Algebra Boole'a i logika cyfrowa

7.X. 2009

1 Aksjomatyczna definicja algebry Boole'a

Do opisywanie układów cyfrowych będziemy używali formalizmu nazywanego algebrą Boole'a. Formalnie algebra Boole'a to struktura matematyczna złożona z uniwersum B i zdefiniowanych na nim trzech działań: dwuargumentowych **and** i **or**, oznaczanych przez $\cdot$ i $+$ oraz jednoargumentowego **not**, oznaczanego przez $^-$ (pozioma kreska nad argumentem). Priorytet operatorów: not, and, or. Podajemy następujący zestaw aksjomatów algebry Boole'a (spotykane są też inne warianty):

1. łączność i przemienność $+$ i $\cdot$
2. istnieje element neutralny działania $+$, oznaczany przez 0, czyli $x + 0 = x$ dla dowolnego $x \in B$;
3. istnieje element neutralny działania $\cdot$, oznaczany przez 1, czyli $x \cdot 1 = x$ dla dowolnego $x \in B$;
4. $x + \overline{x} = 1$
5. $x \cdot \overline{x} = 0$
6. prawo podwójnego zaprzeczenia: $\overline{\overline{x}} = x$
7. rozdzielność $\cdot$ względem $+$, czyli $x \cdot (y + z) = x \cdot y + x \cdot z$
8. rozdzielność $+$ względem $\cdot$, czyli $x + (y \cdot z) = (x + y) \cdot (x + z)$
9. $\overline{xy} = \overline{x} + \overline{y}$ (prawo de Morgana)
10. $\overline{x + y} = \overline{x} \cdot \overline{y}$ (prawo de Morgana)

Z aksjomatów tych wynika szereg dodatkowych własności, m.in.:

1. $0 \cdot x = 0$, $1 + x = 1$
2. idempotentność: $x + x = x$, $x \cdot x = x$
3. prawo absorpcji: $x(x + y) = x$, $x + xy = x$

Modelami algebry Boole'a są rodziny podzbiorów ustalonego zbioru, z działaniami przekroju, sumy i dopełnienia zbiorów. Model, który nas będzie interesował: Zbiór $B = \{0, 1\}$ z działaniami logicznej sumy, iloczynu i negacji. Elementy $0, 1$ są czasem nazywane *fałszem* i *prawdą*. Jak się niedługo przekonamy w takim modelu bardzo wygodnie opisuje się działanie cyfrowych układów komputerowych. Tak naprawdę nasz model jest *izomorficzny* ze zbiorem podzbiorów zbioru 1-elementowego.

2 Wyrażenia i funkcje boolowskie

Od teraz będziemy się poruszać w naszym modelu dwuelementowym. Używając symboli działań, stałych 0 i 1 oraz zmiennych (zazwyczaj $x, y, z, \ldots$) możemy budować **wyrażenia boolowskie**. Każde wyrażenie w naturalny sposób definiuje **funkcję boolowską**. Funkcję taką możemy opisać używając **tabeli prawdy**. Przykład:

x	y	z	$F = x + \overline{y}z$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Czasem konstruując tablicę prawdy dodajemy jeszcze dla wygody kolumny pośrednie (np. $\overline{y}z$).

Okazuje się, że każda funkcja boolowska da się opisać za pomocą wyrażenia boolowskiego. Dowód: konstrukcja wyrażenia w dysjunkcyjnej postaci normalnej (...). Dwa wyrażenia boolowskie nazywamy **równoważnymi** jeśli opisywane przez nie funkcje boolowskie są identyczne. Oczywiście każde wyrażenie ma nieskończenie wiele wyrażeń równoważnych, a stąd każda funkcja może być zapisana na nieskończenie wiele sposobów. Najlepsze są reprezentacje możliwie najprostsze (w jakimś sensie).

2.1 Upraszczenie wyrażeń boolowskich

Używając aksjomatów i praw algebry Boole'a możemy próbować upraszczać wyrażenia boolowskie (proces ten jest odpowiednikiem upraszczenia obwodów cyfrowych komputera). Przykład: $F(x, y, z) = \overline{x}yz + \overline{x}y\overline{z} + xz$. Upraszamy: $\overline{x}y(z + \overline{z}) + xz = \overline{x}y(1) + xz = \overline{x}y + xz$. Trudniejszy przykład:

$$
\begin{aligned}
& F(x, y, z) = xy + \overline{x}z + yz \\
& = xy + \overline{x}z + yz(1) \text{ (el. neutralny)} \\
& = xy + \overline{x}z + yz(x + \overline{x}) \text{ (uzupełnienie)} \\
& = xy + \overline{x}z + (yz)x + (yz)\overline{x} \text{ (rozdzielność)} \\
& = xy + \overline{x}z + x(yz) + \overline{x}(yz) \text{ (przemienność)} \\
& = xy + \overline{x}z + (xy)z + (\overline{x}z)y \text{ (łączność)} \\
& = xy + (xy)z + \overline{x}z + (\overline{x}z)y \text{ (przemienność)} \\
& = xy(1 + z) + \overline{x}z(1 + y) \text{ (rozdzielność)} \\
& = xy(1) + \overline{x}z(1) \\
& = xy + \overline{x}z
\end{aligned}
$$

W podobny sposób możemy dowodzić praw algebry Boole'a (czyli uzasadniać, że dwa różne wyrażenia boolowskie opisują tą samą funkcję).

Upraszczając wyrażenia na podstawie praw algebry Boole'a zdajemy się nie tyle na algorytm co na własną pomysłowość. Co więcej zauważmy, że my w zasadzie nie wiemy co to znaczy najprostsza postać wyrażenia... Zdefinujmy zatem pewne standardowe „proste" postacie wyrażenia. Dowodząc, że każda funkcja boolowska może być zapisana jako wyrażenie boolowskie użyliśmy wyrażeń boolowskich o szczególnej budowie: w **dysjunkcyjnej postaci normalnej**. Wyrażenia w dysjunkcyjnej postaci normalnej to sumy iloczynów: $F(x, y, z) = xy + \overline{x}yz + y\overline{z}$. Każdą

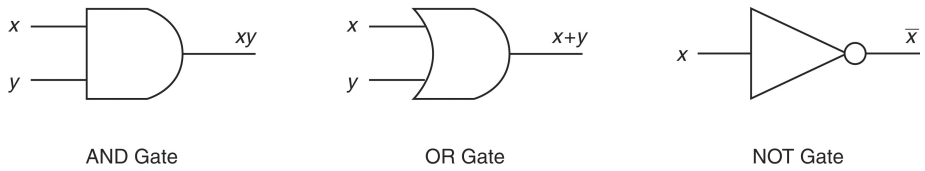
funkcję boolowską można zapisać w tej postaci. Podobnie możemy wprowadzić **koniunkcyjną postać normalną** – postać iloczynu sum. Iloczyny w d.p.n. nazywamy **mintermami**. Sumy w k.p.n. to **makstermy**. Zauważ, że funkcja wciąż może mieć wiele reprezentacji w d.p.n. Dla nas najlepsze będą te, które mają najmniejszą możliwą liczbę mintermów, a wśród reprezentacji z jednakową liczbą mintermów preferować będziemy te, które mają mniej zmiennych w mintermach. Niestety takie określenie wciąż nie definiuje jednoznacznej minimalnej postaci. Np. $\overline{x}\,\overline{y} + xy + \overline{x}z = \overline{x}\,\overline{y} + xy + yz$, to dwie minimalne postaci tego samego wyrażenia.

3 Bramki logiczne

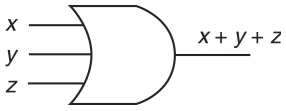
Rysunki w tym rozdziale zostały pobrane z oficjalnej strony podręcznika Lindy Null i Julii Lobur.

Dobra strona o logice cyfrowej: `http://www.play-hookey.com/digital/` (zawiera opisy układów, interaktywne diagramy, ...).

Bramki logiczne są podstawowymi elementami z jakich budujemy komputery. W zależności od typu bramka logiczna zbudowana jest z jednego, dwóch lub kilku tranzystorów. Na początek wprowadzamy bramki odpowiadające operatorom algebry Boole'a – bramki **AND**, **OR**, **NOT**. Oprócz bramek dwuwejściowych możemy używać ich naturalnych kilkuwejściowych wersji. Czasami, oprócz podstawowego wyjścia bramki dorysowujemy jej drugi wyjście, będące negacją pierwszego. Czasem, zamisat rysować bramkę negacji, na wejściu kolejnej bramki rysujemy puste kółeczko.



Rysunek 1: Symbole podstawowych bramek logicznych

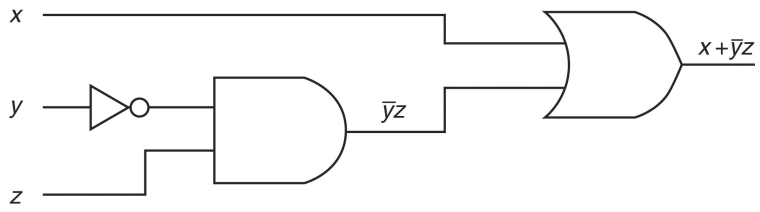


Rysunek 2: Trzywejściowa bramka OR

Używając **diagramów logicznych** możemy reprezentować wyrażenia boolowskie. Przykład $F(x, y, z) = x + \overline{y}z$.

Ogólnie bramki odpowiadają dwu- lub kilkuagrumentowym funkcjom logicznym. Jest zatem 2^4 typów bramek o dwóch wejściach. Niektóre z nich sa popularniejsze od innych i mają swoje własne symbole i nazwy. Często wykorzystywaną bramką jest bramka **XOR** (rys. 4). Kolejne bramki: **NOR** (rys. 5) i **NAND** (rys. 6).

Mówimy, że zbiór bramek (lub odpowiadających im funkcji logicznych) jest funkcjonalnie pełny jeśli można za jego pomocą wyrazić każdą funkcję logiczną. Pokazaliśmy już, że zbiór { AND, OR, NOT } jest funkcjonalnie pełny (konstrukcja wyrażenia w dysjunkcyjnej postaci normalnej). Łatwo wyeliminować z niego bramkę AND lub OR (używając prawa de Morgana



Rysunek 3: Prosty diagram logiczny

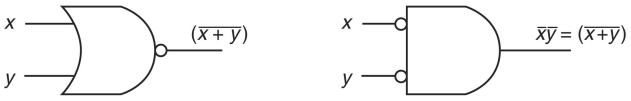
x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

(a)

(b)

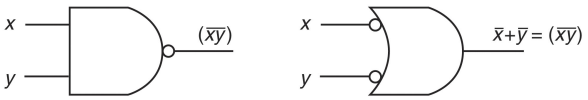
Rysunek 4: Bramka XOR i jej tablica prawdy

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0



Rysunek 5: Bramka NOR: tablica prawdy, symbol, realizacja za pomocą AND i NOT

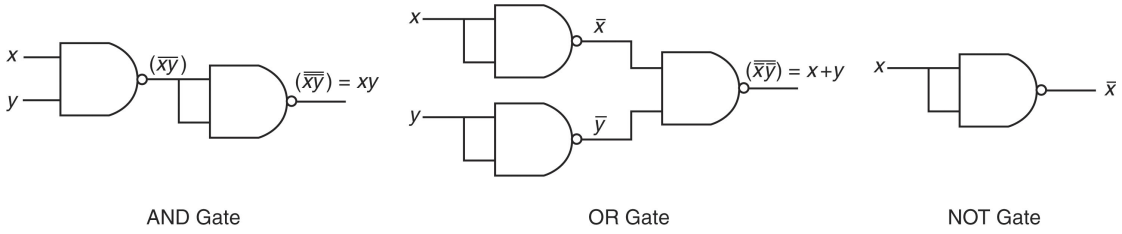
x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0



Rysunek 6: Bramka NAND: tablica prawdy, symbol, realizacja za pomocą OR i NOT

symulujmemy jedną za pomocą drugiej i negacji). Okazuje się, że bramka NAND (podobnie jak i NOR) stanowi sama w sobie zbiór funkcjonalnie pełny. Dowód: realizujemy AND, OR i NOT przez NAND (rys. 7). Wynika stąd, że w pełni funkcjonalny komputer można by zbudować używając tylko bramek NAND. W praktyce używa się bramek różnego typu, bo to po prostu pozwala budować mniej skomplikowane obwody.

Wiadomo, że konstruując obwody logiczne warto starać się używać jak najmniejszej liczby bramek. Zwróćmy jeszcze w tym miejscu uwagę na jedną rzecz jaką warto brać pod uwagę. Popatrzmy na przykład: $F = ((ab + c)d) + e = abd + cd + e$. Która postać jest lepsza i dlaczego?



Rysunek 7: Realizacja AND, NOT i OR przez NAND

Różnica polega na liczbie poziomów bramek (narysuj oba obwody). Fizyczna realizacja drugiej wersji będzie wyliczała wartość wyrażenia nieco szybciej niż pierwszej.

Ponieważ każde wyrażenie można zapisać w dysjunkcyjnej postaci normalnej, więc każdy obwód można zrealizować używając tylko dwóch poziomów bramek. Nie znaczy to, że zawsze jest to praktyczne i pożądane (podstawowa niedogodność: potrzebujemy do tego celu bramek o dużej liczbie wejść, które same w sobie muszą być skomplikowane).