ROZDZIAŁ DRUGI: **ALGEBRA BOOLE'A**

Obwody logiczne są podstawą nowoczesnych systemów komputerowych. Aby zrozumieć, jak system komputerowy posługuje się nimi, musisz zrozumieć logikę cyfrową i algebrę Boole'a. Ten rozdział wprowadza tylko podstawowe informacje na temat algebry Boole'a. Temat ten jest często tematem całych podręczników. W rozdziale tym, skupimy się na tych aspektach, które będą wsparciem przy czytaniu następnych rozdziałów.

2.0 WSTĘP

Logika boolowska stwarza podstawy dla wykonywania obliczeń w nowoczesnych, binarnych systemach komputerowych. Możemy przedstawiać różne algorytmy, lub różne elektroniczne obwody komputerowe używając systemu równań boolowskich. Ten rozdział dostarczy krótkiego wprowadzenia do algebry Boole'a, tablic prawdy, postaci kanonicznej, funkcji boolowskich, upraszczania boolowskich funkcji, projektów logicznych, kombinatoryki i obwodów sekwencyjnych, i równoważników hardware/software. Ten materiał jest szczególnie ważny dla tych ,którzy chcą projektować obwody elektroniczne lub pisać programy sterujące obwodami elektronicznymi. Nawet, jeśli nigdy nie planowałeś projektować hardware'u lub pisać programów nim sterujących, wprowadzenie do algebry boolowskiej, przedstawione w tym rozdziale jest jeszcze ważniejsze, ponieważ możesz używać takiej wiedzy do optymalizowania pewnych złożonych wyrażeń warunkowych, jak IF,WHILE i wielu innych wyrażeń. Sekcja o minimalizowaniu (optymalizowaniu) funkcji logicznych używa *Diagramów Veitch'a* lub *Map Karnaugh'a.* Technika optymalizacja to redukcja wielu warunków w funkcjach boolowskich. Musisz zdawać sobie sprawę, że wielu ludzi uważa tą technikę optymalizacji za przestarzałą ,ponieważ redukcja wielu warunków w równaniach nie jest już tak ważna jak była kiedyś. W tym rozdziale używamy metody mapowania jako przykład optymalizowania funkcji boolowskich, ale nie jako technikę stosowaną regularnie. Jeśli jesteś zainteresowany w projektowaniu obwodów i optymalizacji, musisz sięgnąć po teksty bardziej zaawansowane. Chociaż ten rozdział jest głównie zorientowany sprzętowo, przyjmij do wiadomości, że wiele pojęć w tym tekście będzie używało boolowskich równań (funkcji logicznych). Dlatego też powinieneś umieć sobie radzić z funkcjami boolowskimi przed kontynuowaniem dalszego czytania tej książki.

2.1 ALGEBRA BOOLE'A

Algebra Boole'a jest systemem matematycznym zamkniętym w granicach wartości zero i jeden (prawda lub fałsz). Operator binarny „° " określony przez ten zbiór wartości, przyjmuje parę wartości boolowskich jako dane wejściowe i zwraca pojedynczą wartość boolowską. Na przykład, boolowski operator AND, przyjmuje dwie wartości boolowskie na wejściu i zwraca na wyjściu pojedynczą wartość boolowską . Z danego systemu algebraicznego wynika kilka początkowych założeń ,lub aksjomatów, w jakim kierunku ten system pójdzie. Możesz wywnioskować dodatkowe zasady, twierdzenia, i inne właściwości systemu z tego zbioru podstawowych aksjomatów. System algebry boolowskiej często stosuje następujące aksjomaty:

∗ Zamknięcia. System boolowski jest zamknięty jeśli dla każdej pary wartości boolowskich, daje boolowski wynik. Na przykład, logiczne AND jest zamknięte w systemie boolowskim, ponieważ przyjmuje boolowskie operandy i daje tylko boolowskie wyniki.

∗ Przemienności. Mówimy, że operator binarny „° " jest przemienny jeśli A°B =B°A dla wszystkich możliwych wartości boolowskich A i B

∗ Łączności. Mówimy, że operator binarny „° " jest łączny jeśli (A°B) °C = A°(B°C) dla wszystkich wartości boolowskich A, B i C

∗ Rozdzielności. Dwa operatory binarne „°" i „%" są rozdzielne jeśli A°(B%C) = (A°B) % (A°C) dla wszystkich wartości boolowskich A, B i C

∗ Tożsamości. Mówimy, że wartość boolowska I jest elementem tożsamym w stosunku do operatora binarnego „°” jeśli A°I = A .
∗ Elementu odwrotnego. Mówimy, że boolowska wartość I jest elementem odwrotnym w stosunku do operatora binarnego „°′jeśli A°I=B a B jest wartością przeciwną do A w systemie boolowskim.
Dla naszych celów oprzemy algebrę boolowską na następującym zbiorze operatorów i wartości:
Dwie możliwe wartości w systemie boolowskim to zero i jeden. Często będziemy nazywać te wartości (odpowiednio) fałsz i prawda.
Symbol „• ” przedstawia logiczną operację AND; np. A•B jest wynikiem logicznego ANDowania boolowskich wartości A i B. Kiedy używamy pojedynczych liter w nazwach zmiennych, wyrzucamy symbol „•” Dlatego też, AB również przedstawia logiczny AND zmiennych A i B (będziemy to również nazywać iloczynem A i B).
Symbol „+” przedstawia logiczną operację OR; np. A+B jest wynikiem logicznego ORowania wartości boolowskich A i B (będziemy to również nazywać sumą A i B).
Logiczne dopełnienie, negacja lub nie, jest operatorem bez znakowym. W tym tekście będziemy używać
symbolu (‘) dla oznaczenia logicznej negacji .
Jeśli kilka różnych operatorów pojawia się w pojedynczym wyrażeniu boolowskim, wynik wyrażenia zależy od „pierwszeństwa” operatorów. Będziemy stosować następujące zasady pierwszeństwa (od najwyższej do najniższej) dla operatorów boolowskich: nawiasy, logiczne NOT, logiczne AND potem logiczne OR.
.Jeśli dwa operatory o tym samym pierwszeństwie są sąsiadujące, musisz oceniać je od lewej do prawej strony
Możemy także użyć następujących zbiorów postulatów:
P1 Algebra Boole’a jest zamknięta dla operacji AND, OR I NOT
P2 Tożsamość elementów, ze względu, na to że „•” reprezentuje jeden a „+” zero. Brak tożsamości elementów
dla operacji logicznej NOT .
P3. Operatory „•” i „+” są zamienne.
P4 • i + są rozdzielne względem siebie, tzn. A• (B+C) = (A•B)+(A•C) i A+(B•C)=(A+B) • (A+C).
P5 Dla każdej wartości A istnieje wartość A’ taka, że A•A’= 0 i A+A’=1. Ta wartość jest logicznym uzupełnieniem (albo NOT) wartości A.
P6 • i + oba są łączne. Tzn. (A•B) •C = A• (B•C) i (A+B)+C=A+(B+C).
Możemy udowodnić wszystkie inne twierdzenia w algebrze boolowskiej używając tych postulatów. Ten tekst nie będzie się zagłębiał w formalne dowodzenie tych twierdzeń, jednakże to dobra myśl aby zaznajomić się z kilkoma ważnymi teoriami w algebrze boolowskiej. Oto próbka:

Th1: A + A = A
Th2: A • A = A
Th3: A + 0 = A
Th4: A • 1 = A
Th5: A • 0 = 0
Th6: A + 1 = 1
Th7: (A+B)’ = A’ • B’
Th8: (A•B)' = A’ + B’
Th9: A + A•B = A
Th10: A•(A+B) = A
Th11: A +A’B = A + B
Th12: A’• (A+B’) =A’B’
Th13: AB + AB’ = A’
Th14: (A’+B’) • (A’+B) = A’
Th15: A + A’ = 1
Th16: A • A’ = 0

Twierdzenia siedem i osiem są nazywane Prawami DeMorgana, na cześć matematyka ,który je odkrył. Powyższe twierdzenia występują parami. Każda para (np. Th1 i Th2,Th3 i Th4) ma postać dualną. Najważniejszą zasadą w systemie algebry boolowskiej jest ta dualność. Każde ważne wyrażenie można stworzyć używając aksjomatów i twierdzeń algebry boolowskiej, korzystając z wymiany operatorów i stałych pojawiających się w wyrażeniu. Ściślej, jeśli wymieniamy operatory • i + i zamieniamy wartości 0 i 1 w wyrażeniu, otrzymujemy wyrażenie przestrzegające wszystkich zasad algebry boolowskiej. Nie znaczy to, że wyrażenia dualne obliczają takie same wartości., to tylko znaczy że oba wyrażenia są prawidłowe w systemie algebry boolowskiej. Mimo, że w tym tekście nie będziemy

udowadniać żadnych twierdzeń ze względu na algebrę boolowską, będziemy używać tych teorii dla pokazania ,że dwa boolowskie równania są identyczne. To jest ważna operacja, wtedy kiedy chcemy stworzyć postać kanoniczną wyrażenia boolowskiego lub kiedy upraszczamy wyrażenie boolowskie.

2.2 FUNKCJE BOOLOWSKIE I TABLICE PRAWDY

Wyrażenie boolowskie jest sekwencją zer, jedynek i literałów oddzielonych operatorami boolowskim. Literał jest .nazwą zmiennej. Dla naszych celów wszystkie nazwy zmiennych będą pojedynczymi znakami alfabetu. Funkcja boolowska jest określonym boolowskim wyrażeniem; zazwyczaj nadajemy funkcji boolowskiej literę „F" czasami z indeksem dolnym. Na przykład, rozpatrzmy następującą funkcję:

$F_0 = AB+C$

Ta funkcja oblicza logiczne AND z A i B a następnie logiczne OR z C. Jeśli A=1,B=0 a C=1 wtedy F_0 zwraca wartość jeden (1•0+1).

Innym sposobem przedstawienia funkcji boolowskiej jest tablica prawdy. W poprzedni rozdziale mieliśmy tablice prawdy przedstawiające funkcje AND i OR. Wyglądają następująco:

AND	0	1
0	0	0
1	0	1

Tablica 6: Tablica prawdy AND

OR	0	1
0	0	1
1	1	1

Tablica 7: Tablica prawdy OR

Dla operatorów binarnych i dwóch zmiennych wejściowych, taka forma tablic prawdy jest bardzo naturalna i dogodna. Jednak, rozpatrzmy jeszcze raz powyższa funkcję F_0 . Ta funkcja ma trzy zmienne wejściowe nie dwie ..Zatem nie możemy używać tablic prawdy w formie jaka jest przedstawiona powyżej. Na szczęście, jest bardzo łatwo zbudować tablice prawdy dla trzech lub więcej zmiennych. Poniższy przykład pokaże sposób zrobienia takiej tablicy dla funkcji dla trzech lub czterech zmiennych:

F =AB +C		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

Tablica 8: Tablica prawdy dla funkcji z trzema zmiennymi

F =AB +CD		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

Tablica 9: Tablica prawdy dla funkcji z czterema zmiennymi

W powyższych tablicach prawdy ,cztery kolumny przedstawiają cztery możliwe kombinacje zer i jedynek dla zmiennych A i B (B jest bardziej znaczącym bitem ,A jest mniej znaczącym bitem).Podobnie cztery kolumny w drugiej tablicy prawdy przedstawiają cztery możliwe kombinacje zer i jedynek dla zmiennych C i D.D jest bardziej znaczącym bitem a C mniej znaczącym bitem. Tablica 10 pokazuje inny sposób przedstawiania tablic prawdy. Ta forma jest łatwiejsza do wypełniania .Zauważ, że powyższe tablice prawdy uwzględniają wartości dla trzech

oddzielnych funkcji z trzema zmiennymi. Chociaż można stworzyć ogromny zbiór funkcji boolowskich, nie wszystkie będą unikalne. Na przykład, F=A i F=AA są dwiema różnymi funkcjami. Jednak według twierdzenia 2,łatwo pokazać że te dwie funkcje są równoważne ,tzn. przyniosą dokładnie takie same dane wyjściowe dla wszystkich kombinacji danych wejściowych. Jeśli określisz liczbę zmiennych wejściowych, otrzymasz skończoną liczbę unikalnych, możliwych funkcji boolowskich. Na przykład, jest tylko 16 unikalnych funkcji boolowskich przy dwóch danych wejściowych i tylko 256 możliwych funkcji boolowskich dla trzech danych wejściowych. Dla danych n zmiennych wejściowych, jest $2**(2^n)$ (dwa do potęgi 2) unikalnych funkcji boolowskich z tych n-zmiennych wejściowych. Dla dwóch zmiennych wejściowych mamy $2\^(2^2) = 2^4$ lub 16 różnych funkcji. Dla trzech wartości wejściowych mamy $2**(2^3=2^8$lub 256 możliwych funkcji. Dla czterech wartości wejściowych tworzymy $2**(2)^4$lub 2^{16} lub 65,536 możliwych unikalnych funkcji boolowskich.

C	B	A	F= ABC	F= AB+C	F=A+BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Tablica 10: Inny format tablicy prawdy

Kiedy mamy do czynieni z 16 funkcjami boolowskimi dosyć łatwo jest nazwać każdą funkcję .Poniższa tablica zawiera 16 możliwych funkcji boolowskich dla dwóch zmiennych wejściowych wraz z ich popularnymi nazwami i:

Funkcja #	Opis
0	Zero lub Czyszczenie. Zawsze zwraca zero bez wzgledu na wartosci wejsciowe A i B
1	Logiczne NOR (NOT (A OR B)) = (A+B)’
2	Inhibicja = BA’ (B not A). Równiez równowazne B > A lub A < B.
3	NOT A. Ignoruje B i zwraca A’
4	Inhibicja =AB’ (A not B). Równiez równowazne lub B<A
5	NOT B. Zwraca B’ i ignoruje A
6	Exclusive – or (XOR) = A ⊕ B. Równiez równowazne A ≠ B
7	Logiczne NAND (NOT (A AND B)) = (A • B)’
8	Logiczne AND = A • B. Zwraca A AND B
9	Równowaznosc = (A = B). Równiez znana jako exclusive-NOR (not exclusive-or)
10	Kopia B. Zwraca wartosc B i ignoruje wartosc A
11	Implikacja, B implikuje A = A+B’. (jesli B wtedy A). Równiez równowaznik B >=A
12	Kopia A. Zwraca wartosc A i ignoruje wartosc B
13	Implikacja , A implikuje B = B+A’ (jesli A wtedy B). Równiez równowaznik A >=B
14	Logiczne OR = A + B. Zwraca A OR B
15	Jeden lub ustawione. Zawsze zwraca jeden bez wzgledu na wartosci wejsciowe A i B

Tablica 11: 16 możliwych funkcji boolowskich dla dwóch zmiennych

f0- funkcja stała,
f1- funkcja NOR,
f2- funkcja implikacji (zakazu),
f3- negacja A,
f4- funkcja implikacji (zakazu),
f5- negacja B,

f_6- funkcja sumy wyłączającej, sumy modulo 2 lub funkcja EXOR,
f_7- funkcja NAND,
f_8- funkcja iloczynu,
f_9- funkcja równoważności,
f_{10}- funkcja tożsama ze zmienną,
f_{11}- funkcja implikacji,
f_{12}- funkcja tożsama ze zmienną,
f_{13}- funkcja implikacji,
f_{14}- funkcja sumy,
f_{15}- funkcja stała.
Odwołujemy się do numeru funkcji raczej niż do jej nazwy .Na przykład F_8 oznacza logiczne AND zmiennych A i B dla dwuwejściowej funkcji ,a F_{14} jest logiczną operacją OR. Tylko problemem jest ustalenie numeru funkcji. Na przykład dana jest funkcja z trzema zmiennymi F=AB+C, jaki jest jej odpowiedni numer? Ten numer jest łatwy do wyliczenia patrząc na tablicę prawdy dla funkcji (zobacz Tabela 14). Jeśli potraktujemy zmienne A,B i C jako bity w liczbie binarnej, z C jako najbardziej znaczącym bitem a A jako najmniej znaczącym bitem, stworzą one liczby binarne w zakresie od zera do siedmiu. Skojarzone z każdym z tych binarnych łańcuchów jest zero lub jeden jako wynik funkcji. Jeśli zbudujemy wartość binarną przez umieszczenie wyniku funkcji w miejscu określonym przez A,B i C to wartość końcowa liczby binarnej jest numerem funkcji. Rozpatrzmy tablicę prawdy dla F=AB+C:

CBA	7	6	5	4	3	2	1	0
F=AB+C	1	1	1	1	1	0	0	0

Jeśli potraktujemy wartość funkcji jako liczbę binarną ,otrzymamy wartość $F8_{16}$lub 248_{10}.Zwykle będziemy oznaczać numery funkcji w systemie dziesiętnym. To również pozwala zrozumieć dlaczego jest $2^{**}2^n$ różnych funkcji z n zmiennych: Jeśli mamy n zmiennych wejściowych, jest 2^n bitów w numerze funkcji. Jeśli mamy m bitów, jest 2^m różnych wartości .Dlatego też dla n wartości wejściowych mamy m.=2^n możliwych bitów i 2^n lub $2^{**}2^n$ możliwych funkcji.

2.3 ALGEBRAICZBNE DZIAŁANIA NA WYRAŻENIACH BOOLOWSKICH

Możemy przetworzyć jedno wyrażenie boolowskie na odpowiadające mu wyrażenie przez zastosowanie aksjomatów i twierdzeń algebry boolowskiej. Jest to ważne jeśli chcesz przekształcić dane wyrażenie do postaci kanonicznej (formy ujednoliconej) lub jeśli chcesz zminimalizować liczbę literałów lub warunki w wyrażeniu. Minimalizowanie warunków i wyrażeń może być ważne ponieważ obwody elektryczne często składają się z pojedynczych komponentów które implementują każdy warunek lub literał dla danego wyrażenia. Minimalizowanie wyrażenia pozwala projektantowi zużyć mniej elektrycznych komponentów i dlatego też może zredukować koszt systemu. Niestety, nie ma stałych zasad które pozwalają optymalizować dane wyrażenie. Podobnie jak budowa matematycznych dowodów ,indywidualne zdolności ułatwiają zrobienie tej transformacji. Niemniej jednak, można pokazać kilka przykładów ich możliwości:

ab + ab’ + a’b	=	a(b+b’) + a’b	przez P4
	=	a•1 + a’b	przez P5
	=	a + a’b	przez Th4
	=	a + a’b + 0	przez Th3
	=	a + a’b +aa’	przez P5
	=	a + b(a + a’)	przez P4
	=	a + b•1	przez P5
	=	a + b	przez Th4
(a’b +a’b’ + b’)’	=	(a’(b+b’) +b’)’	przez P4
	=	(a’ + b’)’	przez P5
	=	((ab)’)’	przez Th8
	=	ab	brak definicji
b(a+c) + ab’ +bc’ + c	=	ba + bc +ab’ +bc’ + c	przez P4
	=	a(b+b’)+b(c+c’) + c	przez P4
	=	a•1 + b•1 + c	przez P5
	=	a + b + c	przez Th4

Chociaż wszystkie te przykłady używają transformacji algebraicznych do upraszczania wyrażeń boolowskich, możemy również użyć operacji algebraicznych dla innych celów. Następna sekcja opisuje postacie kanoniczne wyrażeń boolowskich. Postać kanoniczna rzadko jest optymalna.

2.4 POSTAĆ KANONICZNA

Ponieważ jest skończona liczba funkcji boolowskich z n zmiennymi wejściowymi, mimo to jest skończona liczba możliwych wyrażeń logicznych dzięki którym możemy budować z tych n zmiennych wejściowych, nie mniej jest ogromna liczba wyrażeń logicznych które są odpowiednikami (tj. dają te same wyniki przy danych tych samych zmiennych wejściowych) To pozwala eliminować możliwe pomyłki, projektanci logiczni generalnie wyszczególniają funkcje boolowskie używane w formie kanonicznej lub ujednoliconej. Dla każdej danej funkcji boolowskiej istnieje unikalna postać kanoniczna. To eliminuje pomyłki kiedy pracujemy z funkcjami boolowskimi.
W rzeczywistości jest kilka różnych postaci kanonicznych. Będziemy tu omawiać tylko dwie i stosować tylko pierwszą z nich. Pierwsza jest nazywana **sumą pełnych iloczynów** a druga **iloczynem pełnych sum.** Używając zasady dualności ,jest bardzo łatwa konwersja między nimi. Warunek jest zmienną lub iloczynem (logiczne AND) kilku różnych .literałów. Na przykład, jeśli masz dwie zmienne A i B, istnieje osiem możliwych warunków: A,B,C,A',B',C',A'B',A'B,AB' i AB. Dla trzech zmiennych mamy 26 różnych wartości:
A,B,C,A',B',C',A'B',A'B,AB',AB,A'C',A'C,B'C',B'C,BC',BC,A'B'C',AB'C;,A'BC',ABC',A'B'C,AB' C,A'BC i ABC. Jak widzimy, jeśli liczba zmiennych wzrasta, liczba warunków wzrasta drastycznie. Implikant jest iloczynem zawierającym dokładnie n literałów. Na przykład, implikant dla dwóch zmiennych to A'B',AB',A'B i AB. Podobnie implikant dla trzech zmiennych A,B i C to:
A'B'C',AB'C',A'BC',ABC',A'B'C,AB'C,A'BC i ABC. Generalnie mamy 2^m implikantów dla m zmiennych. Zbiór możliwych implikantów jest bardzo prosty do stworzenia ponieważ one korespondują z porządkiem liczb binarnych

Odpowiednik binarny (CBA)	Implikant
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

Tablica 12: Implikanty dla trzech zmiennych wejściowych

Możemy wyspecyfikować każda funkcję boolowska używając sumy (logiczne OR) pełnych iloczynów. Danej funkcji F_{248} = AB+C odpowiada postać kanoniczna ABC+A'BC+AB'C+ABC'. Algebraiczne możemy pokazać te dwa odpowiedniki jako

ABC + A'BC+AB'C+A'B'C+ABC' = BC(A+ A') + B'C(A+A') + ABC'
= BC•1 +B'C•1 +ABC'
= C(B+B') + ABC'
= C+ ABC'
= C+ ab

Oczywiście postać kanoniczna nie jest forma optymalną, Z drugiej strony jest duża korzyść z sumy pełnych iloczynów postaci kanonicznej: jest bardzo łatwo stworzyć tablice prawdy dla funkcji w postaci kanonicznej. Co więcej jest również bardzo łatwo stworzyć logiczne równanie dla tabeli prawdy. Budowa tablicy prawdy z formy kanonicznej to prosta konwersja każdego implikantu wewnątrz wartości binarnej poprzez zastąpienie „1" dla. zmiennej „pozytywnej" j i „0" dla zmiennej „zanegowanej". Potem umieścić „1" na odpowiedniej pozycji (wyspecyfikowanej przez binarną wartość implikantu) w tabeli prawdy:
1) Konwersja implikantu do binarnego odpowiednika:
F_{248}=CBA+CBA'+CB'A+CB'A+C'BA
= 111+ 110+ 101+ 100+ 011
2) Zastępujemy jedynkę w tabeli prawdy dla każdej powyższej pozycji

C	B	A	F =AB+C

0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabela 13:Tworzenie tabeli prawdy dla implikantów ,Krok Pierwszy

Na końcu dodajemy zera do tych pozycji ,które nie są wypełnione jedynkami w kroku pierwszym

C	B	A	F =AB+C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabela 14: Tworzenie tabeli prawdy dla implikantów ,Krok Drugi

Generowanie funkcji logicznej z tabeli prawdy jest prawie równie łatwe .Po pierwsze, lokalizujemy wszystkie punkty zawierające jeden. W tabeli powyżej, jest to ostatnie pięć punktów. Liczby punktów w tabeli zawierają jedynki wskazujące liczbę implikantów w równaniu kanonicznym. Tworząc pojedynczego implikanta, zastępujemy A,B i C przez jedynki a A’,B’ i C’ zerami w tabeli prawdy .Potem obliczamy sumę tych punktów. W powyższym przykładzie,F_{248} zawierał jeden dla CBA=111,110,101,100,011.Zatem F_{248}=CBA+CBA’+CB’A+CB’A’+C’BA. Pierwszy warunek ,CBA pochodzi z ostatniego punktu w powyższej tabeli. C,B i A wszystkie zawierają jedynki więc tworzą implikant CBA (lub ABC jeśli wolisz)Drugi punkt zawiera 110 dla CBA więc tworzymy implikant CBA’. Podobnie 101 tworzy CB’A;100 tworzy CB’A i 011 tworzy C’BA. Oczywiście operacje logiczne OR i logiczne AND mogą przestawiać warunki wewnątrz implikantów jak sobie życzymy, i możemy przestawiać implikanty wewnątrz sumy jak zakładamy.

Ten proces pracuje równie dobrze z każda liczbą zmiennych. Rozważmy funkcję F_{53504} = ABCD+A’BCD+A’B’CD+A’B’C’D. Umiejscawiając jedynki na odpowiednich miejscach w tabeli prawdy otrzymujemy coś takiego

D	C	B	A	F=ABCD+A’BCD+A’B’CD+A’B’C’D’
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1

1	1	0	1	
1	1	1	0	1
1	1	1	1	

Tabela 15: Tworzenie tabeli prawdy dla czterech zmiennych z implikanta

Pozostałe elementy w tabeli prawdy zawierają zera. Być może najłatwiejszym sposobem stworzenia postaci kanonicznej funkcji boolowskiej jest po pierwsze wygenerowanie tabeli prawdy dla tej funkcji a potem budowanie postaci kanonicznej z tabeli prawdy .Będziemy używać tej techniki ,na przykład, będziemy konwertować między dwoma postaciami kanonicznymi przedstawionymi w tym rozdziale Jest również prostą sprawą generowanie sumy pełnych iloczynów .Algebraicznie, poprzez użycie przedstawionych praw i twierdzenia 15 (A+A'=1) zrobimy to zadanie łatwo. Rozważmy F_{248}=AB+C.Ta funkcja zawiera dwa warunki AB i C. Ale one nie są implikantami. Implikanty zawierają każdą z możliwych zmiennych w „pozytywnej" lub „zanegowanej" formie. Możemy skonwertować najpierw warunek do sumy pełnych iloczynów jak następuje:

AB	=	AB • 1	Th4
	=	AB • (C + C')	Th15
	=	ABC + ABC'	prawo rozdzielności
	=	CBA + C'BA	prawo łączności

Podobnie możemy skonwertować drugi warunek w F_{248} do sumy pełnych iloczynów. Jak następuje:

C	=	C • 1	Th4
	=	C • (A + A')	Th15
	=	CA + CA'	prawo rozdzielności
	=	CA•1 +CA'•1	Th4
	=	CA•(B +B') +CA' • (B+ B')	Th4
	=	CAB+ CAB' =CA;B + CA'B'	prawo rozdzielności
	=	CBA + CBA'+CB'A +CB'A'	prawo łączności

Ostatni krok (przestawiania warunków) w tych dwóch konwersjach jest opcjonalny. Dostaliśmy finalną postać kanoniczną dla funkcji:

F_{248}	=	(CBA + C'BA) + (CBA + CBA' + CB'A + CB'A')
	=	CBA + CBA' + CB'A+ CB'A'+C'BA

Innym sposobem wygenerowania postaci kanonicznej jest użycie iloczynu pełnych sum. Implicent jest sumą (logiczne OR) wszystkich danych wejściowych, "pozytywnych" lub „zanegowanych" Na przykład, rozpatrzmy następującą funkcje logiczną G z trzema zmiennymi:

G=(A+B+C)*(A'+B+C)*(A+B'+C).

Jako postać sumy pełnych iloczynów jest dokładnie jednak iloczynu pełnych sum dla każdej możliwej funkcji logicznej. Oczywiście, dla każdego iloczynu pełnych sum jest odpowiednia suma pełnych iloczynów. Faktycznie, funkcja G, powyżej, jest odpowiednikiem:

F_{248}=CBA+CBA'+CB'A=CB'A'+C'BA-AB+C

Generowanie tablicy prawdy dla iloczynu pełnych sum nie jest dużo trudniejsze niż budowanie jej z sumy pełnych iloczynów. Używamy zasady dualności dla osiągnięcia tego. Pamiętaj, że zasada dualności mówi o wymianie AND na OR i zer na jedynki (i vice versa). Dlatego też dla zbudowania tabeli prawdy, musisz wymienić literały „pozytywne" i „negatywne" na przeciwne w G:

G=(A'+B'+C')*(A+B'+C')*(A'+B+C')

Następnym krokiem jest zamiana logicznego OR i AND. To da:

G=A'B'C'+AB'C'+A'BC'

Na koniec musisz wymienić wszystkie zera na jedynki. To znaczy, że musisz przechować zera w tablicy prawdy dla każdej z powyższej pozycji, a potem wypełnić resztę tablicy prawdy jedynkami To umiejscowi zera w punktach zerowych, jeden i dwa w tablicy prawdy. Wypełnienie pozostałych pozycji jedynkami da F_{248}. Możesz łatwo konwertować między tymi dwoma postaciami kanonicznymi przez generowanie tablic prawdy dla jednej postaci i cofając się z tablic prawdy stworzyć drugą postać. Na przykład, rozpatrzmy funkcję z dwoma zmiennymi F_7=A+B. Suma pełnych iloczynów to F_7 =A'B+AB'+AB Tablica prawdy przyjmuje formę:

F_7.	A	B
0	0	0
0	1	0
1	0	1
1	1	1

Tablica 16 F_7 (OR) tablica prawdy dla dwóch zmiennych

Cofając się, otrzymamy iloczyn pełnych sum, który ma zlokalizowane wszystkie pozycje zawierające zero. To jest punkt gdzie A i B równają się zero. To daje nam pierwszy krok G=A'B'. Jednak jeszcze musimy odwrócić wszystkie zmienne G=AB. Poprzez zasadę dualności musimy zamienić logiczne OR i logiczny AND zawierające G=A+B. To jest postać kanoniczna iloczynu pełnych sum. Ponieważ praca z iloczynem pełnych sum wygląda trochę inaczej niż z sumą pełnych iloczynów, w tym tekście generalnie używać będziemy sumy pełnych iloczynów. Ponadto suma pełnych iloczynów jest bardziej popularna w pracy z logika boolowska. Jednak spotkasz się z obiema postaciami kiedy będziesz studiował projektowanie logiczne.

2.5 UPROSZCZENIA FUNKCJI BOOLOWSKICH

Ponieważ jest nieskończenie wiele wariantów funkcji boolowskich dla n zmiennych, ale tylko skończona liczba unikalnych funkcji boolowskich z tych n zmiennych, możesz się zastanawiać, czy jest jakaś metoda która pozwoli uprościć daną boolowska funkcję do stworzenia formy optymalnej Oczywiście możesz zawsze użyć algebraicznej transformacji do stworzenia optymalnej postaci, ale użycie heurystyki nie zagwarantuje optymalnej transformacji. Są ,a jakże, metody które pozwolą zredukować daną boolowską funkcje do jej optymalnej postaci .W tym tekście będziemy stosować metodę map, zobacz inne teksty o projektowaniu logicznym jeśli chcesz poznać inne metody. Od kiedy dla każdej funkcji logicznej musi istnieć forma optymalna, możesz dziwić się dlaczego nie użyto postaci optymalnej dla postaci kanonicznej. Są dwa powody: po pierwsze, może być kilka optymalnych form. Nie gwarantują jednak, że będą unikalne. Po drugie łatwo jest konwertować pomiędzy postacią kanoniczną a tablicą prawdy. Używanie metody map do optymalizacji funkcji boolowskich jest praktyczne tylko dla funkcji z dwoma, trzema i czterema zmiennymi. Ostrożnie, możesz używać jej dla funkcji z pięcioma i sześcioma zmiennymi ale metoda map jest nie efektywna do użycia w tym miejscu Dla więcej niż sześciu zmiennych zastosowanie metody map nie byłoby mądrym posunięciem. Pierwszym krokiem w użyciu metody map jest zbudowanie dwuwymiarowej tablicy prawdy dla funkcji, zobacz rysunek 2.1

		A	
		0	1
B	0	B'A'	B'A
	1	BA'	BA

Tablica Prawdy dla dwóch zmiennych

		BA			
		00	01	11	10
C	0	CB'A'	C'B'A	C'AB	C'BA'
	1	CB'A'	CB'A	CAB	CBA'

Tablica Prawdy dla trzech zmiennych

		BA			
		00	01	11	10
DC	00	D'C'B'A'	D'C'B'A	D'C'AB	D'CBA'
	01	D'CB'A'	D'CB'A	D'CAB	D'CBA'
	11	DCB'A'	DCB'A	DCAB	DCBA'

10	DC’B’A’	DC’B’A	DC’AB	DC’BA’

Tablica prawdy dla czterech zmiennych

Rysunek 2.1 Dwu, trzy i cztero dwuwymiarowe mapy prawdy

Ostrzeżenie: bardzo uważnie patrz na te tablice prawdy. Nie używają takich samych form jakie pojawiały się wcześniej w tym rozdziale. W szczególności ciąg wartości wynosi 00,01,11,10 a nie 00,01,10,11.Jest to bardzo ważne! Jeśli organizujesz tablice prawdy według kolejności binarnej, optymalizacja metodą mapowania nie pracuje właściwie. Będziemy to nazywać mapą prawdy w odróżnieniu od standardowych tablic prawdy. Twoje funkcje boolowskie w postaci kanonicznej (suma pełnych iloczynów),wstawiają jedynki do każdego punktu mapy prawdy odpowiadającego implikantowi w funkcji. Miejsca zerowe gdziekolwiek indziej. Na przykład, rozważmy funkcje z trzema zmiennymi

F=C’B’A+C’BA’+C’BA+CB’A’+CB’A+CBA’+CBA. Rysunek 2.2 pokazuje mapę prawdy dla tej funkcji

	BA			
	00	01	10	11
C 0	0	1	1	1
C 1	1	1	1	1

F = C’B’A+C’BA’+C’BA+CB’A’+CB’A+CBA’+CBA

Rysunek 2.2 : Próbka mapy prawdy

Następnym krokiem jest narysowanie prostokątów wokół grup jedynek.. Prostokąty otaczające muszą mieć boki których długości są potęgami dwóch. Dla funkcji z trzema zmiennymi, prostokąty mogą mieć boki których długość wynosi jeden, dwa i cztery. Zbiór prostokątów narysowanych musi otaczać wszystkie komórki zawierające jedynki w mapie prawdy. Sztuką jest namalować wszystkie możliwe prostokąty chyba że prostokąt byłby zupełnie otoczony wewnątrz innego. Zauważ że prostokąty mogą zachodzić na siebie jeśli jeden nie otacza drugiego. W mapie prawdy na rysunku 2.2 są trzy takie prostokąty .Zobacz rysunek 2.3

	BA			
	00	01	10	11
C 0	0	1	1	1
C 1	1	1	1	1

Rysunek 2.3: Otaczanie prostokątami grup jedynek w mapie prawdy

Każdy prostokąt przedstawia warunek uproszczenia funkcji boolowskiej. Dlatego też uproszczenie funkcji boolowskiej będzie zawierało tylko trzy warunki. Zbudujemy każdy warunek używając procesu eliminacji. Wyeliminujemy każdą zmienną której „pozytywne” lub „negatywne” formy zawierają się wewnątrz prostokąta. Rozpatrzmy długi, chudy prostokąt powyżej który jest obecny w wierszu gdzie C=1.Ten prostokąt zawiera oba ,A i B, w „pozytywnej” lub „negatywnej” postaci. Dlatego też możemy wyeliminować A i B z warunku. Ponieważ prostokąt jest obecny w regionie C=1,prostokat przedstawia pojedynczy liberał C. Teraz rozpatrzmy kwadrat Ten kwadrat zawiera C,C’,B , B i A .Dlatego przedstawia pojedynczy warunek A Podobnie kwadrat z wykropkowaną linią zawiera C,C’,A,A’ i B Przedstawia on pojedynczy warunek B. Na koniec, funkcja jest przedstawiana przez trzy kwadraty .Zatem F=A+B+C. Nie musimy rozpatrywać kwadratów zawierających zera. Prawy brzeg mapy prawdy „owija się” wokół lewego brzegu (i vice-versa).Podobnie górny brzeg „owija się” wokół dolnego. To przedstawia dodatkowe możliwości kiedy otaczamy grupy jedynek w mapie. Rozpatrzmy funkcje boolowska F=C’B’A’+C’BA’+CB’A+CBA’. Rysunek 2,4 pokazuje mapę prawdy dla tej funkcji

	BA			
	00	01	10	11

C 0	1	0	0	1
1	1	0	0	1

F=C'B'A'+C'BA'+CB'A+CBA'

Rysunek 2.4: Tablica prawdy dla funkcji F=C'B'A'+C'BA'+CB'A+CBA'

Przy pierwszym z nią zetknięciu ,można pomyśleć, że są tu dwa możliwe prostokąty ,jak pokazuje rysunek 2.5.Ponieważ mapa prawdy jest stałym obiektem połączonym z prawej i lewej strony, możemy stworzyć, jeden prostokąt jak to pokazuje rysunek 2.6 Wiec jak? Dlaczego mamy martwić się jeśli mamy jeden prostokąt lub dwa w mapie prawdy? Odpowiedź: są większe prostokąty i możemy wyeliminować więcej warunków. Mniejsze prostokąty, mniej warunków pojawi się w końcowej funkcji boolowskiej. Na przykład, był przykład z generowaniem dwóch prostokątów funkcji z dwoma warunkami. Pierwszy prostokąt (na lewo) eliminuje zmienną C, pozostawiając A'B' jako jej warunki. Drugi prostokąt, na prawo, również eliminuje zmienną C pozostawiając warunek BA'. Dlatego ta mapa prawdy stworzy równanie F=A'B'+AB'. Wiemy, że nie jest to optymalne, zobacz twierdzenie 13.Teraz rozważmy drugą mapę prawdy. Mamy tu pojedynczy prostokąt, więc nasza funkcja boolowska będzie miała jeden warunek. Wyraźnie jest to bardziej optymalne niż równanie z dwoma warunkami. Ponieważ ten prostokąt zawiera i C i C' jak również B i B', tylko warunek z lewej to A'. Dlatego też, ta funkcja boolowska daje w wyniku F=A'. Są dwa przypadki kiedy mapa prawdy nie może być zastosowana właściwie: mapa prawdy zawiera same zera lub mapa prawdy zawiera same jedynki. Te dwa przypadki odpowiadają (odpowiednio) funkcji boolowskiej F=0 i F=1.Te funkcje są łatwe do stworzenia poprzez zbadanie mapy prawdy. Ważna rzecz jaka musisz zapamiętać kiedy optymalizujesz funkcję boolowska używając metody mapowania jest to, żebyś zawsze chciał wybierać największy prostokąt którego

	BA 00	01	10	11
C 0	1	0	0	1
1	1	0	0	1

Rysunek 2.5: Pierwsza próba otoczenia prostokątem jedynek

	BA 00	01	10	11
C 0	1	0	0	1
1	1	0	0	1

Rysunek 2.6: Prawidłowy prostokąt dla tej funkcji

długość jest potęgą dwójki. Musisz to zrobić nawet dla zachodzących na siebie prostokątów (chyba że jeden prostokąt zawiera inny). Rozważmy funkcje boolowska :F=CB'A'+C'BA'+CB'A'+C'AB+CBA'+CBA. Daje ona mapę prawdy pokazaną na rysunku 2.7 Pierwszą pokusa jest stworzenie jednego zbioru prostokątów założonych na rysunku 2.8 Jednakże prawidłowe mapowanie jest pokazane na rysunku 2.9.Wszystkie trzy mapingi tworzą funkcje boolowska z dwóch warunków. Najpierw są stworzone wyrażenia F=B+A'B' i F=AB+A' Trzecia forma F=B+A' Oczywiście ta ostatnia forma jest najbardziej optymalna niż dwie pozostałe (zobacz twierdzenia 11 i12)Dla funkcji z trzema zmiennymi, rozmiar prostokąta jest określony liczbą warunków ją reprezentujących:

∗ Prostokąt zawierający kwadrat przedstawiający implikant .Skojarzony implikant będzie miał trzy literały.

∗ Prostokąt otaczający dwa kwadraty zawierające jedynki przedstawia warunek zawierający dwa literały.

∗ Prostokąt otaczający cztery kwadraty zawierające jedynki przedstawia warunki zawierające pojedynczy literał.

∗ Prostokąt otaczający osiem kwadratów przedstawia funkcję F=1.

Mapy prawdy tworzone dla funkcji z czterema zmiennymi są nawet podstępniejsze Jest tak ponieważ jest dużo miejsc prostokątnych mogących ukrywać się wzdłuż brzegów. Rysunek 2.10 pokazuje kilka możliwych prostokątnych miejsc ukrycia.

BA

		00	01	10	11
C	0	1	0	1	1
	1	1	0	1	1

Rysunek 2.7: Mapa prawdy dla F=CB’A’+C’BA’+CB’A’+C’AB+CBA’+CBA

BA

		00	01	10	11
C	0	1	0	1	1
	1	1	0	1	1

BA

		00	01	10	11
C	0	1	0	1	1
	1	1	0	1	1

Rysunek 2.8: Oczywisty wybór prostokątów

BA

		00	01	10	11
C	0	1	0	1	1
	1	1	0	1	1

Rysunek 2.9: Prawidłowy zbiór prostokątów dla F=CB’A’+C’BA’+CB’A’+C’AB+CBA’+CBA

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				

10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

00	01	11	10

00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

	00	01	11	10
00				
01				
11				
10				

Rysunek 2.10: Wzory dla mapy prawdy 4x4

Ta lista wzorów nie obejmuje wszystkich z nich! Na przykład, te diagramy nie pokazują żadnego z prostokątów 1x2.Musisz ćwiczyć ostrożnie kiedy pracujesz z mapą dla czterech zmiennych ,zapewnić sobie wybór największej możliwej liczby prostokątów, zwłaszcza kiedy zachodzą na siebie. jest to szczególnie ważne kiedy następny prostokąt masz z brzegu mapy prawdy. Podobnie jak funkcje z trzema zmiennymi, rozmiar prostokątów w czterozmiennej mapie prawdy dyktuje liczba warunków ją reprezentująca.

∗ Prostokąt zawierający pojedynczy kwadrat przedstawia implikant.. Powiązany warunek ma cztery literały.
∗ Prostokąt otaczający dwa kwadraty zawierające jedynki przedstawia warunek zawierający trzy literały.
∗ Prostokąt otaczający cztery kwadraty zawierające jedynki przedstawia warunek zawierający dwa literały.
∗ Prostokąt otaczający osiem kwadratów zawierających jedynki przedstawia warunek z dwoma literałami.
∗ Prostokąt otaczający szesnaście kwadratów przedstawia funkcje F=1.

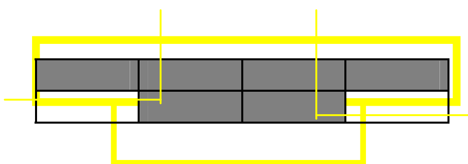
Ten poniższy przykład demonstruje optymalizację funkcji zawierającej cztery zmienne. Mamy funkcję:
F=D’C’B’A’+D’C’B’A+D’C’BA+D’C’BA’+D’CB’A+D’CBA+DCB’A+DCBA+DC’B’A+DC’BA’.Mapa prawdy jest na rysunku 2.11. Mamy tutaj dwa możliwe maksymalne zbiory prostokątów dla tej funkcji, każdy stwarzający trzy warunki (zobacz rysunek 2.12) Obie funkcje są sobie równoważne; obie są tak zoptymalizowane jak można było. Obie będą wystarczające dla naszych celów.

Najpierw rozważmy warunki przedstawiane przez prostokąt tworzony w czterech rogach. Te prostokąty zawierają B,B’,D i D’: więc możemy wyeliminować te warunki. Pozostałe warunki zawarte wewnątrz tego prostokąta to C’ i A’, wiec ten prostokąt przedstawia warunek C’A’. Drugi prostokąt, wspólny dla obu map prawdy z rysunku 2.12 jest prostokątem sformowanym przez cztery środkowe kwadraty .Ten prostokąt zawiera warunki A,B,B’,C,D i D’. Eliminujemy B,B’,D i D’ (ponieważ oba „pozytywne” i „negatywne” warunki istnieją) dostajemy CA jako warunek dla tego prostokąta. Mapa z lewej strony na rysunku 2.12, ma trzy warunki przedstawiane przez górny wiersz, Ten warunek zawiera zmienne A,A’,B,B’C’ i D’. Ponieważ zawiera A,A’,B i B’ możemy

BA

DC	00	01	11	10
00				
01				
11				
10				

Rysunek 2.11:Mapa prawdy dla:
F=D’C’B’A’+D’C’B’A+D’C’BA+D’C’BA’+D’CB’A+D’CBA+DCB’A+DCBA+DC’B’A+DC’BA’



Rysunek 2.12: Dwie kombinacje otaczania zmiennych zawierających trzy warunki.

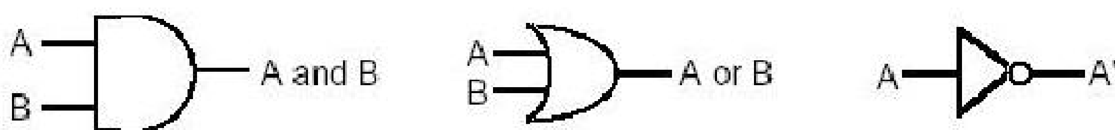
wyeliminować te warunki. Pozostaje nam C'D'. Dlatego też, funkcja przedstawiana przez mapę z lewej strony to F=C'A'+CA+C'D'. Mapa z prawej strony na rysunku 2.12 ma trzy warunki przedstawiane przez górne/środkowe kwadraty Ten prostokąt podsumowuje zmienne A,B,B',C,C' i D'. Możemy wyeliminować B,B' ,C i C' ponieważ oba „pozytywne' i „negatywne" wersje się pojawiają, pozostaje warunek AD. Dlatego tez funkcja przedstawiana przez funkcje z prawej strony to F=C'A'+CA+AD'. Ponieważ oba wyrażenia są sobie równoważne, zawierają ta samą liczbę warunków i tą samą liczbę operatorów, obie formy są równoważne. Chyba, że jest inny powód wybrania jednej z nich, można używać obu form.

2.6 JAKI TO MA ZWIĄZEK Z KOMPUTERAMI?

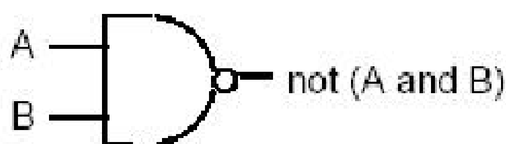
Chociaż istnieje słaby związek między funkcjami boolowskimi i wyrażeniami boolowskimi w językach programowania takich jak C lub Pascal, można się zastanawiać dlaczego spędziliśmy tak dużo czasu nad tym materiałem. Jednakże ,związki między logika boolowską i systemem komputerowym są bardzo silne. Jest to indywidualny związek między funkcjami boolowskimi a układami elektronicznymi. Inżynierowie którzy projektują CPU i inne pokrewne układy komputerowe muszą być gruntownie zaznajomieni z tymi rzeczami Nawet jeśli nie zamierzasz projektować swoich własnych obwodów elektronicznych, zrozumienie tych związków jest ważne jeśli chcesz pracować na systemach komputerowych.

2.6.1 RÓWNOWAŻNOŚĆ MIĘDZY UKŁADAMI ELEKTRONICZNYMI A FUNKCJAMI BOOLOWSKIMI

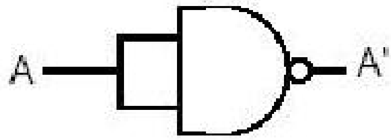
Jest to indywidualna równoważność między układami elektronicznymi a boolowskimi funkcjami. Dla każdej funkcji boolowskiej możesz zaprojektować układ elektroniczny i vice versa. Ponieważ funkcje boolowskie zawierają tylko boolowskie operatory AND,OR i NOT, możemy skonstruować każdy układ elektroniczny używając wyłącznie tych operacji. Boolowskie funkcje AND,OR i NOT odpowiadają następującym układom elektronicznym bramkom AND,OR i inwertorowi (NOT) (zobacz rysunek 2.13) Jednym interesującym faktem jest to ,że potrzebujesz tylko typu pojedynczej bramki do wprowadzenia w życie każdego układu elektronicznego. Tą bramka jest bramka NAND, pokazana na rysunku 2.14 Dowiedziemy, że można zbudować każdą funkcje boolowska używając tylko bramki NAND, musimy tylko pokazać jak zbudować inwerter (NOT),bramkę AND i bramkę OR z NAND (ponieważ tworzymy każdą boolowska funkcję używając tylko AND,NOT i OR).Budowanie inwertera jest łatwe, należy jedynie połączyć dwa wejścia razem (zobacz rysunek 2.15). Zaraz po tym jak zbudowaliśmy inwerter, budowa bramki AND jest łatwa – jedynie trzeba odwrócić wartości wyjściowe z bramki NAND. Przecież NOT (NOT(A AND B)) jest odpowiednikiem A AND B .Oczywiście, bierzemy dwie bramki NAND, do stworzenia pojedynczej bramki AND, ale nie można powiedzieć



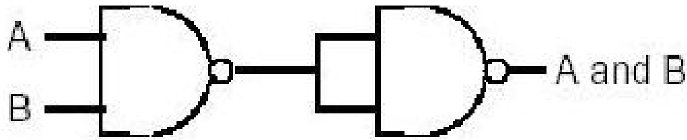
Rysunek 2.13: Bramki AND,OR i inwerter (NOT)



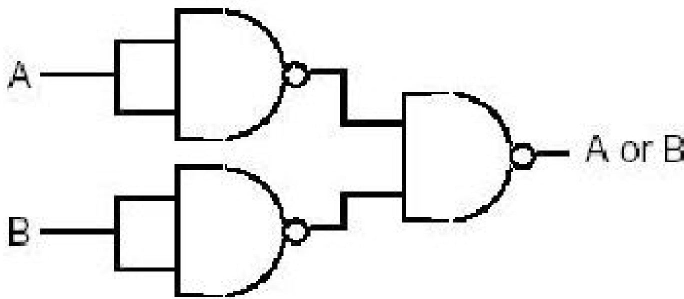
Rysunek 2.14: Bramka NAND



Rysunek 2.15: Inwerter zbudowany w oparciu o bramkę NAND



Rysunek 2.16: Konstruowanie bramki AND z dwóch bramek NAND



Rysunek 2.17:Konstruowanie bramki OR z bramek NAND

że obwody budowane tylko z bramek NAND będą optymalne, tylko to jest to możliwe do zrobienia. Możemy łatwo skonstruować bramki OR z bramek NAND poprzez zastosowanie twierdzeń DeMorgana

(A or B)' = A' and B' Teoria DeMorgana

A or B = (A' and B') Odwracanie obu stron równania

A or B = A' nand B' Definicja operacji NAND

Poprzez zastosowanie tych transformacji, otrzymamy układ z rysunku 2.17. Teraz możemy być zdziwieni dlaczego zawracaliśmy sobie tym głowę. W końcu ,dlaczego nie używać logicznego AND,OR i inwersji bezpośrednio? Są dwa powody. Po pierwsze bramki NAND generalnie są mniej kosztowne do zbudowania niż bramki pozostałe. Po drugie, jest również dużo łatwiej rozwijać złożone układy scalone z tych samych podstawowych bloków niż konstruować układy scalone używając różnych bramek podstawowych. Zauważ, nawiasem mówiąc ,że jest możliwe zbudowanie każdego logicznego układu używając tylko bramek NOR Równoważność między logicznym NAND i NOR jest ortogonalna do równoważności między dwoma postaciami kanonicznymi zawartymi w tym rozdziale (suma pełnych iloczynów i iloczyn pełnych sum).Podczas gdy logiczne NOR jest przydatne dla wielu układów, większość projektów elektronicznych używa logicznego NAND

2.6.2 UKŁADY KOMBINACYJNE

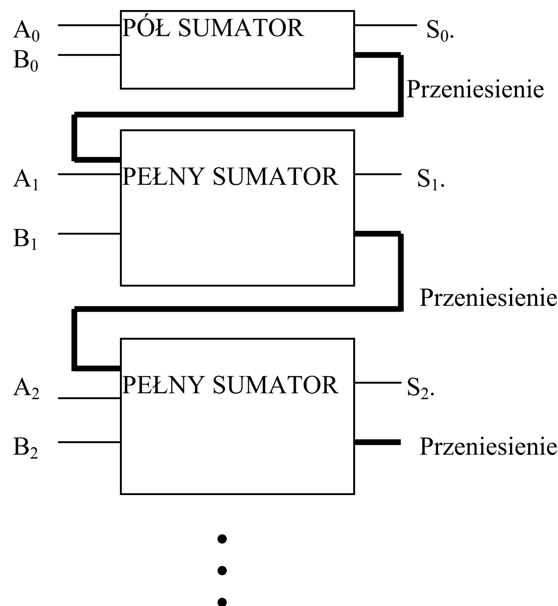
Układ kombinacyjny jest układem zawierającym podstawowe operacje boolowskie (AND,OR,NOT),kilka danych wejściowych i zbiór danych wyjściowych. Ponieważ każda dana wyjściowa odpowiada pojedynczej funkcji logicznej ,układ kombinacyjny często oferuje kilka różnych funkcji boolowskich. Ważnym jest abyś zapamiętał ten fakt - każda dana wyjściowa reprezentuje różne funkcje boolowskie. CPU komputera zbudowane jest z różnych układów kombinacyjnych. Na przykład możesz wprowadzić dodatkowy układ używający funkcji boolowskich. Przypuśćmy, że masz dwie jednobitowe liczby A i B .Możesz stworzyć jednobitową sumę i jednobitowe przeniesienie z tego dodawania używając dwóch funkcji boolowskich:

S=AB'+A'B Suma A i B

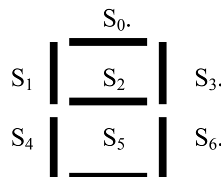
C=AB Przeniesie z dodawania A i B

Te dwie funkcje boolowskie określają „pół sumatorem". Inżynierowie nazywają to „pół sumatorem" ponieważ dodajemy dwa bity razem ale nie można dodać przeniesienia z poprzedniej operacji. Sumator pełny dodaje trzy jedno bitowe wartości wejściowe (dwa bity plus przeniesienie z poprzedniego dodawania) i stwarza dwie wartości wyjściowe, sumę i przeniesienie. Dwa logiczne równania dla „pełnego sumatora" to

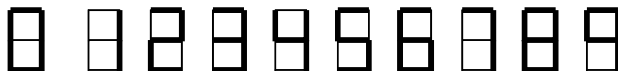
$S = A'B'C_{in}+A'BC'_{in}+AB'C_{in}'+ABC_{in}$
$C_{out} = AB+AC_{in}+BC_{in}$
Chociaż te logiczne równania tworzą tylko pojedyncze bity wyniku (ignorujemy przeniesienie),łatwo jest zbudować n-bitową sumę poprzez kombinacyjne dodawanie układów (zobacz rysunek 2.18).Ten przykład wyraźnie ilustruje ,że możemy użyć funkcji logicznych do wprowadzania operacji arytmetycznych i boolowskich.. Innym popularnym układem kombinacyjnym jest dekoder siedmiosegmentowy Jest to układ kombinacyjny który przyjmuje cztery dane wejściowe i ustala który z siedmiu segmentów na siedmiosegmentowym wyświetlaczu będzie załączony (logiczne jeden) lub wyłączony (logiczne zero).Ponieważ siedmiosegmentowy wyświetlacz zawiera siedem wartości wyjściowych (jedna dla każdego segmentu) będzie siedem logicznych funkcji stowarzyszonych z wyświetlaczem (od segmentu zero do sześć). Zobacz rysunek 2.19 Rysunek 2.20 pokazuje segmenty przydzielone dla każdej z dziesięciu wartości.
Cztery wartości wejściowe dla każdej z siedmiu funkcji boolowskich są czterema bitami z liczby binarnej z zakresu 0 do 9. D jest najbardziej znaczącym bitem tej liczby a A najmniej znaczącym .Każda funkcja logiczna tworzy jeden (segment załączony) dla danego wejścia jeśli określony segment będzie wyświetlony. Na przykład S_4 (segment cztery) będzie załączony (on)



Rysunek 2.18: Budowanie n-bitowych sumatorów przy użyciu „Pół i Pełnego sumatora”



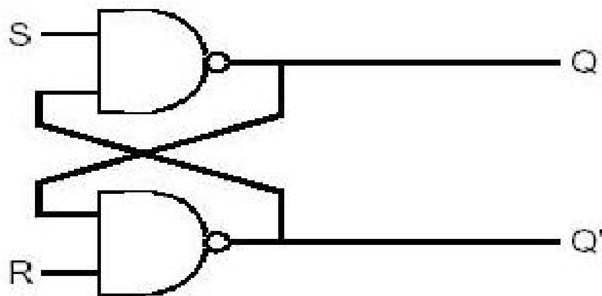
Rysunek 2.19: Siedmiosegmentowy wyświetlacz



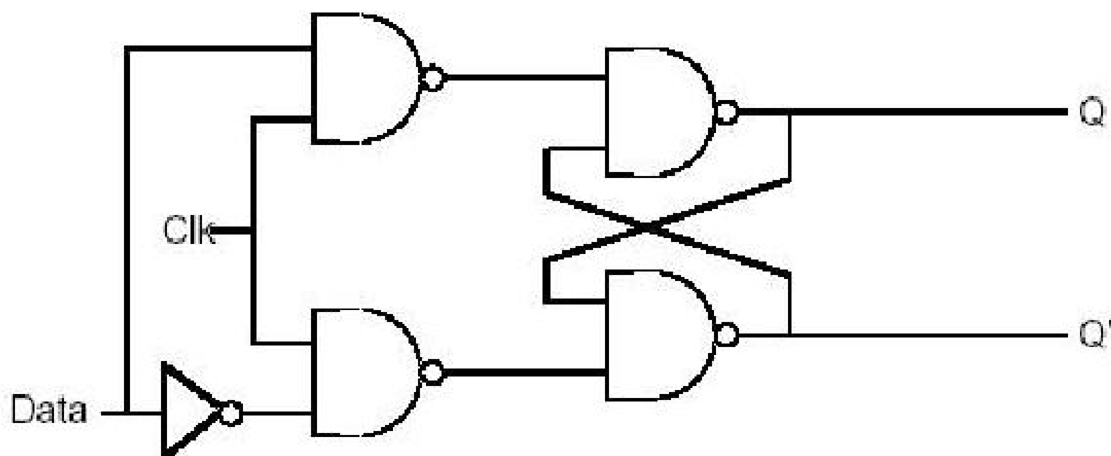
Rysunek 2.20 : Siedmiosegmentowe wartości od „0” do „9”
dla wartości binarnej 0000,0010,0110 i 1000.Dla każdej wartości wyświetla się segment będziemy mieć jeden implikant w logicznym równaniu:
$S_4= D'C'B'A+D'C'BA'+D'CBA'+DC'B'A'$
S_0 jako drugi przykład, jest włączony dla wartości zero, dwa, trzy pięć, sześć, siedem ,osiem i dziewięć. Dlatego też, logiczna funkcja dla S_0 jest :

S_0 = D'C'B'A'+D'C'BA'+D'C'BA+D'CBA'+D'CBA+DC'B'A'+DC'B'A
Możemy stworzyć inne pięć funkcji logicznych w podobny sposób. Układy kombinacyjne są podstawą dla wielu składników podstawowego systemu komputerowego. Możesz skonstruować układy dla dodawania, odejmowania, porównania, mnożenia, dzielenia i wielu innych operacji używających logiki kombinacyjnej.

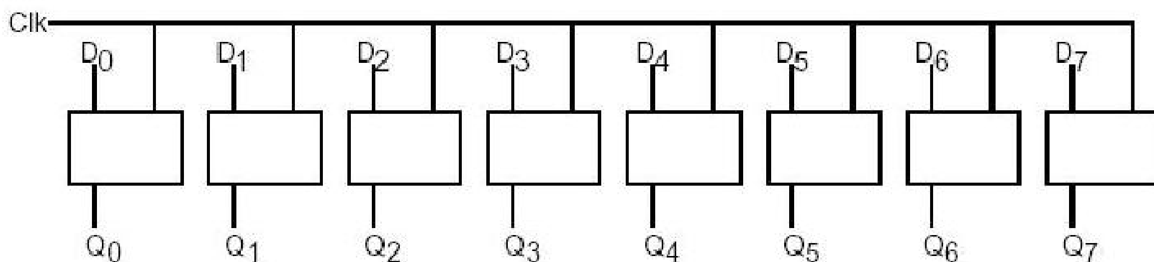
2.6.3 UKŁADY SEKWENCYJNE I LICZNIKI
W teorii, wszystkie logiczne funkcje wyjściowe zależą tylko od bieżących danych wejściowych. Każda zmiana wartości wejściowych natychmiast odbija się na danych wyjściowych. Niestety, komputery potrzebują zdolności zapamiętywania rezultatów poprzednich obliczeń. Jest to domena logiki sekwencyjnej i liczników. Komórka pamięci jest to układ elektroniczny który zapamiętuje wartości wejściowe po usunięciu tychże wartości. Najbardziej podstawową jednostką pamięci jest „przerzutnik SR" .Możesz zbudować przerzutnik SR używając dwóch bramek NAND jak pokazano na rysunku 2.21 Wartości wejściowe S i R są normalnie w stanie wysokim. Jeśli chwilowo ustawisz S na zero i zmienisz ją ponownie na jeden to wartość wyjściowa Q ustawi się na jeden. Podobnie jeśli przełączysz R z jeden na zero i ponownie na jeden to ustawisz Q na zero Q' jest ogólnie rzecz biorąc odwrotnością wyjściowego Q. Zauważ że jeśli oba ,S i R ,mają wartość jeden, wtedy Q jest zależne od Q'. To znaczy, cokolwiek zdarzyłoby się z Q, górny NAND kontynuować będzie przetwarzanie tej wartości. jeśli Q miało początkowo jeden ,tak więc są dwie jedynki jako dane wejściowe dla dolnego przerzutnika(Q nand R) dające na wyjściu zero (Q').



Rysunek 2.21:Przerzutnik SR zbudowany z bramek NAND
Dlatego też dwie wartości wejściowe na górnym NAND to zero i jeden. Podaje to wartość jeden na wyjście (odpowiadający oryginalnej wartości Q). Jeśli pierwotnie wartość Q była zero wtedy dane wejściowe dolnej bramki NAND to Q=0 i R=1.Dlatego też, dane wyjściowe tej bramki to jeden. Wartości wejściowe górnej bramki NAND to S=1 i Q'=1.To daje zero na wyjściu, pierwotna wartość Q. Przypuśćmy że Q=0,S=0 i R=0.Te ustawienia dwóch wartości wejściowych do przerzutnika to jeden i zero, ustawiając wyjście (Q) na jeden .Przywrócenie S do stanu wysokiego nie zmienia wcale wartości wyjściowej. Możemy uzyskać ten sam rezultat jeśli Q jest jeden, S zero a R jest jeden. Ponownie uzyskamy na wyjściu jeden. Ta wartość pozostanie jedynką nawet kiedy S przełączy się z zera na jeden .Dlatego też, przełączanie wejściowego S z jeden na zero i ponownie na jeden ,daje jeden na wyjściu (tj. ustawia przerzutnik) Ta sama zasada ma zastosowanie do wejścia R poza ustawieniem Q wyjściowego na zero zamiast jeden. Jest to jedno zastosowanie dla tego układu. Nie działa właściwie jeśli są ustawione oba wejścia S i R równocześnie. To ustawia oba wyjścia Q i Q' na jeden (co jest logiczną sprzecznością).którekolwiek wyjście pozostanie na dłużej zerem, ustali końcowy stan przerzutnika. Jeśli przerzutnik pracuje w tym trybie ,mówimy że jest niestabilny. Problemem z przerzutnikiem RS jest to ,iż musimy używać oddzielnie danych wejściowych do pamiętania zero lub jeden Komórka pamięci będzie bardziej wartościowa dla nas jeśli wyspecyfikujemy wartość danej do zapamiętania na jednym wejściu i dostarczymy sygnał zegarowy do zatrzasku wartości wejściowej. Ten typ przerzutnika (przerzutnik D) pokazano na rysunku 2.22.Zakladając,że ustaliłeś Q i Q' wyjściowe na wartości 0/1 lub 1/0,wysyłając impuls zegarowy ,z zera na jeden i zero, skopiujemy wejście D na wyjście Q. To również skopiuje D' na Q'. Chociaż zapamiętywanie pojedynczych bitów często jest ważne, w większości systemów komputerowych chcielibyśmy zapamiętywać grupy bitów. Możemy zapamiętywać sekwencje bitów poprzez połączenie kilku przerzutników D równolegle Połączenie przerzutników przechowujących n-bitową wartość tworzy rejestr. Elektroniczny schemat z rysunku 2.23 pokazuje jak zbudować ośmiobitowy rejestr ze zbioru przerzutników D.



Rysunek 2.22:Implementacja przerzutnika D z bramek NAND

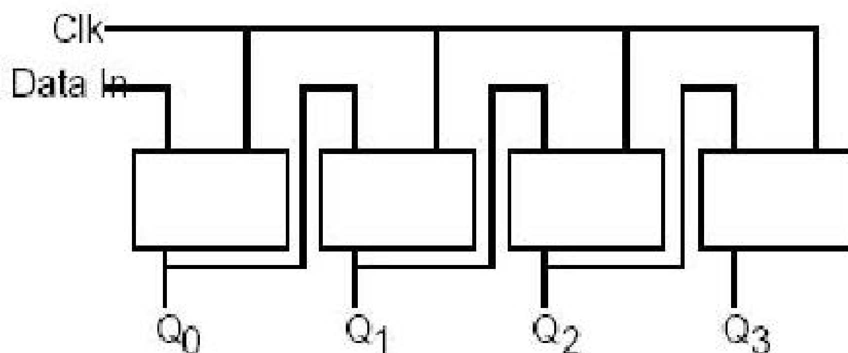


Rysunek 2.23: Ośmiobitowy rejestr stworzony z ośmiu przerzutników D

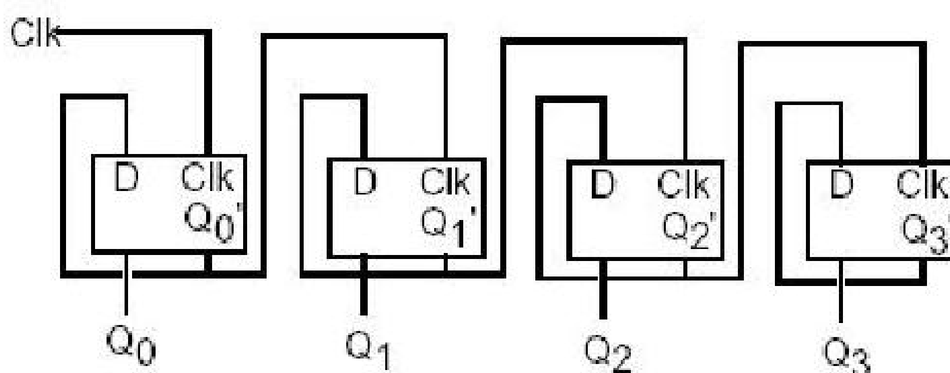
Zauważ że osiem przerzutników używa wspólnej linii zegarowej. Ten diagram nie pokazuje wyjścia Q' przerzutników ponieważ są one rzadko wymagane w rejestrze. Przerzutniki D są używane do budowania wielu układów sekwencyjnych nie tylko rejestrów Na przykład, możemy zbudować rejestr przesuwny który przesuwa bity z jednej pozycji na lewo przy każdym takcie zegarowym. czterobitowy rejestr przesuwny pokazano na rysunku 2.24 Można nawet zbudować licznik, który liczy wiele razy przełączanie zegara z zero do jeden i ponownie na zero używając przerzutników. Układ na rysunku 2.25 implementuje czterobitowy licznik używający przerzutnika. Możemy zbudować cały CPU z układów kombinacyjnych i tylko kilku dodatkowych układów sekwencyjnych poza tymi.

2.7 OKAY,ALE CO NAM TO DAJE PRZY PROGRAMOWANIU?

Gdy tylko mamy rejestry liczniki i rejestry przesuwne, możemy zbudować 'maszynę stanów" .Implementacja algorytmów w sprzętowym używaniu „maszyny stanu' wybiega poza zakres tego tekstu .Jednakże, jeden ważny punkt musi być powiedziany: algorytm można implementować w software, można również implementować bezpośrednio w sprzęcie Wskazuje to, że logika boolowska jest podstawą obliczeń na nowoczesnych systemach komputerowych. Każdy program który napiszesz, można wyszczególnić jako równania boolowskie. Oczywiście, jest dużo łatwiej wyszczególnić rozwiązanie problemu programistycznego używając takich języków jak Pascal, C lub nawet asembler, niż używając równań boolowskich,. Zatem niepodobnym jest abyśmy zawsze wyszczególniali cały program jako zbiór układów logicznych. Niemniej jednak jest czas kiedy implementacja sprzętowa jest lepsza. Rozwiązanie sprzętowe może być jedno ,dwa, trzy lub więcej razy szybsze niż analogiczne rozwiązanie programowe. Dlatego też czasami krytyczne operacje mogą wymagać rozwiązań sprzętowych. Bardziej interesującym faktem jest to, że odwrotność tych powyższych spraw jest także prawdą. Nie tylko można implementować wszystkie funkcje programowe w hardware ale jest również możliwa implementacja wszystkich funkcji sprzętowych w programie. Jest to ważne ponieważ wiele operacji ,które zazwyczaj są implementowane w sprzęcie są dużo tańsze do implementacji używając programów w mikroprocesorze. Istotnie, jest to podstawowe zastosowanie języka asemblera w nowoczesnym systemie jako



Rysunek 2.24 Czterobitowy: rejestr przesuwny zbudowany z przerzutników D



Rysunek 2.25 : Czterobitowy licznik zbudowany z przerzutników D

niedrogi zamiennik złożonych układów elektronicznych. Często jest możliwa zamiana dziesięcio – lub studolarowych elektronicznych komponentów na pojedynczy 25 dolarowy chip mikrokomputerowy. Cała grupa „systemów wbudowanych" uporała się z tym problemem. "Systemy wbudowane" są systemami komputerowymi osadzonymi w innych produktach. Na przykład większość kuchenek mikrofalowych, odbiorników TV, gier wideo, odtwarzaczy CD i innych urządzeń konsumenckich zawiera jeden lub więcej kompletnych systemów komputerowych których jedynym celem jest zastępowanie złożonych projektów sprzętowych. Inżynierowie używają komputerów do tego celu ponieważ są mniej kosztowne i łatwiejsze do zaprojektowania niż tradycyjne układy elektroniczne. Łatwo możemy stworzyć program do odczytania stanu przełączników (zmienne wejściowe) i przełączać silnik, LED'y lub światło zamykać lub otwierać drzwi itp. (funkcje wyjściowe).Do napisania takiego programu będziemy potrzebować zrozumienia funkcji boolowskich i tego jak zaimplementować taką funkcję w programie. Oczywiście jest drugi powód do studiowania funkcji boolowskich nawet jeśli nigdy nie zamierzasz pisać programów przeznaczonych dla „systemów wbudowanych" lub pisać programy dla urządzeń rzeczywistych. Wiele języków wysokiego poziomu przetwarza wyrażenia boolowskie (np. te wyrażenia które sterują wyrażeniem. if lub pętlą while).Przez zastosowanie transformacji takich jak twierdzenia DeMorgana lub optymalizacji mapowej jest często możliwe poprawienie wyników kodu języków wysokiego poziomu. Dlatego, studiowanie funkcji boolowskich jest ważne nawet jeśli nigdy nie zamierzasz projektować układów elektronicznych. Może ci to pomóc napisać lepszy kod w tradycyjnym języku programowania. Na przykład, przypuśćmy, że masz następujące wyrażenie w Pascalu:

If ((x=y) and (a<>b)) or ((x=y) and (c<=d)) then SomeStmt;

Możemy użyć rozpowszechnionych praw do uproszczenia tego do:

If ((x=y) and (a<>b) or (c<=d)) then SomeStmt;

Podobnie możemy użyć twierdzeń DeMorgana do redukcji:

While (not((a=b) and (c=d)) do Something;

Do

While (a<>b) or (c<>d) do Something;

2.8. OGÓLNE FUNKCJE BOOLOWSKIE

Dla określonej aplikacji, możemy stworzyć funkcję logiczną która osiąga określone wyniki. .Przypuśćmy, że potrzebujemy napisać program do symulowania wszystkich możliwych funkcji boolowskich. Na przykład, na dołączonej dyskietce ,jest program który pozwoli ci wejść do przypadkowej funkcji boolowskiej ,z jedną do czterech zmiennych. Ten program będzie odczytywał dane wejściowe i tworzył odpowiednie wyniki. Ponieważ liczba unikalnych czterech zmiennych funkcji jest duża (65,536,dokładnie) nie jest możliwe do zastosowania w praktyce zawrzeć określone rozwiązanie dla każdej jedynki w programie. Co jest konieczne dla ogólnej funkcji logicznej, po pierwsze, obliczanie wyników dla przypadkowej funkcji. Ta sekcja opisuje jak napisać taką funkcję. Ogólna funkcja boolowska dla czterech zmiennych potrzebuje pięciu parametrów. – cztery parametry wejściowe i piąty parametr który wyszczególnia funkcję do obliczenia. Podczas gdy jest wiele sposobów do wyszczególnienia funkcji do obliczeń. podamy numer funkcji boolowskiej jako piąty parametr. Na pierwszy rzut oka możemy być zdumieni ,jak można obliczyć funkcje używając numeru funkcji. Jednakże, pamiętajmy, że bity które stanowią numer funkcji pochodzą bezpośrednio z tablicy prawdy dla tej funkcji. Dlatego też ,jeśli wyciągniemy te bity z numeru funkcji, możemy zbudować tablicę prawdy dla tej funkcji .Istotnie, jeśli zaznaczymy i-ty bit z numeru funkcji, gdzie i=D*8+C*4+B*2+A,otrzymamy wynik funkcji dla poszczególnych wartości A,B,C i D. Następujące przykłady w C i Pascalu, pokazują jak napisać taką funkcję.

```
/***************************************************************************************/
/*                                                                                     */
/*  Ten program w C demonstruje jak napisać ogólną funkcję logiczną, która może obliczyć dowolną funkcję     */
/*   czterech zmiennych. Podane operatory manipulowania bitami  C wraz  z heksadecymalnym I/O czynią to     */
/*   zadanie łatwym do wykonania w języku C                                             */
/*                                                                                     */
/***************************************************************************************/

#include<stdlib.h>
#include <stdio.h>

/* Ogólna funkcja logiczna. Parametr „Func" zawiera 16 bitowy numer funkcji logicznej. Jest to w      */
/* rzeczywistości zaszyfrowana tablica prawdy dla funkcji. Parametry a, b, c, d  są danymi wejściowymi do      */
/* funkcji logicznej  . Jeśli potraktujemy 'func' jako tablicę bitów 2 x 2 x 2 x 2.      */
/*  ta określona funkcja wybierze bit „func [d,c,b,a] z func.      */

int
generic (int  func,  int a,  int b,  int c,  int d)
{
        /* Zwraca bit określony przez a, b, c i d */
        return (func >> (a + b*2 + c* 4 + d*8)) & 1;
}
/* Program główny do kierowania ogólną funkcją logiczną napisaną w C  */

main{}
{
        int  func , a ,b ,c ,d;
        /*Powtarzanie dopóki użytkownik nie wprowadzi zera*/
        do
        {
                /* Pobranie numery funkcji (tablica prawdy) */
                printf("Wprowadź wartość funkcji (hex):  „);
                scanf („%x, &func);
                /* Jeśli użytkownik określił zero jako numer funkcji nastąpi zatrzymanie programu */
                if (func != 0)
                {
                        printf („Wprowadź wartości dla d, c, b i a:  „);
                        scanf („%d%d%d%d", &d, &c, &b, &a);
                        printf ("Wynik to %d \n", generic (func, a, b, c, d));
                        printf ("Func = %x, A=%d, B=%d, C=%d, D = %d \n", func, a, b, c, d);
                }
```

```
        ) while (func != 0);
}
```

Następujący program w Pascalu jest napisany w Pascalu standardowym. Standardowy Pascal nie zawiera żadnych operacji do manipulowania bitami, więc ten program jest przydługi, ponieważ musi symulować używanie bitów w tablicy liczb całkowitych. Większość nowoczesnych Pascali (w szczególności Turbo Pascal)zawiera wbudowane operacje na bitach lub biblioteki które operują na bitach. Ten program byłby łatwiejszy do napisania przy użyciu takich nie standardowych cech.

```
Program GenericFunc (input , output);
{*Ponieważ standardowy Pascal nie dostarcza łatwego sposobu bezpośredniej manipulacji bitami w liczbie        *}
{*, zasymulujemy numer funkcji używając tablicy 16 liczb całkowitych . „GFTYPE” jest typem tej tablicy      *}

type
        gftype = array [0..15] of integr;
var
        a, b, c, d : integer;
        fresult; integer;
        func: gftype;

(* Standardowy Pascal nie dostarcza możliwości przesuwania danej całkowitej z lewa w prawo. Dlatego też  *)
(* zasymulujemy 16 bitową wartość używając tablicy 16 liczb całkowitych,. Możemy zasymulować              *)
(* poprzez przenoszenie danych wokół tablicy. Zauważ ,że Turbo Pascal dostarcza operatorów shl i shr        *)
(* Jednak, kod ten jest napisany do działania ze standardowym Pascalem  , a nie Turbo Pascalem tylko.        *)
(* ShiftLeft przesuwa wartości w func na pozycję w lewo  i wprowadza przesuniętą wartość na „pozycję bitu”*)
(* zero                                                                                                    *)

procedure ShiftLeft(shiftin: integr);
var  i : integer;
begin
        for 1 := 1 5 downto 1 do func[i] := func[i-1];
        func[0] := shiftin;
end;
(* ShiftNibble przesuwa daną w func w lewo o cztery pozycje I wprowadza cztery bity a (L.O.), b, c  i d (H.O) *)
(* na wakujące pozycje                                                                                     *)

procedure ShiftNibble (d,c,b,a: integer);
begin
        ShiftLeft (d);
        ShiftLeft ( c );
        ShiftLeft (b);
        ShiftLeft(a);
end;
(* ShiftRight przesuwa dane w func jedną pozycję w prawo. Przesuwa zero do bitu H.O. tablicy                *)

procedure ShiftRight;
var     i : integer;
begin
        for i := 0 to 14 do func[i] := func[i+1];
        func[15] := 0;
end;
(* ToUpper konwertuje małe znaki na duże znaki.                                                            *)

procedure toupper (var ch:char);
begin
        if (ch in[‘a’ .. ‘z’]) then ch :=(ord(ch)- 32_;
```

```
end;
(* ReadFunc odczytuje numer funkcji heksadecymalnej od użytkownika i odkłada t a wartość do tablicy func     *)
(* (bit po bicie)                                                                                           *)
funkcja ReadFunc: integer;
var     ch:char;
        i, val : integer;
begin
        write (‘Wprowadź numer funkcji (heksadecymalnie): ‘);
        for i := 0 to 15 do func[i] := 0;
        repeat
                read (ch);
                if not eoln then begin

                        toupper (ch);
                        case ch of
                                ‘0’: ShiftNibble(0,0,0,0);
                                ‘1’: ShiftNibble(0,0,0,1);
                                ‘2’: ShiftNibble(0,0,1,0);
                                ‘3’: ShiftNibble(0,0,1,1);
                                ‘4’: ShiftNibble(0,1,0,0);
                                ‘5’: ShiftNibble(01,0,1);
                                ‘6’: ShiftNibble(0,1,1,0);
                                ‘7’: ShiftNibble(0,1,1,1);
                                ‘8’: ShiftNibble(1,0,0,0);
                                ‘9’: ShiftNibble(1,0,0,1);
                                ‘A’: ShiftNibble(1,0,1,0);
                                ‘B’: ShiftNibble(1,0,1,1);
                                ‘C’: ShiftNibble(1,1,0,0);
                                ‘D’: ShiftNibble(1,1,0,1);
                                ‘E’: ShiftNibble(1,1,1,0);
                                ‘F’: ShiftNibble(1,1,1,1);
                        end;
                end;
        until   eoln;
        val     := 0;
        for i := 0 t o15 do val  := val + func[i];
        ReadFunc := val
end;
(* Generic – oblicza ogólną funkcję logiczną określoną przez numer funkcji “func” na czterech danych      *)
(* zmiennych a ,b, c  i d. Robi to przez zwracane bity d*8 + c*4 +b*2 + a z funkcji                       *)

function Generic (var func: gftype; a,b,c,d: integer): integer;
begin
        Generic := func [a+b*2 + c*4 + d*8];
end;
begin  (* main *)
                repeat
                        fresult := ReadFunc;
                        if (fresult  <> 0) then begin
                                write (Wprowadź wartości dla D , C., B i A (0/1): ‘);
                                readln (d,c,b,a);
                                writeln(“Wynik to ‘, Generic(func, a,b,c,d);
                        end;
                until   fresult = 0;
end.
```
Następujący kod pokazuje potęgę operacji manipulowania .Ta wersja kodu powyżej używa specjalnych cech

przedstawionych w Turbo Pascalu, które pozwalają programistom na przesuwanie w lewo i w prawo i robienia bitowania logicznego AND na zmiennych całkowitych:

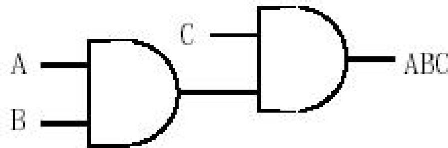
```
program GenericFunc (input, output) ;
const
        hex = [‘a’..’f’, ‘A’…’F’];

        dziesiętnie = [‘0’…’9’];
var
        a,b,c,d :integer;
        fresult: integer;
        func: integer;
(* Tu mamy drugą wersję ogólnej funkcji pascalowskiej , która używa cech Turbo Pascala  do uproszczenia      *)
(* programu                                                                                                   *)

function  ReadFunc: integer;
var     ch: char;
        i, val : integer;
begin
        write (‘Wprowadź numer funkcji (heksadecymalnie): ‘;
        repeat
        read (ch);
        func := 0;
        if not eoln then begin
                if (ch in Hex) then
                        func := (func shl 4) + (ord(ch) and 15) + 9
                else if (ch in Decimal)) then
                        func := (func shl 4) + (ord(ch) and 15)
                else write(chr(7));
                end;
        until   eoln;
        ReadFunc := func;

end;
(* Generic – oblicza ogólną funkcję logiczną określoną przez numer funkcji “func” dla czterech zmiennych      *)
(*  a,b,c  i d. Robi to przez zwracany bit d*8 + c*4 + b*2 + a z func. Wersja ta polega na operatorze przesunięcia*)
(* w prawo Turbo Pascala i jego zdolności do operacji na poziomie bitowym na liczbach całkowitych               *)

function Generic (func, a, b, c, d: integer): integer;
begin
        Generic := (func shr (a+ b*2+ c*4 + d*8)) and 1;
end;

begin    (*main *)
        repeat
                fresult := ReadFunc;
                if      (fresult <> 0) then begin
                        write (‘Wprowadź wartości dla D, C, B I A (0/1): ‘);
                        readln(d,c,b,a);
                        writeln (‘Wynik to ‘, Generic(func,a,b,c,d));
                end;
        until   fresult = 0;
end.
```

2.11 PODSUMOWANIE

Algebra Boole’a dostarcza podstaw dla sprzętu i programów komputera. Pobieżne zrozumienie tego systemu matematycznego może pomóc ci lepiej rozumieć połączenia między programem a sprzętem. Algebra boolowska jest

systemem matematycznym ze swoim własnym zbiorem zasad (aksjomaty),twierdzeniami i wartościami. Pod wieloma względami, algebra boolowska jest podobna do prawdziwej algebry arytmetycznej, którą zajmowałeś się w szkole. Jednak pod wieloma względami algebra boolowska jest właściwie łatwiejsza do nauczenia się niż prawdziwa algebra. Ten rozdział zaczął się od omówienia cech tego systemu algebraicznego, zawierającego :operatory ,zamknięcie, przemienność, rozdzielność, łączność ,tożsamość i element odwrotny. Potem przedstawionych jest kilka ważnych aksjomatów i twierdzeń z algebry boolowskiej i omówiona zasada dualności która pozwala łatwo dowieść dodatkowych twierdzeń w algebrze boolowskiej po szczegóły zajrzyj:
* „Algebra Boole'a"
Tablice prawdy są dogodnym sposobem wizualnej reprezentacji funkcji boolowskich lub wyrażeń Każda boolowska funkcja (lub wyrażenie) ma odpowiadającą mu tabelę prawdy, która dostarcza wszystkich możliwych wyników dla każdej kombinacji danych wejściowych. Ten rozdział przedstawia kilka różnych sposobów do budowania boolowskich tablic prawdy. Chociaż jest nieskończona liczba funkcji boolowskich można stworzyć danych n wartości wejściowych okazuje się że jest skończona liczba unikalnych funkcji możliwa dla danej liczby danych wejściowych. W szczególności jest 2^2^2 unikalnych funkcji boolowskich z n danych wejściowych. Na przykład. jest 16 funkcji dla dwóch zmiennych (2^2^2 = 16). Ponieważ jest mało funkcji boolowskich tylko z dwoma danymi wejściowymi, łatwo jest przydzielić różne nazwy dla każdej z tych funkcji (np. .AND,OR,NAND itp. Dla funkcji z trzema lub więcej zmiennymi, liczba funkcji jest zbyt duża aby dawać każdej funkcji jej własną nazwę Dlatego też. ,przydzielamy liczbę do tych funkcji opartą na bitach pojawiających się w tabeli prawdy funkcji. Po szczegóły zajrzyj:
* „Funkcje Boolowskie I Tablice Prawdy"
Możemy manipulować funkcjami boolowskimi i wyrażeniami algebraicznymi. Pozwala to nam dowodzić nowych teorii w algebrze boolowskiej, upraszczać wyrażenia ,konwertować wyrażenia do postaci kanonicznych lub pokazywać że dwa wyrażenia są sobie równoważne. Zobacz kilka przykładów z algebraicznej manipulacji wyrażeniami algebraicznymi, sprawdź.
* „Manipulacja Algebraiczna Wyrażeniami Boolowskimi"
Ponieważ jest nieskończony wybór możliwych funkcji boolowskich ,mimo to skończona liczba unikalnych funkcji boolowskich (dla stałej liczby danych wejściowych), jest nieskończona liczba różnych funkcji boolowskich które obliczają takie same wyniki. Aby uniknąć zamieszania, projektanci logiczni zwykle wyszczególniają funkcje boolowskie używając postaci kanonicznych. Jeśli dwa kanoniczne równania są różne. wtedy przedstawiają różne funkcje boolowskie. Ta książka opisuje dwie różne postacie kanoniczne: sumę pełnych iloczynów i iloczyn pełnych sum. Naucz się o tych postaciach kanonicznych ,jak konwertować przypadkowe boolowskie równania do formy kanonicznej i jak konwertować między dwoma postaciami kanonicznymi zobacz
* „Postacie Kanoniczne"
Chociaż postacie kanoniczne dostarczają unikalnych przedstawień dla danej funkcji boolowskiej, wyrażenia pojawiające się w postaci kanonicznej, rzadko są optymalne. To znaczy ,wyrażenie kanoniczne często używa literałów i operatorów ,równoważników, wyrażeń. Znajomość jak stworzyć formę zoptymalizowaną boolowskiego wyrażenia jest bardzo ważna. Ten tekst omawia ten temat w
* „Upraszczanie Funkcji Boolowskich"
Algebra boolowska nie jest systemem zaprojektowanym przez jakiegoś szalonego matematyka o małym znaczeniu w świecie. .Algebra Boole'a jest podstawą logiki cyfrowej, podstawą dla projektantów komputerowych. Co więcej jest indywidualna równoważność między cyfrowym hardware a komputerowym software Cokolwiek zbudujesz w hardware możesz zbudować z software i vice versa Ten tekst opisuje jak zaimplementować dodatkowo, dekodery, pamięć, rejestry przesuwne i liczniki używając tych funkcji boolowskich. Podobnie ten tekst opisuje jak poprawić wydajność software (np. Programy Pascala) przez zastosowanie zasad i teorii algebry boolowskiej. Wszystkie te szczegóły zobacz:
* „Jaki to ma związek z komputerami"
* „Równoważność między układami elektronicznymi a funkcjami boolowskimi"
* „Układy kombinacyjne"
* „Okay, co nam to da przy programowaniu?"
2.12 PYTANIA:
1. Jaki jest tożsamy element pod względem :
a) AND b) OR c)XOR d)NOT e)NAND f)NOR
2. Stwórz tabele prawdy dla następujących funkcji z dwoma zmiennymi:
a) And b)OR c)XOR d)NAND e) NOR f)Równoważnej g)A<B h)A>B i) A
3. Stwórz tablice prawdy dla następujących funkcji z trzema zmiennymi wejściowymi:

a) ABC(AND) b) A+B+C (OR) c) (ABC)’ (NAND) d) (A+B+C)’ (NOR)
e) równoważnik (ABC)+(A’B’C’) f) XOR (ABC+A’B’C’)’
4. Pokaż schematycznie (diagram układu elektrycznego) jak zaimplementować każdą z funkcji w pytaniu trzecim używając tylko dwóch bramek wejściowych i inwertora. Np.
a) ABC =



5.Pokaż implementację bramek AND,OR i inwertera przy użyciu jednej lub więcej bramek NOR.
6. Co to jest zasada dualności?
7. Zbuduj pojedynczą tabelę prawdy która uwzględnia dane wyjściowe dla następujących funkcji boolowskich z trzema zmiennymi:
F_x=A+BC F_y=AB+C’B F_z=A’B’C+ABC+C’B’A
8. Uzyskaj numer funkcji dla trzech funkcji z pytania siedem.
9.Ile możliwych (unikalnych) funkcji boolowskich mamy jeśli funkcja ma:
a) jedno wejście b) dwa wejścia c) trzy wejścia d)cztery wejścia e) pięć wejść
10. Uprość następujące funkcje boolowskie używając transformacji algebraicznych.
a) F=AB+AB’ b) F=ABC+BC’+AC+ABC’ c) F=A’B’C’D+A’B’C’D+A’B’CD+A’B’CD’
d) F=A’BC+ABC’+A’BC’+AB’C’+ABC+AB’C
11. Uprościj funkcje boolowskie z pytania 10 używając metody map.
12. Ułóż równania logiczne w postaci kanonicznej dla funkcji boolowskich S_0...S_6 dla siedmiu segmentów wyświetlacza (zobacz „Układy kombinacyjne”)
13. Stwórz tablice prawdy dla każdej funkcji z pytania 12
14. Zminimalizuj każdą funkcję z pytania 12 używając metody map
15. Równanie logiczne dla „pół sumatora” (w postaci kanonicznej) to:
Sum=AB’+A’B Carry=AB
a)stwórz diagram układu elektronicznego dla „pół sumatora” używając bramek AND,OR i inwertera.
b)stwórz układ używając tylko bramki NAND
16.Równania kanoniczne dla „pełnego dodawania” przyjmują formę:
Sum=A’B’C+A’BC’+AB’C’+ABC
Carry=ABC+ABC’+AB’C+A’BC
a)stwórz schemat dla tych układów używając bramek AND,OR i inwertera.
b)optymalizuj te równania używając metody map
c)stwórz układ elektroniczny dla wersji zoptymalizowanej (używając bramek AND,OR i inwertera)
17.Załóżmy,że masz przerzutnik D (użyj definicji x tego tekstu) którego dane wyjściowe obecnie to Q=1 a Q’=0. Opisz w najdrobniejszych szczegółach dokładnie co zdarzy się kiedy na linię zegara dojdzie:
a) zmiana stanu z niskiego na wysoki przy D=0
b) zmiana stanu z wysokiego na niski przy D=0
18.Przepisz następujące wyrażenia Pascala tak ,aby uczynić je bardziej wydajnymi:
a) if (x or (not x and y)) then write(‘1’);
b) while(not x and not y) do somefunc(x,y);
c) if not ((x<>y) and (a=b) then something;
19. Sprowadź do postaci kanonicznej (suma pełnych iloczynów) :
a)F(A,B,C)=A’BC+AB+BC b)F(A,B,C,D)=A+B+CD’+D
c) F(A,B,C)=A’B+B’A d) F(A,B,C,D)=A+BD’
e)F(A,B,C,D)=A’B’C’D+AB’C’D’+CD+A’BCD’
20. Przekształć sumę pełnych iloczynów z pytania 19 do iloczynu pełnych sum