
110	110	111
111	111	011



Rysunek 4.52. Mapy Karnaugha funkcji wzbudzeń typu *T* dla tablicy przejść z tab. 4.27



Rysunek 4.53. Układ realizujący tablicę przejść z rysunku 4.51

kolejnym stanom przypiszemy kolejne liczby w kodzie NKB. Załóżmy, że pierwszemu stanowi przypiszemy słowo 011, drugiemu 100, trzeciemu 101, czwartemu 110 i piątemu 111. Mówimy wówczas, że licznik liczy od 3 do 7. Założenie takie powoduje, że otrzymamy zakodowaną tablicę przejść pokazaną w tablicy 4.27. Na rysunku 4.52 pokazano 3 mapy Karnaugha dla funkcji wzbudzeń przerzutników T_0, T_1 i T_2. Zakładając, że wyjściami układu będą wyjścia przerzutników otrzyma się układ jak na rysunku 4.53.

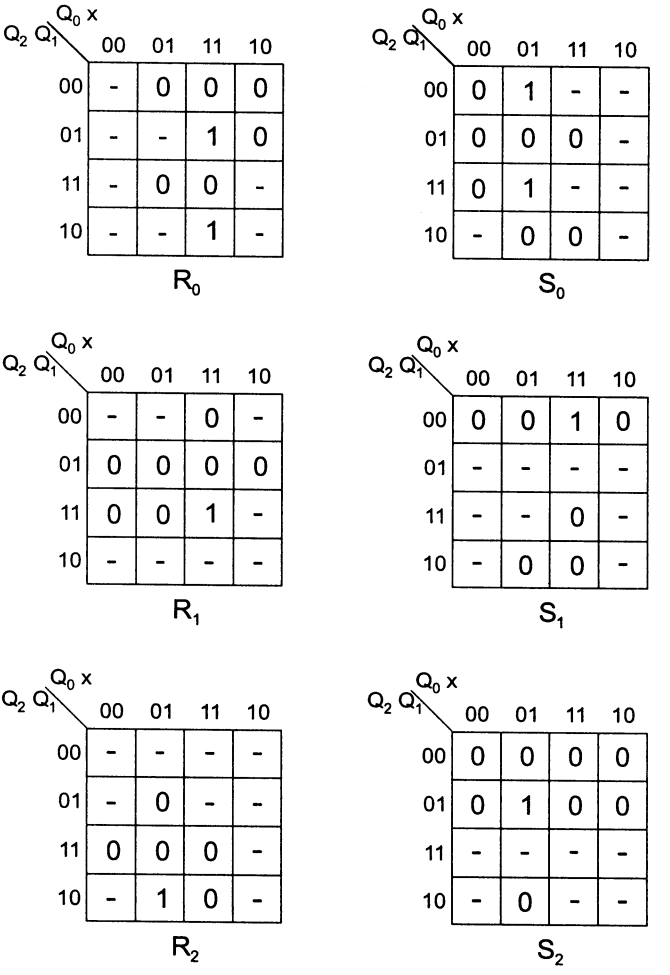
Asynchroniczny licznik działający wg tablicy przejść i wyjść z rysunku 4.51

Stany projektowanego licznika należy zakodować tak, aby nie wystąpiło zjawisko wyścigu. Przyjmując zakodowanie jak dla licznika synchronicznego otrzymamy, że w pierwszym wierszu wystąpią zmiany wszystkich trzech przerzutników. Zatem taki sposób zakodowania jest niedobry.

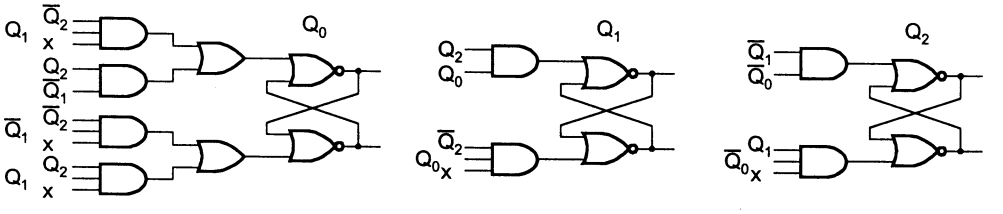
W tablicy 4.28 pokazano zakodowaną tablicę przejść projektowanego automatu przyjmując kod Graya. Aby uniknąć wyścigów dodano dodatkowe stany (111, 101 i 100) i dodatkowe przejścia ze stanu 110 do stanu 000. Na rysunku 4.54 pokazano mapy Karnaugha dla funkcji wzbudzeń trzech przerzutników *RS*, a na rysunku 4.55 pokazano cały automat.

Tablica 4.28. Tablica przejść licznika zakodowana według kodu Graya

$Q_2\ Q_1\ Q_0 \backslash x$	0	1
000	000	001
001	001	011
011	011	010
010	010	110
110	110	111
111	-	101
101	-	100
100	-	000

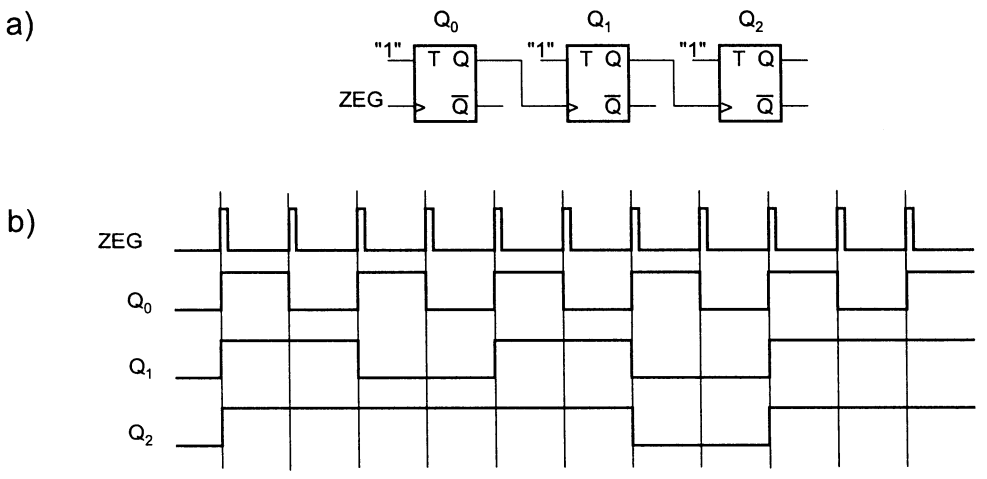


Rysunek 4.54. Funkcje wzbudzeń przerzutnika *RS* przykładowego automatu



Rysunek 4.55. Układ asynchroniczny realizujący automat działający według tablicy przejść z tablicy 4.28

W przypadku rozwiązania synchronicznego licznik ma dwa wejścia: wejście zliczające i wejście zegarowe. W takim układzie następuje zliczanie impulsów z wejścia zliczającego, ale tylko tych, które pojawią się synchronicznie z impulsami zegarowymi. Jest to pewna wada układu i dlatego niekiedy stosuje się inne rozwiązanie polegające na tym, że wejście



Rysunek 4.56. Szeregowy licznik poprzednikowy 8-stanowy: a) układ, b) wykres czasowy

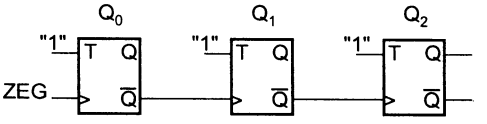
zegarowe jest wykorzystywane jako wejście zliczające, a wejście zliczające x jest wtedy wejściem dostępu E. Jeśli $E = 0$, to licznik nie zlicza (automat pozostaje w stanie w którym był), a jeśli $E = 1$, to następuje proces zliczania impulsów zegarowych. W przypadku rozwiązania asynchronicznego jest tylko jedno wejście i jest ono wejściem zliczającym.

Z przedstawionego opisu wynika, że można zaprojektować wiele rodzajów liczników i dlatego przedstawimy tutaj pewne klasyfikacje. W tym celu rozpatrzmy działanie układu pokazanego na rysunku 4.56a. W układzie tym na wejścia przerzutników typu T podano na stałe wartość 1, co powoduje, że zmieniają one swój stan przy każdym impulsie zegarowym (z definicji tego przerzutnika). Natomiast impulsy podawane na wejście zegarowe pochodzą z różnych źródeł, co oznacza, że automat jest asynchroniczny, choć zrealizowany z synchronicznych przerzutników typu T. Zakładając, że przerzutnik zmienia stan w czasie narastającego (z 0 na 1) zbocza impulsu na wejściu zegarowym, otrzyma się wykres czasowy pokazany na rysunku 4.56b. Ilustruje on przebiegi zmian przerzutników na tle przebiegu zegarowego bez uwzględnienia czasów opóźnienia. Liczba stanów (**okres licznika**) wynosi 8, a kolejność stanów przedstawiana w kodzie binarnym zmienia się według porządku 7, 6, 5, 4, 3, 2, 1 i 0. Taki licznik nazywa się **szeregowym licznikiem poprzednikowym** (ang. *up-down counter*).

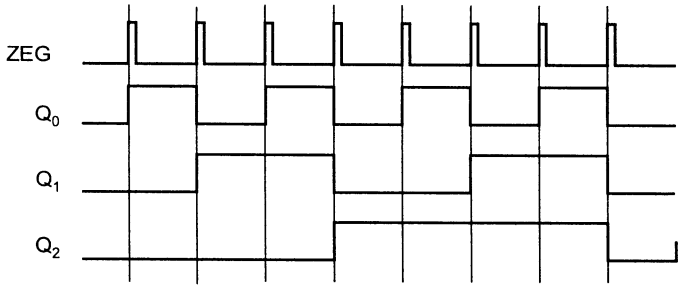
Słowo *szeregowy* ma oddawać asynchroniczność zmian stanów przerzutników, a słowo poprzednikowy ma oddawać kolejność binarną jego stanów. Można zaprojektować licznik zmieniający stany w czasie opadającego zbocza zegara. Na rysunku 4.57a pokazano **szeregowy licznik następnikowy** (ang. *bottom-up counter*), a na rysunku 4.57b wykres czasowy jego działania. Należy zauważyć, że zmiana stanów przerzutników Q_2 i Q_1, pokazana w czasie opadającego zbocza przerzutnika poprzedniego, wynika z tego, że wejścia zegarowe są sterowane z zanegowanych wyjść poprzedniego przerzutnika. Sekwencja stanów licznika następnikowego to stany 0, 1, 2, 3, 4, 5, 6 i 7.

Na rysunku 4.58a przedstawiono tablicę przejść **następnikowego licznika równoległego**, tj. licznika synchronicznego, a na rysunku 4.58b jego układ zrealizowany na przerzutnikach typu T. Na rysunku 4.58c przedstawiono poprzednikowy licznik równoległy.

a)



b)



Rysunek 4.57. Seryjny licznik następnikowy 8-stanowy: a) układ, b) wykres czasowy

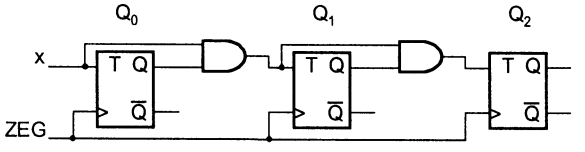
Można zauważyć, że funkcja wzbudzeń i-tego przerzutnika licznika następnikowego, to $T_i = xQ_0Q_1Q_2 \ldots Q_{i-1}$, a licznika poprzednikowego $T_i = x\overline{Q}_0\overline{Q}_1\overline{Q}_2 \ldots \overline{Q}_{i-1}$.

Niekiedy istnieje potrzeba dynamicznej (w czasie pracy licznika) zmiany kierunku liczenia (poprzednikowy na następnikowy lub odwrotnie). Można to osiągnąć wprowadzając

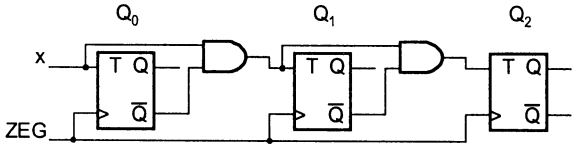
a)

$Q_2\,Q_1\,Q_0 \backslash x$	0	1
000	000	001
001	001	011
011	011	010
010	010	110
110	110	111
111	111	101
101	101	100
100	100	000

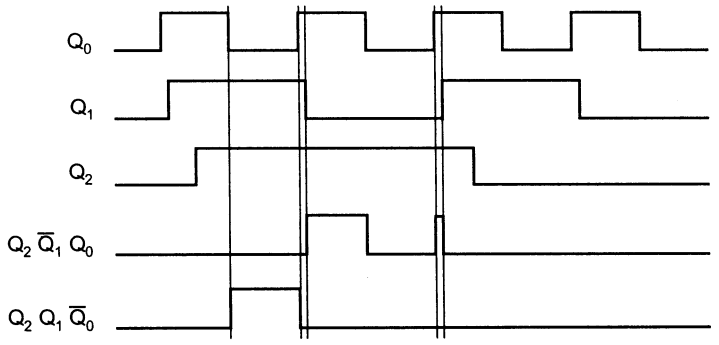
b)



c)



Rysunek 4.58. Liczniki równoległe: a) tablica przejść licznika następnikowego, b) licznik następnikowy c) licznik poprzednikowy

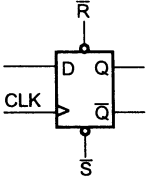


Rysunek 4.59. Przebiegi czasowe w liczniku poprzednikowym

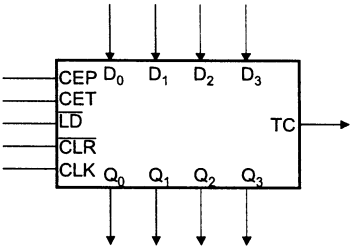
dodatkowe wejście sterujące, a takie liczniki są nazywane **licznikami rewersyjnymi**. Funkcje wzbudzeń takiego licznika są złożeniem funkcji wzbudzeń licznika następnikowego i poprzednikowego.

Zmiany stanów przerzutników w licznikach równoległych następują w tym samym momencie i dlatego ich sygnały wyjściowe także zmieniają się w tym samym czasie (z dokładnością do różnicy w opóźnieniach samych przerzutników). Natomiast inaczej jest w licznikach szeregowych. Wadą liczników szeregowych jest to, że zmiany stanów kolejnych przerzutników następują z pewnym opóźnieniem względem siebie. Może to powodować pewne niekorzystne zjawiska, szczególnie podczas dekodowania stanów. Rozpatrzmy przebiegi czasowe pokazane na rysunku 4.56, ale uwzględniając opóźnienia wprowadzane przez przerzutniki. Załóżmy ponadto, że zachodzi potrzeba zdekodowania stanów 5 i 6 tego licznika. Na rysunku 4.59 pokazano przebiegi na wyjściach przerzutników i dwóch iloczynów dekodujących stany 5 i 6. Na wyjściu iloczynu dekodującego stan 6 pojawi się impuls w czasie zaznaczonym na rysunku, tj. w spodziewanym momencie. Natomiast na wyjściu iloczynu dekodującego stan 5 pojawi się impuls nie tylko w spodziewanym czasie zaznaczonym na rysunku, ale także w momencie kiedy spodziewany jest stan 3. Ten dodatkowy impuls ma czas trwania równy opóźnieniu wnoszonemu przez przerzutnik. Jest to szkodliwe zjawisko hazardu.

W serii układów TTL istnieje układ scalony o numerze SN 7474, który zawiera dwa przerzutniki synchroniczne typu *D*. Przerzutniki te można używać także jako asynchroniczne przerzutniki $\overline{R}\,\overline{S}$. Na rysunku 4.60 pokazano schemat takiego przerzutnika. Układ taki stosuje

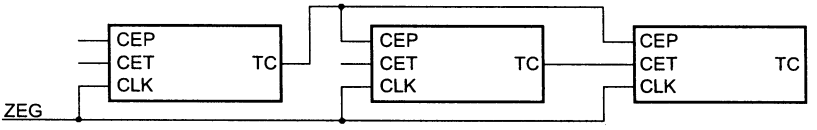


Rysunek 4.60. Przerzutnik synchroniczny typu *D* wraz z asynchronicznymi wejściami zerującym i ustawiającym



Rysunek 4.61. Równoległy licznik 4-bitowy

się w automatach synchronicznych, ale wejścia $\overline{R}$ i $\overline{S}$ mogą służyć na przykład do wstępnego ustawiania stanu. W serii układów TTL produkowane są liczniki szeregowe (np.SN 7493) i liczniki równoległe. Na rysunku 4.61 pokazano oznaczenie typowego 4-bitowego licznika równoległego SN 7416x. Licznik jest wyposażony w wejście zegarowe *CLK* oraz w cztery wejścia sterujące. Wejście sterujące *LD* (ang. *load*) służy do wpisywania danych z wejść $D_0 - D_3$ do licznika. Jest to wejście o oddziaływaniu zerem logicznym (wejście zanegowane) może być wejściem synchronicznym i może być wejściem asynchronicznym. Wejście zerujące *CLR* (ang. *clear*) służy do wyzerowania wszystkich przerzutników zerem logicznym (wejście zanegowane) i też może być wejściem synchronicznym lub asynchronicznym. Synchroniczność tych wejść oznacza, że wpisanie stanu lub wyzerowanie licznika nastąpi po przyjściu sygnału *LD* lub *CLR* i impulsu zegarowego. Natomiast asynchroniczność oznacza, że wpisanie stanu lub wyzerowanie licznika nastąpi od razu po przyjściu sygnału *LD* lub *CLR*.



Rysunek 4.62. Kaskadowe połączenie liczników 4-bitowych

Ponadto są dwa wejścia dostępu: *CEP* (ang. *count enable parallel*) umożliwiające zliczanie i *CET* (ang. *count enable trickle*) umożliwiające powstanie przeniesienia *TC* (ang. *terminal count*). Wyjście *TC* służy do łączenia licznika z innymi takimi samymi układami. Na rysunku 4.62 pokazano sposób łączenia trzech 4-bitowych liczników celem zwiększenia okresu liczenia do 4096 stanów. W układzie występuje pełny synchronizm z przebiegiem zegarowym, gdyż wszystkie przerzutniki wszystkich liczników zmieniają swoje stany jednocześnie.

Często występuje problemem projektowania liczników o różnych okresach. Przykładowo można na wejście *LD* podać sygnał *TC*, który jest generowany po osiągnięciu stanu 1111. W takim układzie po stanie 1111 następuje załadowanie do licznika nowej zawartości. W zależności od tego czy wejście *LD* jest synchroniczne, czy asynchroniczne dalsze działanie będzie różne. Gdy wejście *LD* jest synchroniczne, to w czasie następnego impulsu zegarowego do licznika zostanie wpisana wartość z jego wejść. Jeśli przykładowo będzie to 1010 (dziesiętnie 5), to okres takiego licznika wynosi zatem $16 - 5 = 11$, a sekwencja stanów

przebiega od 1010 do 1111. Gdy wejście *LD* jest asynchroniczne, to do licznika wpisywana jest wartość z jego wejść w momencie pojawienia się sygnału *LD*. Wobec tego załadowanie stanu 5 nastąpi od razu i w tym samym momencie zakończy się stan 1111 i zaniknie sygnał *TC*. Zatem stan 1111 będzie trwał tylko przez czas opóźnienia sygnału w układzie. Okres takiego licznika wynosi $15-5=10$, a sekwencja stanów przebiega od 1010 do 1110 (stan 1111 pojawia się na krótko). Podobny efekt można osiągnąć przez dekodowanie jakiegoś stanu licznika i podanie na wejście *LD* wyjścia z dekodera.

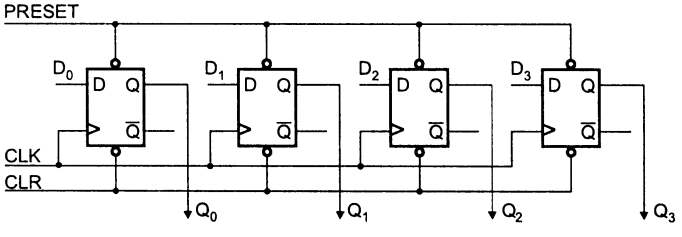
Inny sposób zmiany okresu liczenia liczników można zrealizować wykorzystując wejście *CLR*. W tym przypadku dekoduje się wybrany stan licznika i z wyjścia dekodera podaje się sygnał na wejście *CLR*. Załóżmy, że za pomocą bramki NAND dekoduje się stan 13. Jeśli wyjście tej bramki steruje wejściem zerującym (zakłada się, że wejście to jest wejściem o oddziaływaniu poziomem niskim), to licznik zostanie wyzerowany, gdy pojawi się stan 1101. Jeśli wejście *CLR* jest wejściem synchronicznym, to licznik taki jest licznikiem 14-stanowym. Jeśli natomiast wejście *CLR* jest wejściem asynchronicznym, to stan 13 pojawi się tylko na krótko (opóźnienie licznika i bramki NAND) i w efekcie licznik ten będzie licznikiem 13-stanowym.

Zadania

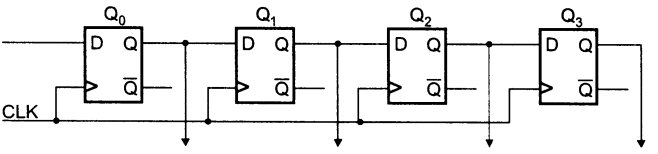
1. Zrealizować licznik liczący od 0 do 599 za pomocą liczników 16-stanowych.
2. Zaprojektować za pomocą 16-stanowego licznika licznik 8-stanowy liczący od stanu 0 do stanu 4 oraz od stanu 9 do 12. Wykonać różne projekty w zależności od rodzaju wejść *LD* i *CLR*. ⊠

4.4.2. Rejestry

Rejestr jest to układ składający się z przerzutników zwykle wspólnie sterowanych sygnałem zegarowym i wykonujących podobne funkcje. Na przykład są one jednocześnie odczytywane lub zapisywane, jednocześnie zerowane lub ustawiane w stan 1. Na rysunku 4.63 pokazano rejestr składający się z 4 przerzutników. Jest on wyposażony w 4 wejścia informacyjne D_i oraz 3 wejścia sterujące: wejście zegarowe *CLK*, wejście zerujące *CLR* i wejście *PRESET* ustawiające stan jeden wszystkich przerzutników. W takim rejestrze wpisu nowej zawartości, czyli stanów na wejściach D_i dokonuje się sygnałem zegarowym. Po podaniu na wejścia D_i jakiegoś słowa i po przyjściu impulsu zegarowego słowo to zostanie zapamiętane w tym rejestrze i pojawi się na jego wyjściach Q_i. Natomiast pozostałe wejścia sterujące, w zależności od typu rejestru, mogą zmieniać stan przerzutników w sposób synchroniczny z zegarem lub też asynchronicznie. Pokazany rejestr jest nazywany rejestrem równoległo-równoległym, gdyż zarówno odczyt, jak i zapis do niego odbywają się w sposób równoległy.

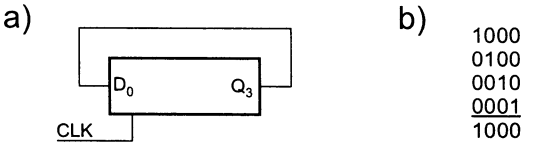


Rysunek 4.63. Rejestr 4-bitowy



Rysunek 4.64. Rejestr przesuwający 4-bitowy

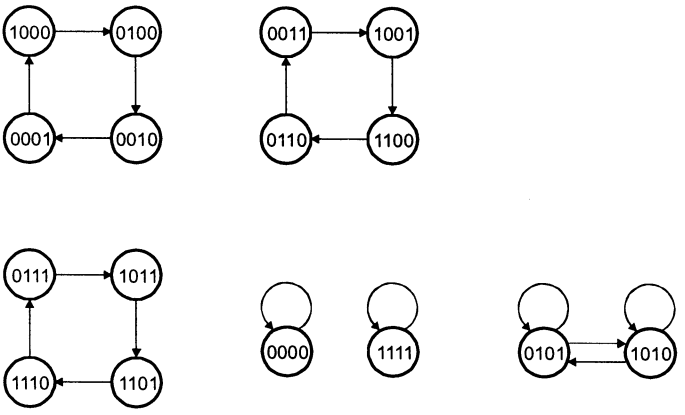
Często wykorzystuje się także inne rodzaje rejestrów. Jednym z nich jest rejestr, w którym wejścia i wyjścia przerzutników są połączone jak na rysunku 4.64. Pokazano tam kaskadowy sposób połączenia przerzutników, który tworzy układ nazywany **rejestrem przesuwającym** (ang. *shift register*). W pokazanym układzie jest jedno wejście informacyjne D_0, jedno wejście z generatora zegara *CLK* i cztery wyjścia Q_i. Taki rejestr nazywa się rejestrem szeregowo-równoległym, gdyż wprowadzanie informacji do niego odbywa się w sposób szeregowy, a wyprowadzanie informacji z rejestru ma charakter równoległy. Istnieją także rejestry równoległo-szeregowe i szeregowo-szeregowe. Rozpatrzmy działanie rejestru przesuwającego pokazanego na rysunku 4.64 przy założeniu, że pierwszy stan jest 0000 i na wejściu jest 1. Kolejne stany rejestru to: 1000, 1100, 1110 i 1111.



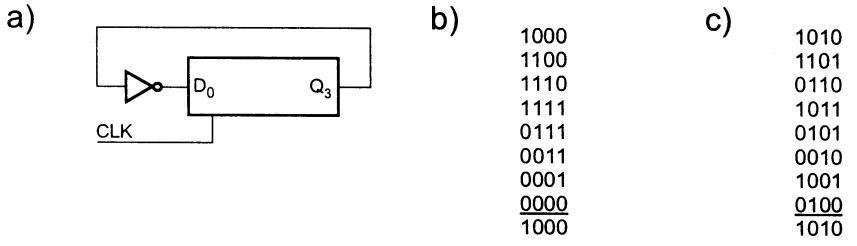
Rysunek 4.65. Licznik pierścieniowy (a) i sekwencja stanów (b)

Na rysunkach 4.65 – 4.67 przedstawiono działanie kilku układów sekwencyjnych zbudowanych z użyciem opisanego rejestru przesuwającego. Na rysunku 4.65a pokazano jeden z możliwych sposobów połączenia wejścia D_0 i wyjścia Q_3 rejestru przesuwającego. Przy założeniu, że pierwszym stanem przerzutników będzie stan 1000, układ będzie zmieniał swoje stany zgodnie z sekwencją pokazaną na rysunku 4.65b. Podczas ciągłego podawania sygnału zegarowego sekwencja tych czterech stanów powtarza się. Taki jednowejściowy układ sekwencyjny (przykładowy układ sekwencyjny ma tylko wejście zegarowe) może pełnić rolę licznika impulsów zegarowych. Każdy stan rejestru jest powtarzany po określonej liczbie impulsów wejściowych. Rozważany układ jest nazywany czterobitowym (czterostanowym) **licznikiem pierścieniowym**. W tym przypadku liczba przerzutników jest równa okresowi licznika. Zauważmy, że inaczej przebiega sekwencja stanów, jeśli rozpocznie się ona od innego stanu. Graf takiego układu jest pokazany na rysunku 4.66.

Na rysunku 4.67a pokazano układ ze sprzężeniem z wyjścia na wejście przez inwerter, a na rysunku 4.67b jego sekwencję stanów przy założeniu, że pierwszy stan jest 1000. Otrzymana sekwencja ma 8 stanów pokazanych na rysunku 4.67b. Tym razem czteroprzerzutnikowy rejestr przesuwający jest licznikiem do ośmiu. Taki licznik jest nazywany licznikiem w **kodzie Johnsona**. Natomiast jeśli stanem początkowym będzie stan 1010, to sekwencja stanów będzie obejmować także 8 stanów pokazanych na rysunku 4.67c.

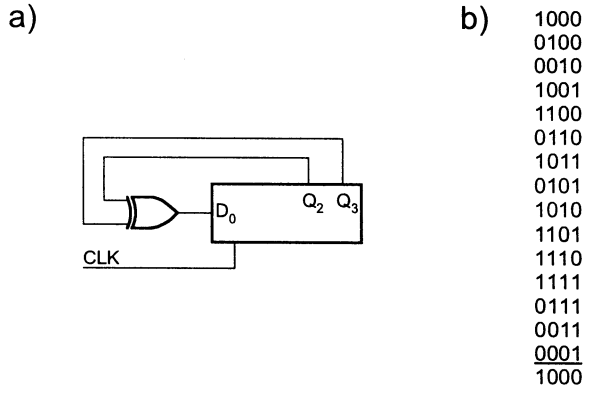


Rysunek 4.66. Graf przejść licznika pierścieniowego



Rysunek 4.67. Licznik w kodzie Johnsona

Jeszcze jeden układ pokazano na rysunku 4.68a i sekwencję stanów na rysunku 4.68b. Nosi on nazwę **licznika łańcuchowego**. Sprzężenie zwrotne uzyskano przez bramkę EXOR z wyjścia trzeciego i czwartego przerzutnika. Długość sekwencji stanów takiego licznika wynosi $2^n - 1$. W naszym przykładzie sekwencja stanów ma długość 15 i jedynie stan 0000



Rysunek 4.68. Licznik łańcuchowy

nie może do niej należeć, gdyż układ nie ma możliwości wyjścia z tego stanu. W przedstawionej na rysunku sekwencji stanów trudno doszukać się jakiejś regularności. Dlatego licznik łańcuchowy jest często wykorzystywany jako generator sekwencji tzw. pseudoprzypadkowej.

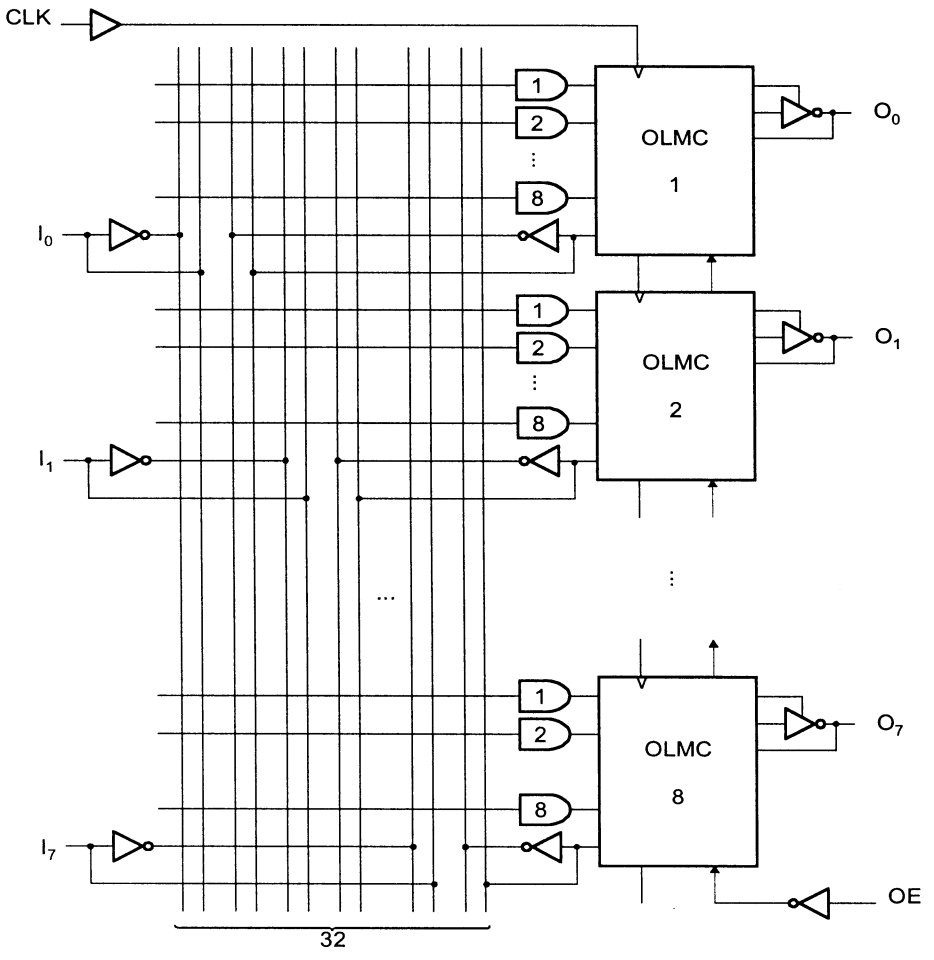
4.5. Realizacje automatów za pomocą układów LSI

Projektując złożone automaty składające się z dużej liczby bramek, przerzutników lub nawet liczników czy rejestrów, warto dążyć do realizacji w jak najmniejszej liczbie układów scalonych, a najlepiej w jednym układzie scalonym. Jeśli seria produkcyjna tych automatów jest bardzo długa, to projektuje się dla nich specjalny układ scalony typu *ASIC* (ang. *application specific integrated circuits*). Są to przykładowo automaty do samochodów, pralek, sprzętu audio/wideo itp. Proces projektowania takiego układu jest drogi, lecz ze względu na wielkość produkcji sam układ jest tani. Przeciwnie jest w przypadku, gdy automat produkowany jest w krótkich seriach lub w pojedynczych sztukach. Wtedy koszt opracowania nie może być wysoki, a koszt samego układu może być stosunkowo duży. W tym drugim przypadku wykorzystuje się programowane matryce logiczne, jak to wcześniej opisano w poprzednim rozdziale. Dalej przedstawiono dwa układy *PLD*, które służą do realizacji automatów. Przedstawiony będzie układ reprogramowalny typu *GAL* (ang. *generic array logic*) firmy *LATTICE* oraz układ *FPGA* (ang. *field programmable gate array*) firmy *ALTERA*. Następnie podane będą przykłady zastosowania języka *ABEL* do projektowania układów sekwencyjnych.

Programowane układy matrycowe stosowane do projektowania układów sekwencyjnych budowane są albo w podobny sposób jak układy kombinacyjne, a tylko matryce są uzupełniane przerzutnikami albo jako matryca komórek (w nowoczesnych układach nawet z pamięcią), których połączenia są programowane, a także programowana jest funkcja wykonywana przez samą komórkę.

Na rysunku 4.69 pokazano układ GAL 16V8. Jest to układ scalony o 20 końcówkach, który jest układem reprogramowanym, tj. może być wielokrotnie wykorzystywany do budowy różnych automatów. Układ ma 8 wejść dla zmiennych wejściowych oraz dwa wejścia sterujące: jedno *CLK* dla sygnału zegarowego taktującego wewnętrzne przerzutniki i drugie *OE* do sterowania wyjściami, tj. wprowadzania ich w stan trzeci. Układ składa się z dwóch części: programowanej matrycy AND oraz 8 programowanych komórek OLMC (ang. *output logic macrocell*) spełniających zaprogramowane funkcje. Programowana matryca AND składa się z 8 matryc (po jednej dla każdej komórki) po 8 wierszy (tyle można zrealizować iloczynów). Bramka sumy dla tych iloczynów znajduje się wewnątrz komórki OLMC. Matryca AND ma 32 kolumny: 16 dla wejść (zmienne wejściowe i ich negacje) i 16 dla wyjść przerzutników (sygnały wyjściowe i ich negacje) znajdujących się w OLMC.

Na rysunku 4.70 pokazano układ komórki OLMC. Układ ten składa się z przerzutnika sterowanego z bramki sumy oraz trzech multiplekserów sterujących sygnałami wyjściowymi. Konfiguracja OLMC może być zaprogramowana przez użytkownika specjalnym słowem konfiguracji przesyłanym do układu w czasie programowania go. Mapa programowania układu jest pokazana na rysunku 4.71. Do układu przesyła się 36 słów 64-bitowych, które mają różne znaczenie. Pod adresy $0-31$ przesyła się informację o umieszczeniu zwarć w matrycy AND. Pod adres 32 przesyła się 64-bitowe słowo o znaczeniu nadanym przez użytkownika (np. opis układu, określenie użytkownika, numer

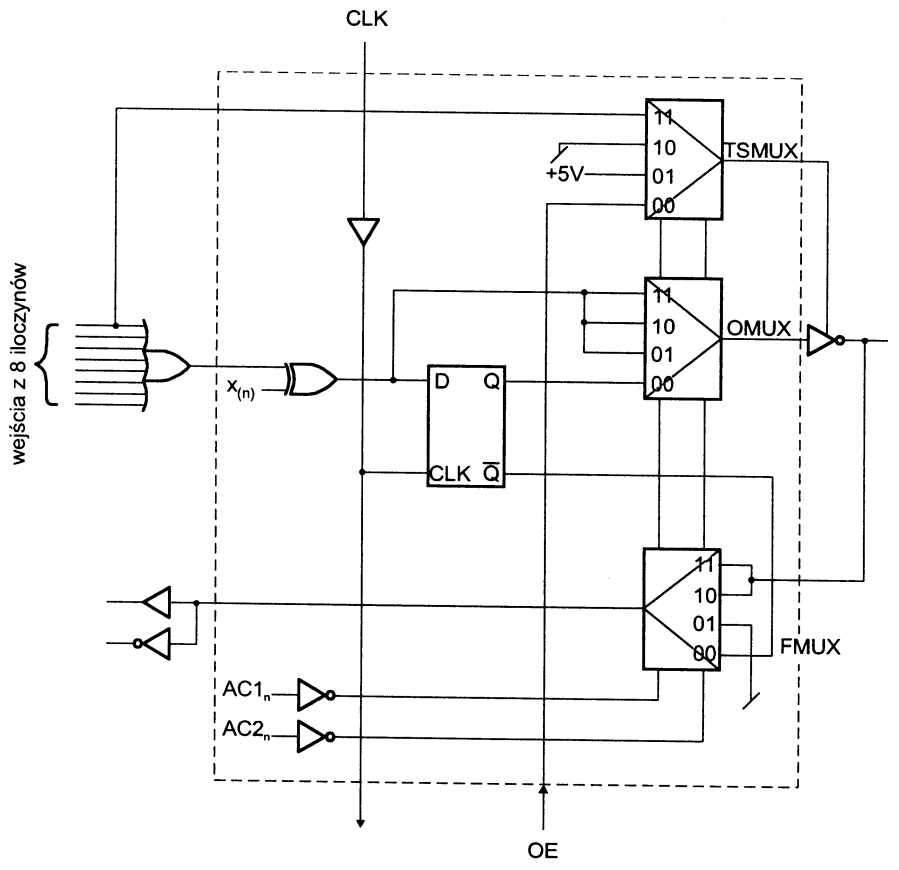


Rysunek 4.69. Schemat układu GAL 16V8

projektu itp.). Pod adres 60 (adresy 33 do 59 są zarezerwowane przez producenta na ewentualne rozszerzenia) przesyła się 82-bitowe słowo konfiguracji. Ostatnie 3 jednobitowe słowa to:

— bit ochrony uniemożliwiający odczyt danych z matrycy (ochrona zawartości przed kopiowaniem), który może być zmieniony jedynie przez ponowne programowanie matrycy,
— bit zarezerwowany przez producenta,
— bit blokowego wymazywania zawartości.

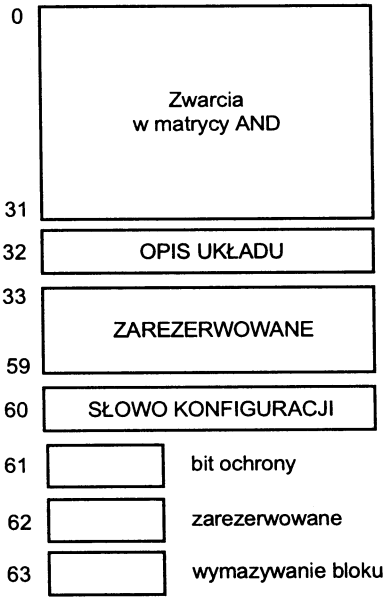
Słowo konfiguracji zawiera 64 bity dezaktywujące poszczególne kolumny matrycy AND, 8 bitów sygnałów $X(n)$ negujących lub nie funkcję OR sterującą przerzutnikami (celem realizacji postaci sumacyjnej lub iloczynowej), 8 bitów $AC1_n$ sterujących multiplekserami, bit $AC2_n$ sterujący multiplekserami wspólny dla wszystkich komórek oraz bit SYN ustawiający tryb pracy układu (układ może pracować w trybie kombinacyjnym lub w trybie rejestrowym).



Rysunek 4.70. Schemat komórki OLMC

Wyjściowa bramka trójstanowa jest sterowana z wyjścia multipleksera *TSMUX* (wejście *OE*). Wybór źródła aktywacji bramki następuje sygnałami $AC1_n$ i $AC2_n$. Dla kombinacji tych sygnałów 11 sygnał *OE* powstaje z matrycy AND (wybrane wejście 1 multipleksera *TSMUX*) a dla kombinacji tych sygnałów 00 sygnał *OE* jest wspólny dla wszystkich przerzutników i przychodzi z końcówki *OE*. Kombinacje te są używane w trybie rejestrowym. Dla kombinacji sygnałów $AC1_n$ i $AC2_n$ równych 01 sygnał *OE* jest równy 1 i bramka wyjściowa jest stale aktywna. Jest to kombinacyjny tryb pracy, dla którego dana końcówka jest wyjściem układu sterowanym z wyjścia bramki EXOR poprzez multiplekser *OMUX*. Dla kombinacji sygnałów 10 sygnał *OE* jest równy 0, co dezaktywuje bramkę wyjściową i dana końcówka nie jest używana. W tym przypadku multiplekser *FMUX* podaje na wejścia matrycy AND sygnał z sąsiedniego wyjścia. Dla skrajnych końcówek brany jest sygnał *OE* albo sygnał zegarowy. W innych przypadkach multiplekser *FMUX* podaje na wejścia matrycy AND stan przerzutnika lub stan na wyjściu bramki trójstanowej (w trybie rejestrowym).

W przedstawionym układzie można zaprogramować układ kombinacyjny składający się z 8 funkcji 8 zmiennych. Można też zaprogramować układ sekwencyjny do 256 stanów (8 przerzutników sterowanych z jednego wejścia *OE*) lub kilka układów sekwencyjnych,



Rysunek 4.71. Mapa programowania układu GAL 16V8

których wyjścia sterowane są oddzielnie. Zaprogramowanie układu wymaga użycia odpowiednich narzędzi sprzętowych (programator) oraz programowych, które umożliwią utworzenie odpowiedniego pliku dla programatora (format JEDEC) na podstawie odpowiedniego opisu układu. Ponieważ dotyczy to opisów układów sekwencyjnych, a więc wspomaganie programowe powinno akceptować formalne opisy tych układów, więc np. tablice przejść i wyjść automatów. Zostanie to przedstawione na przykładach, a językiem opisu będzie przedstawiony już wcześniej język ABEL.

Przykład 4.11. Zaprojektować automat realizujący daną tablicę przejść i wyjść podaną w tablicy 4.29.

Rozwiązanie. W języku ABEL automat można opisać kilkoma sposobami. Do opisu funkcji przejść można użyć trzech konstrukcji:

1. `IF-THEN-ELSE`,
2. `GOTO`,
3. `CASE`.

Tablica 4.29. Tablica przejść i wyjść przykładowego automatu

s \x1 x0	00	01	11	10	
s_1	s_1	s_4	s_2	-	0
s_2	-	s_5	s_3	s_2	1
s_3	s_1	s_4	s_3	s_6	0
s_4	s_2	s_4	s_3	s_1	1
s_5	s_5	s_1	s_1	s_3	0
s_6	s_2	s_2	s_4	s_1	0

Funkcję przejść automatu asynchronicznego można opisać także za pomocą równań.

Zapiszmy zadaną tablicę przejść każdym z wymienionym sposobów. W pierwszej deklaratywnej części opisu jest wymieniona liczba przerzutników (Q_0 i Q_1), nazwany sygnał zegarowy oraz sygnały wejściowe i wyjściowe.

```
module  Automat_Moore;

title 'tablica 4.29';

   Automat device 'P16V8';

   Clk,X1,X0          pin 1,2,3;
   Q0,Q1,Q2           pin 12, 13, 14 istype 'reg';
   Y                  pin 16 istype 'com';

   Ck,X               =  .C., .X. ;

" kodowanie stanow i wyjsc

" state_register

   SY = [Q0,Q1,Q2];
   S1 = [0, 0, 0];
   S2 = [0, 0, 1];
   S3 = [0, 1, 0];
   S4 = [0, 1, 1];
   S5 = [1, 0, 0];
   S6 = [1, 0, 1];

equations
   SY.C = Clk;
   Y = !Q0 & !Q1 & Q2 + !Q0 & Q1 & Q2;

state_diagram [Q0,Q1,Q2]
   state S1:
      case
           (X1 == 0 & X0 == 0): S1;
           (X1 == 0 & X0 == 1): S4;
           (X1 == 1 & X0 == 1): S2;
           (X1 == 1 & X0 == 0): X;
      endcase;
   state S2:
      case (X1 == 0 & X0 == 0): X;
           (X1 == 0 & X0 == 1): S5;
           (X1 == 1 & X0 == 1): S3;
           (X1 == 1 & X0 == 0): S2;
      endcase;
```

```
   state S3:
      case (X1 == 0 & X0 == 0): S1;
           (X1 == 0 & X0 == 1): S4;
           (X1 == 1 & X0 == 1): S3;
           (X1 == 1 & X0 == 0): S6;
      endcase;
   state  S4:
      case (X1 == 0 & X0 == 0): S2;
           (X1 == 0 & X0 == 1): S4;
           (X1 == 1 & X0 == 1): S3;
           (X1 == 1 & X0 == 0): S1;
      endcase;
   state  S5:
      case (X1 == 0 & X0 == 0): S5;
           (X1 == 0 & X0 == 1): S1;
           (X1 == 1 & X0 == 1): S1;
           (X1 == 1 & X0 == 0): S3;
      endcase;
   state  S6:
      case (X1 == 0 & X0 == 0): S2;
           (X1 == 0 & X0 == 1): S2;
           (X1 == 1 & X0 == 1): S4;
           (X1 == 1 & X0 == 0): S1;
      endcase;



test_vectors 'Sprawdzenie stanow'


   ( [Clk , X1, X0 ] -> [Q0,Q1,Q2])

      [ Ck , 0 , 0  ] ->     S1 ;
      [ Ck , 0 , 1  ] ->     S4 ;
      [ Ck , 1 , 1  ] ->     S3 ;
      [ Ck , 1 , 0  ] ->     S6 ;

      [ Ck , 0 , 0  ] ->     S2 ;
      [ Ck , 0 , 1  ] ->     S5 ;
      [ Ck , 1 , 1  ] ->     S1 ;
      [ Ck , 1 , 1  ] ->     S2 ;

      [ Ck , 1 , 1  ] ->     S3 ;
      [ Ck , 0 , 1  ] ->     S4 ;
      [ Ck , 0 , 1  ] ->     S4 ;
      [ Ck , 0 , 0  ] ->     S2 ;
```

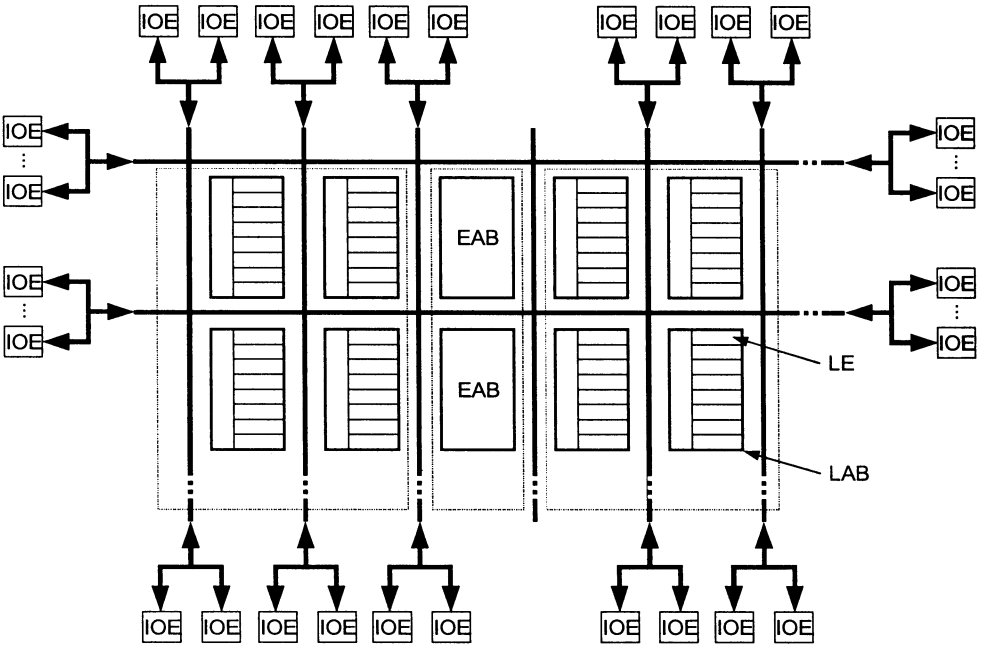
```
[ Ck , 1 , 0  ] ->     S2 ;
[ Ck , 0 , 1  ] ->     S5 ;
[ Ck , 0 , 0  ] ->     S5 ;
[ Ck , 0 , 1  ] ->     S1 ;

[ Ck , 1 , 1  ] ->     S2 ;
[ Ck , 1 , 1  ] ->     S3 ;
[ Ck , 1 , 1  ] ->     S3 ;
[ Ck , 0 , 0  ] ->     S1 ;

[ Ck , 1 , 1  ] ->     S2 ;
[ Ck , 0 , 1  ] ->     S5 ;
[ Ck , 1 , 0  ] ->     S1 ;
[ Ck , 1 , 0  ] ->     S3  ;

[ Ck , 1 , 0  ] ->     S6 ;
[ Ck , 1 , 0  ] ->     S1 ;
[ Ck , 1 , 1  ] ->     S2 ;
[ Ck , 1 , 1  ] ->     S3 ;

[ Ck , 1 , 0  ] ->     S6 ;
[ Ck , 1 , 1  ] ->     S4 ;
[ Ck , 1 , 1  ] ->     S3 ;
[ Ck , 1 , 0  ] ->     S6  ;

[ Ck , 0 , 1  ] ->     S2 ;
[ Ck , 1 , 1  ] ->     S3 ;
[ Ck , 0 , 1  ] ->     S4 ;
[ Ck , 1 , 0  ] ->     S1 ;

end
```

⊠

Inną grupę układów programowanych stanowią matryce programowanych komórek łączonych ze sobą za pomocą magistral (ang. *bus*), których połączenia są także programowane. Przykładem takiego układu może być układ FPGA (ang. *field programmable gate arrays*) firmy ALTERA o nazwie FLEX. Schemat układu FLEX 10k pokazano na rysunku 4.72. W układzie wyróżnić można 3 podstawowe elementy:

1) matryce tzw. wbudowanych bloków matryc logicznych LAB (ang. *logic array block*) i EAB (ang. *embedded array block*),
2) przecinające się magistrale połączeń (ang. *fast track interconnect*),
3) układy współpracy z wejściem-wyjściem.

Blok LAB składa się z 8 programowanych elementów logicznych LE (ang. *logic element*) zawierających przerzutnik oraz układy do realizacji jego funkcji wzbudzeń. Elementy te mogą być łączone za pomocą magistrali lokalnej (ang. *local interconnect*). Jeden blok LAB może realizować układ sekwencyjny zawierający 8 przerzutników.



Rysunek 4.72. Układ FLEX 10k

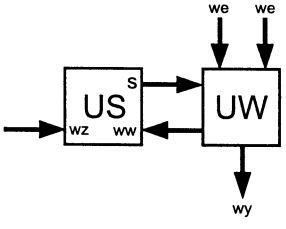
Odpowiada to jednemu układowi średniej skali integracji. W zależności od wykonania układy FLEX 10k zawierają w sumie ok. 4 tysięcy przerzutników. Dlatego można zbudować za jego pomocą układ odpowiadający ok. 500 układów scalonych średniej skali integracji. Wykorzystując matrycę EAB można zbudować pamięć RAM (ang. *random access memory*) — pamięć zarówno odczytywaną jak i zapisywaną, pamięć ROM (ang. *read only memory*), — pamięć tylko odczytywaną, tj. pamięć stałą, pamięć FIFO (ang. *first-in first-out*) lub inne. Wielkość pamięci to 2048 bitów konfigurowana w różny sposób, np. 256×8, 512×4, 1024×2 lub 2048×1.

Wyjścia bloków łączone są ze sobą poprzez programowane magistrale. Są magistrale wierszy (ang. *row interconnect*) i magistrale kolumn (ang. *column interconnect*). Oba typy magistral są dołączone do układów wejścia i wyjścia (*IOE*). W takim układzie daje się zaimplementować układ cyfrowy o złożoności mikroprocesora. Aby jednak opisać taki układ trzeba zastosować dosyć zaawansowane narzędzia umożliwiające przejście od opisu układu do odpowiednich połączeń. W tym celu w firmie ALTERA opracowano system wspomagania projektowania MAX+PLUS II używający języka AHDL (ang. *ALTERA high level description language*). Programowanie polega na składaniu schematu z gotowych elementów (standardowe układy MSI) na ekranie komputera. Zastosowany kompilator przetworzy rysunek struktury na opis programowania układu FLEX. System ma także swój symulator, edytor przebiegów czasowych a także analizator przebiegów czasowych. Projektant ma możność łatwego zmieniania projektu, dobudowywania dodatkowych bloków, łączenia ze sobą różnych bloków itp. Po cyklu projektowania i programowania projektant może otrzymać pełną dokumentację projektu.

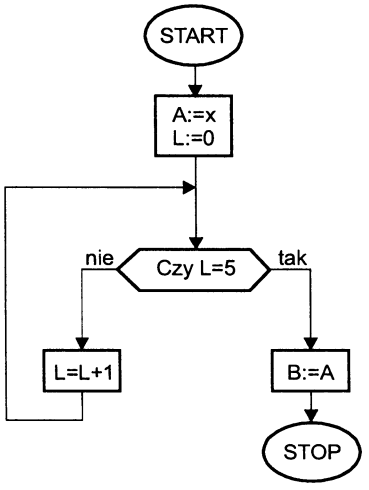
5 Układy mikroprogramowane

W niektórych układach cyfrowych można wyodrębnić wejścia i wyjścia informacyjne oraz wejścia i wyjścia sterujące. W zależności od stanu sygnałów na wejściach sterujących układ realizuje różne działania na słowach pojawiających się na wejściach informacyjnych. Wyniki tych działań wysyłane są na wyjścia informacyjne, a pewne cechy wyników (wartość określonego bitu, znak liczby, parzystość jedynek itp.) pojawiają się na wyjściach sterujących. Takie układy cyfrowe można projektować dekomponując je na dwa bloki: blok wykonawczy i blok sterujący. Układ taki jest pokazany na rysunku 5.1. Blok wykonawczy projektuje się tak, aby wykonywał elementarne działania, z których można składać bardziej złożone operacje. Przykładowo operację mnożenia można wykonać poprzez odpowiednią sekwencję działań elementarnych, tj. dodawań i przesuwań. Ta sekwencja działań elementarnych, tworząca program działania całego układu, jest generowana przez blok sterujący i podawana na wejścia sterujące bloku wykonawczego, wymusza odpowiednie działanie całego układu. Zadaniem układu sterującego jest pamiętanie sekwencji wysterowań bloku wykonawczego, a więc pamiętanie programu. Program korzysta z informacji generowanej przez blok wykonawczy i są to warunki wewnętrzne (na rysunku 5.1 sygnały *WW*) i z informacji przychodzącej do całego układu z zewnątrz i są to warunki zewnętrzne (na rysunku 5.1 sygnały *WZ*).

Przedstawione wyżej działania elementarne są nazywane **mikrooperacjami** (mikrorozkazami), a sekwencja wysterowań bloku wykonawczego pamiętana w bloku sterującym jest nazywana **mikroprogramem**. Ponieważ sekwencja wysterowań jest programem, to można przedstawić go za pomocą algorytmu realizującego odpowiednie zadanie. Dla ułatwienia przejścia od opisu słownego danego zadania do programu projektanci często posługują się siecią działań (ang. *flow-diagram*). Ponieważ rozkazy mogą być bezwarunkowe lub warunkowe, to sieć działań zawiera dwa rodzaje klatek: klatki bezwarunkowe i klatki warunkowe. Sekwencja wykonywania rozkazów jest prezentowana w sieci działań przez powiązania pomiędzy klatkami. Na rysunku 5.2 pokazano przykładową sieć działań układu przepisującego zawartość rejestru *A* do rejestru *B* po 5 taktach zegarowych. Po rozpoczęciu



Rysunek 5.1. Ogólna struktura układu mikroprogramowego



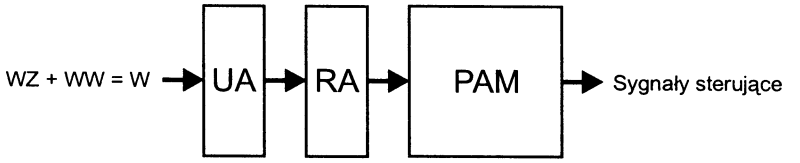
Rysunek 5.2. Przepisywanie zawartości z rejestru *A* dorejestru *B* po 5 impulsach zegarowych

programu jest ładowany rejestr *A* oraz jest zerowany licznik *L*. Następnie jest wykonywany rozkaz warunkowy skoku do rozkazu przepisującego zawartość rejestru *A* do *B*. Warunkiem wykonania tego rozkazu jest stan licznika *L*. Aby został on wykonany stan licznika musi wynosić 5. Jeśli warunek ten nie jest spełniony, to wykonywany jest rozkaz inkrementacji zawartości licznika.

Najprostszym układem cyfrowym umożliwiającym pamiętanie sekwencji słów sterujących jest pamięć. Pamięcią nazywa się matrycę złożoną z przerzutników, tj. elementów pamiętających jeden bit. Pewien podzbiór tych elementów (słowo) jest wybierany jednocześnie przez podanie adresu na odpowiednie wejście pamięci. Wtedy można odczytać lub zapisać dane słowo. Podstawowym parametrem pamięci jest jej pojemność. Jest to liczba pamiętanych słów *k*-bitowych. Przykładowo mówimy o pamięci o pojemności 128×32, a więc o pamięci o 128 słowach 32-bitowych. Liczba bitów w słowie nazywana jest długością słowa. W zastosowaniu do układów mikroprogramowanych taka pamięć może pamiętać 128 wysterowań bloku wykonawczego, który może mieć 32 wejścia sterujące. Pamięć o 128 słowach wymaga 7 bitów słowa adresowego. Odpowiednie kombinacje tych bitów powodują wybranie właściwych słów z pamięci. Problem kolejności odczytywania takiej pamięci, czyli problem sekwencji wysterowań bloku wykonawczego zależy od sposobu adresowania pamięci.

Na rysunku 5.3 pokazano uproszczoną strukturę układu sterowania, którego zasadniczą częścią jest pamięć i jej układ adresowania. Warunki zewnętrzne i warunki wewnętrzne (rys. 5.1) są podawane na wejście układu adresowania *UA*, który oblicza odpowiedni adres pamięci. Adres ten jest zapisywany do rejestru adresowego *RA*, aby był dostępny przez cały czas potrzebny do ukazania się i utrzymania pożądanego słowa na wyjściu pamięci. W tym czasie układ adresowania *UA* może obliczać nową wartość adresu. Informacja odczytana z pamięci zawiera słowo sterujące układem wykonawczym.

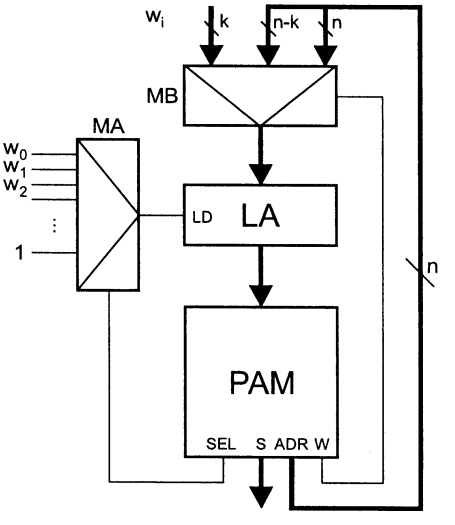
Ponieważ często kolejność wykonywanych mikrorozkazów jest zgodna z ułożoniem ich w pamięci, to jedną z funkcji układu adresowania jest wyliczenie następnika adresu znajdującego się w rejestrze adresowym. Ta kolejność pobierania rozkazów może być zmieniana



Rysunek 5.3. Uproszczona struktura układu sterującego

przez rozkaz skoku. Rozkaz skoku może być wykonany bezwarunkowo lub warunkowo. W tym drugim przypadku warunkami skoku są warunki wejściowe. Sygnały warunków mogą wpływać na ciąg wykonywania sekwencji rozkazów (kolejność następnikowa lub skok) przez odpowiednie adresowanie pamięci.

Na rysunku 5.4 pokazano strukturę układu sterującego. Adres pamięci jest pobierany z licznika adresowego *LA*. Licznik może wykonywać operację następnika, a może być ładowany sygnałami wejściowymi. Która z tych operacji ma zostać wykonana decyduje sygnał na jego wejściu *LD*. Sygnał ten powstaje na wyjściu multipleksera *MA*, na którego wejścia są podawane różne sygnały warunków oraz stałe 0 i 1. O tym który warunek ma w danym momencie być wykorzystany do wysterowania licznika adresowego musi decydować program. Dlatego część bitów (*SEL*) słowa wyjściowego z pamięci przeznaczono do wybrania odpowiedniego wejścia multipleksera, a więc rodzaju wysterowania licznika adresowego. Na wejściu multipleksera może zostać wybrany warunek w_i i jego wartość wskazuje, czy ma zostać wykonany skok czy też operacja następnika. Jeśli ma zostać wykonany skok, to rejestr adresowy *LA* ma być załadowany nową zawartością z wyjścia multipleksera *MB*. Mówimy, że został wykonany skok warunkowy a warunkiem była wartość w_i. Bitami *SEL* może zostać wybrane jedno z wejść multipleksera, na które podana jest odpowiednia wartość stała. Może ona wskazywać na wykonanie operacji następnika, a może wskazywać na



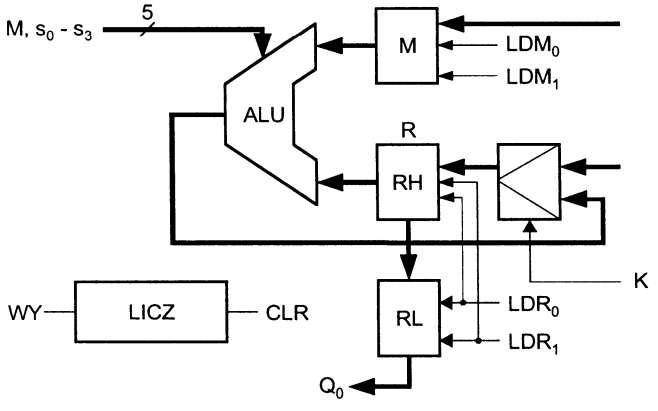
Rysunek 5.4. Struktura układu sterującego

ładowanie rejestru. Jeśli rejestr adresowy został załadowany nową zawartością, to wykonany będzie skok bezwarunkowy. Zwykle adres skoku jest podawany w programie, ale może być także zależny od warunków wejściowych. Dlatego na wejściach multipleksera *MB* muszą być zarówno sygnały warunków, jak i sygnały z samego programu, czyli z wyjścia pamięci. Na rysunku 5.4 pokazano, że jeden z bitów (*W*) słowa wyjściowego z pamięci jest przeznaczony do wskazania zależności adresu skoku od warunków. Jeśli adres skoku nie zależy od warunków, to na wyjściu multipleksera *MB* pojawia się *n* bitów z wyjścia *ADR* pamięci. Jeśli adres zależy od warunków, to na wyjściu multipleksera MB pojawi się tylko $n-k$ bitów z wyjścia *ADR* oraz *k* bitów z wejść warunków w_i.

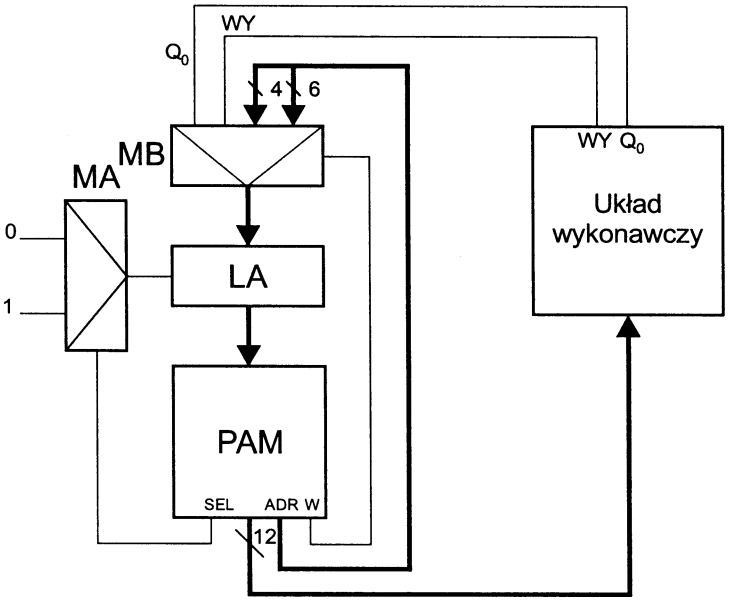
Ważnym problemem występującym przy projektowaniu mikroprogramowanych układów sterowania jest możliwość skrócenia części *ADR* słowa wyjściowego z pamięci. Wynika to z faktu, że bity składające się na część *ADR* są zajmowane we wszystkich słowach pamiętanych, a nie we wszystkich są wykorzystywane. Ze struktury pokazanej na rysunku 5.4 wynika, że długość części słowa wyjściowego *ADR* musi być odpowiednia do liczby słów pamięci. Jeśli pamięć ma 1024 słowa, to jej adres jest 10-bitowy i część słowa wyjściowego *ADR* musi mieć taką długość. Jednym z możliwych sposobów skrócenia słowa *ADR* jest ustalenie maksymalnej liczby adresów skoków. Oczywiście jest ona mniejsza niż liczba wszystkich adresów. Przykładowo jeśli w wyżej opisanej przykładowej pamięci liczba różnych adresów skoków wynosi 16, to można słowo *ADR* skrócić do 4 bitów i zaprojektować układ konwersji słowa 4-bitowego na adres 10-bitowy (często do budowy konwertera wykorzystuje się pamięć stałą — w naszym przykładzie o pojemności 16 słów 10-bitowych).

Drugim ważnym problemem jest automatyzacja tworzenia zawartości pamięci. Stosowane są różne systemy CAD pomagające utworzyć zawartość pamięci na podstawie danego algorytmu zapisanego w języku wysokiego poziomu. W podanym dalej przykładzie pokazano taki sposób.

Na rysunku 5.5 pokazano układ, omówiony w rozdziale 3, realizujący mnożenie dwóch liczb w kodzie NKB. Układ składa się dwóch rejestrów (jeden podwójnej długości), układu sumatora i licznika kroków. Algorytm polega na wykonaniu *n* kroków (dla *n*-bitowego mnożenia), a w każdym kroku wykonywany jest rozkaz warunkowego dodawania (warunkiem jest wartość najmniej znaczącego bitu) i rozkaz przesuwania. Algorytm rozpoczy-



Rysunek 5.5. Układ wykonujący mnożenie



Rysunek 5.6. Układ mnożenia wraz z układem sterowania

na się od załadowania rejestrów *M* i *R* (*RH* i *RL*). Na wejściu rejestru *R* jest multiplekser, gdyż są dwa źródła ładowania: jedno z wyjścia bloku *ALU* i drugie z wejścia układu. Źródło ładowania określa zewnętrzny bit *K*. Blok *ALU* ma 5 wejść sterujących, choć w podanym przykładzie wykonuje on tylko dodawanie (jeśli na wejściach sterujących jest 11111) lub operację pustą (założono kod 00000). Wejścia sterujące LDM_0 i LDM_1 oraz LDR_0 i LDR_1 odpowiednio dla rejestrów *M* i *R* określają jaka operacja ma zostać wykonana przez rejestr. Gdy $LDM_0 = 0$ i $LDM_1 = 0$, to jest operacja pusta (rejestr *M* nie jest ani ładowany ani przesuwany). Gdy $LDM_0 = 1$ i $LDM_1 = 0$, to rejestr *M* wykonuje operację przesuwania. Gdy $LDM_0 = 1$ i $LDM_1 = 1$, to rejestr *M* wykonuje operację ładowania.

W układzie jest także licznik kroków *LICZ*, który na wyjściu *WY* sygnalizuje osiągnięcie zadanej liczby kroków i który może być zerowany sygnałem z wejścia *CLR*. Słowo sterujące takim układem wykonawczym ma 12 bitów. W mikrooperacji założono także jeden bit *SEL* oraz 1 bit *W* sterujący wejściowym multiplekserem licznika adresowego *LA* (rys. 5.4) oraz 6 bitów adresowych *ADR*. Są dwa bity warunków: jeden z układu wykonawczego — najmniej znaczący bit rejestru *R* — Q_0 i drugi *WY* — końcowy stan licznika (w każdym kroku jest on inkrementowany).

Na rysunku 5.6 pokazano uproszczony schemat całego układu mikroprogramowanego (układ wykonawczy i układ sterujący), a na rysunku 5.7 przedstawiono mikroprogram mnożenia w przedstawionym układzie.

Załóżmy, że mikroprogram mnożenia jest wywoływany 6-bitowym adresem 000001. Pierwszy mikrorozkaz umieszczono pod adresem 000001. Wartości bitów tego mikrorozkazu wskazują, że rejestry *M* i *R* zostaną załadowane (na wejściach LDM_0 i LDM_1 oraz LDR_0 i LDR_1 są jedynki). Zostanie wyzerowany licznik *LICZ*, gdyż bit *CLR* jest jedynką. Licznik *LA* zostanie załadowany z multipleksera *MB* (na wyjściu multipleksera *MA* jest 0, gdyż *SEL* = 0). Bit *W* = 1 wskazuje na to, że do licznika *LA* zostanie załadowany adres z mikro-

ADRES	SEL	ALU	LDM_0	LDM_1	LDR_0	LDR_1	K	ADR	W	CLR
000001	0	00000	1	1	1	1	1	000000	1	1
000000	0	00000	0	0	0	0	0	00001x	0	0
000010	1	11111	0	0	1	1	0	xxxxxx	x	0
000011	0	00000	0	0	1	0	x	000x00	0	0
000100										

Rysunek 5.7. Mikroprogram mnożenia w układzie z rys. 5.6

rozkazu (bez warunków). Blok *ALU* wykona operację pustą. Bit *K* jest równy 1, co ustawia źródło ładowania rejestru *RH* na zewnętrzne. Bity adresowe *ADR* wskazują na adres następnego rozkazu 000000.

Pod adresem 000000 jest drugi mikrorozkaz. Jest to rozkaz sprawdzania wartości Q_0. W zależności od tej wartości będzie inny adres następnego rozkazu. Dlatego bit *W* równa się teraz 0. Jeśli $Q_0 = 0$, to nastąpi skok do adresu 000010, a jeśli $Q_0 = 1$, to do adresu 000011. Rejestry nie są ani ładowane, ani przesuwane, gdyż na wejściach LDM_0 i LDM_1 oraz LDR_0 i LDR_1 są bity 00. Blok *ALU* wykonuje operację NOP. Bit *SEL* jest równy 0, co powoduje, że licznik *LA* będzie ładowany z multipleksera *MB*.

Pod adresem 000010 jest mikrorozkaz dodawania. Na wejściu *SEL* jest 1, aby rejestr wykonał operację następnika i wskazał operację przesuwania. Na wejściach *ALU* jest kod dla operacji dodawania. Rejestr *M* nie jest ani ładowany, ani przesuwany, gdyż na wejściach LDM_0 i LDM_1 są bity 00, natomiast rejestr *R* jest ładowany (LDR_0 i LDR_1 są 11). Bit *K* jest równy 0, aby rejestr *R* był ładowany wynikiem dodawania. Bity adresowe w rozkazie i bit *W* mogą przyjmować wartości dowolne.

Pod adresem 000011 jest mikrorozkaz przesuwania. Na wejściu *SEL* jest 0, aby licznik *LA* był ładowany z multipleksera *MB*. Blok *ALU* wykonuje operację pustą. Rejestr *M* nie jest ani ładowany, ani przesuwany, gdyż na wejściach LDM_0 i LDM_1 są bity 00. Rejestr *R* jest przesuwany, gdyż na jego wejściach LDR_0 i LDR_1 są bity 10. Bit *K* przyjmuje wartość dowolną (rejestr nie jest ładowany). Adres w rozkazie zależy od wartości na wyjściu *WY* licznika. Jeśli $WY = 0$, to nastąpi skok do adresu 000000 (sprawdzanie wartości Q_0), a jeśli $WY = 1$, to nastąpi koniec algorytmu i program przejdzie do adresu 000100.

Pokazany program składa się z 4 mikrorozkazów umieszczonych pod adresami 0, 1, 2 i 3. Program zaczyna się od odczytania komórki nr 1 (adres 000001) i wykonania tego mikrorozkazu. Następnie jest wykonywany rozkaz z komórki pamięci nr 0, następnie albo z komórki nr 2 albo 3. Jeśli jest wykonywany rozkaz z komórki pamięci nr 2, to po nim wykonywany jest rozkaz z komórki o nr 3. Cykl powtarza się tyle razy, ile jest mnożonych bitów, a następnie (po osiągnięciu przez licznik stanu maksymalnego $WY = 1$) dla zakończenia programu skacze on do komórki 4 (000100).

Pokazany przykład jest dostatecznie prosty, aby pokazać na nim działanie mikroprogramu, ale jest zbyt uproszczony, aby przedstawić rzeczywiste mikroprogramy. W procesorach o sterowaniu mikroprogramowanym pamięć mikroprogramów zawiera kilka tysięcy słów o długości niekiedy ponad 100 bitów. Oczywiście wyznaczanie zawartości takiej pamięci odbywa się za pomocą komputerowych systemów wspomagających projektowanie CAD. Systemy takie także symulują działanie układów, przeprowadzają testy, umożliwiają modyfikację układów i modyfikację zawartości pamięci. Umiejętność posługiwania się systemami CAD jest dla projektantów warunkiem koniecznym przystąpienia do projektowania układów mikroprogramowanych.

Bibliografia

1. W. Traczyk: *Układy cyfrowe — podstawy teoretyczne i metody syntezy*. WNT 1982.
2. W. Majewski: *Układy logiczne*. WNT, Warszawa 1992.
3. P. Misiurewicz, M. Grzybek: *Półprzewodnikowe układy logiczne*. WNT, Warszawa 1975.
4. B. Wilkinson: *Układy cyfrowe*. WKŁ, Warszawa 2000.
5. J. Biernat: *Arytmetyka komputerów*. PWN, Warszawa 1996.
6. J. Kalisz: *Podstawy elektroniki cyfrowej*. WKŁ, Warszawa 1998.
7. W. Sasal: *Układy scalone — parametry i zastosowania*. WKŁ,Warszawa 1990.
8. M. Łakomy, J. Zabrodzki: *Cyfrowe układy scalone*. PWN, Warszawa 1983.
9. P. Prusinkiewicz, C. Stępień i inni: *Arytmetyka maszyn cyfrowych*. Skrypt PW, Warszawa 1978.
10. H. Kruszyński, A. Rydzewski, A. Śluzek: *Teoria układów cyfrowych*. Skrypt PW, Warszawa 1987.
11. St. Budkowski, A. Papliński, J. Sosnowski: *Zespoły i urządzenia cyfrowe*. WNT, Warszawa 1979.
12. W. Majewski, T. Łuba, K. Jasiński, B. Zbierzchowski: *Programowane moduły logiczne w syntezie układów cyfrowych*. WKŁ, Warszawa 1992.
13. T. Łuba, M.A. Markowski, B. Zbierzchowski: *Komputerowe projektowanie układów cyfrowych w strukturach PLD*. WKŁ, Warszawa 1993.
14. P. Misiurewicz, M. Perkowski: *Teoria automatów*. Skrypt PW, Warszawa 1976.

Skorowidz

P

R

S

T

U

W

Z

Wydawnictwa Komunikacji i Łączności Sp. z o.o.
Wydanie 1. Warszawa 2001.
Montaż, druk i oprawa z dostarczonych
diapozytywów
Nowa Drukarnia Wydawnicza S.A. w Krakowie
Zam. 239/01