Rozdział 2

Algebra Boole'a i układy logiczne

W Dodatku A.1 przedstawiono elementy teorii mnogości a w Dodatku A.2 przedstawiono pokrótce elementy algebry (między innymi struktury algebraiczne).

2.1 Zbiór wartości i operacje

Niech Ω będzie dowolnym zbiorem zawierającym dwa szczególne elementy 0 i 1.

Na elementach zbioru Ω określone są następujące trzy operacje:

dopełnienie $\overline{\square} : \Omega \mapsto \Omega$

suma $+ : \Omega \times \Omega \mapsto \Omega$

iloczyn $\cdot : \Omega \times \Omega \mapsto \Omega$

Algebrą Boolea nazywamy strukturę algebraiczną

$$\langle \Omega, 0, 1, +, \cdot, \overline{\square} \rangle,$$

w której operacje rządzą się następującymi prawami:

łączność $a + (b + c) = (a + b) + c,\ a \cdot (b \cdot c) = (a \cdot b) \cdot c$

przemienność $a + b = b + a,\ a \cdot b = b \cdot a$

absorpcja $a + (a \cdot b) = a,\ a \cdot (a + b) = a$

rozdzielność $a + (b \cdot c) = (a + b) \cdot (a + c),\ a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

pochłanianie $a + \overline{a} = 1,\ a \cdot \overline{a} = 0$

2.2 Przykłady algebr Boole'a

2.2.1 Rodzina podzbiorów

Niech X będzie dowolnym niepustym zbiorem. Za zbiór wartości przyjmijmy zbiór potęgowy $P(X)$, który składa się ze wszystkich podzbiorów zbioru X.

Elementem 0 algebry o zbiorze wartości $\Omega = P(X)$ przyjmiemy zbiór pusty $\varnothing$, natomiast elementem 1 tej algebry będzie cały zbiór X (oczywiście $0, 1 \in \Omega$).

Za operację sumy przyjmujemy operację sumy mnogościowej:

$$A + B = A \cup B = \{x \in X | x \in A \vee x \in B\}.$$

Za operację iloczynu przyjmujemy operację części wspólnej:

$$A \cdot B = A \cap B = \{x \in X | x \in A \wedge x \in B\}.$$

Za operację dopełnienia przyjmujemy dopełnienie mnogościowe:

$$\overline{A} = A^c = \{x \in X | x \notin A\}.$$

Jako ćwiczenie można sprawdzić, że tak przyjęte operacje oraz elementy 0 i 1 spełniają warunku wymagane przez algebrę Boole'a.

2.2.2 Logika boolowska

Za zbiór wartości przyjmiemy zbiór dwuelementowy $\Omega = \{0, 1\}$.

Operacje sumy, iloczynu i dopełnienia zdefiniujemy w postaci poniższych tabelek:

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

a	$\overline{a}$
0	1
1	0

Powyższa algebra w naturalny sposób odpowiada logice rachunku zdań. Wartość 0 odpowiada logicznej wartości fałszu F a wartość 1 odpowiada logicznej wartości prawdy P.

Operacje dopełnienia, sumy i iloczynu odpowiadają kolejno logicznym spójnikom negacji ($\neg$), alternatywy ($\vee$) i koniunkcji ($\wedge$).

W praktyce znajduje szczególne zastosowanie operacja różnicy symetrycznej oznaczanej symbolem $\oplus$, a którą zdefiniować możemy następująco:

$$a \oplus b = (\overline{a} \cdot b) + (a \cdot \overline{b}).$$

W poniższej tabelce podano zależność wartości różnicy symetrycznej od jej argumentów:

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Operacja $\oplus$ ma, między innymi, następujące własności:

1. Jest łączna, czyli można zmieniać kolejność jej wyliczania (przestawiać nawiasy).
2. Dla dowolnej wartości $x \in \{0,1\}$, zachodzi równość $x \oplus x = 0$.
3. Dla dowolnej wartości $x \in \{0,1\}$, zachodzi równość $x \oplus 0 = x$.
4. Dla dowolnej wartości $x \in \{0,1\}$, zachodzi równość $x \oplus 1 = \overline{x}$.

Operację różnicy symetrycznej nazywa się też operacją sumy modulo dwa. Faktycznie jej wartość można policzyć ze wzoru

$$a \oplus b = (a + b) \mod 2,$$

gdzie $+$ jest operacją dodawania liczb całkowitych a operacja $\mod 2$ wylicza resztę z dzielenia przez 2.

Różnicę symetryczną nazywa się również alternatywą wykluczającą bo odpowiada spójnikowi **albo** (ma wartość 1 gdy jeden z argumentów ma wartość 1 ale nie oba jednocześnie).

Wartość 0 i 1 nazywać *bitami* (od ang. **BInary digiT**).

Przedstawimy teraz zastosowanie dla różnicy symetrycznej. Proszę zwrócić uwagę, że poniższy przykład należy traktować poglądowo (w praktyce stosuje się wartości dłuższe niż jednobitowe).

Załóżmy, że dwie osoby Alice i Bob[1] znają wspólny klucz $k \in \{0,1\}$ (jest to ich wspólna tajemnica, którą nie dzielą się z innymi).

Alice ma komunikat $d \in \{0,1\}$. który chce przesłać do Boba. Nie chce jednak przesłać go jawnie, gdyż ktoś podsłuchujący linię komunikacyjną mógłby go poznać.

Przed wysłaniem komunikatu, Alice wylicza szyfrogram $c = d \oplus k$. Oczywiście w pewnych sytuacja bit szyfrogramu c będzie taki sam jak bit oryginalnej wiadomości d, a czasami inny.

Alice wysyła Bobowi szyfrogram c. Bob aby odczytać oryginalną wiadomość wylicza wartość $c \oplus k$. Poprawność tego protokołu przesyłania komunikatów można sprawdzić następująco:

$$c \oplus k = (d \oplus k) \oplus k = d \oplus (k \oplus k) = d \oplus 0 = d.$$

[1] Przy omawianiu protokołów, często przyjmuje się dla pierwszego agenta imię Alice a dla drugiego Bob.

Do końca tego wykładu będziemy zajmować się logiką boolowską z operacjami na bitach.

2.3 Operacje bitowe

Operacje zdefiniowane na pojedynczych bitach można uogólnić na ciągi bitów ustalonych długości (oba argumenty operacji powinny składać się z tej samej liczby bitów).

Aby wyliczyć operację bitową na ciągach bitów, wystarczy liczyć kolejne bity wyniku bit po bicie:

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ + \; 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 \end{array} \qquad \begin{array}{cccc} 0 & 0 & 1 & 1 \\ \cdot \; 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 \end{array} \qquad \begin{array}{cccc} 0 & 0 & 1 & 1 \\ \oplus \; 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \end{array} \qquad \begin{array}{cccc} \multicolumn{4}{c}{} \\ \overline{\square} \; 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 \end{array}$$

2.3.1 Reprezentacja liczb całkowitych

Zanim przystąpimy do omówienia podstawowych operacji na bitach jakie można wykonać np. w języku C/C++, przedstawimy reprezentację liczb całkowitych.

Liczby bez znaku

Istnieje dokładnie 2^n możliwych ciągów złożonych z n bitów. Możemy traktować je jako reprezentacje liczb całkowitych nieujemnych o zakresie wartości od 0 do $2^n - 1$.

Standard języka C/C++ nie określa ile bitów mają reprezentacje standardowych typów. Wiadomo tylko, że są to wielokrotności 8 bitów nazywanych bajtami.

Dostępny jest w C/C++ operator `sizeof`, który dla danego typu podaje ile bajtów pamięci zajmuje dana tego typu.

Standard języka C mówi nam tylko tyle:

$$1 = \text{sizeof(unsigned char)} \leq \text{sizeof(unsigned short int)} \leq$$

$$\leq \text{sizeof(unsigned int)} \leq \text{sizeof(unsigned long int)}.$$

Na komputerze jaki mam w pracy na biurku wartości są następujące:

$$\begin{aligned} \text{sizeof(unsigned char)} &= 1 \\ \text{sizeof(unsigned short int)} &= 2 \\ \text{sizeof(unsigned int)} &= 4 \\ \text{sizeof(unsigned long int)} &= 8 \end{aligned}$$

Zatem dopuszczalne zakresy wartości dla powyższych typów są następujące:

$$\begin{aligned} \text{unsigned char} &= 0\,..\,255 \\ \text{unsigned short int} &= 0\,..\,65535 \\ \text{unsigned int} &= 0\,..\,4294967295 \\ \text{unsigned long int} &= 0\,..\,18446744073709551615 \end{aligned}$$

Liczby ze znakiem

Gdy zastanawiamy się jak można zapisać liczbę ze znakiem, która przyjmowałaby wartości ujemne, dodatnie i zero, to przychodzi do głowy pomysł aby pierwszy od lewej strony bit uznać za bit znaku (0 - liczba nieujemna, 1 - liczba ujemna).

Takie rozwiązanie ma jednak ważną wadę. Byłyby dwie reprezentacje liczby 0. Otóż słowo złożone z samych zer oznaczałoby zero nieujemne a słowo, które po jedynce ma same zera, oznaczałoby zero ujemne (różne od zera nieujemnego).

Przyjęto inny sposób reprezentowania liczb ze znakiem. Nazywa się on *kodem uzupełnień do dwóch* (oznacza U2) i pozwala na n bitach zapisać liczby całkowite od -2^{n-1} do $2^{n-1}-1$.

Pierwszy bit słowa uznaje się za bit znaku. Dla liczb nieujemnych ma on wartość równą 0 a dla liczb ujemnych ma wartość 1.

Jeśli bit znaku ma wartość 0, to na pozostałych $n-1$ bitach zapisanych jest 2^{n-1} wartości nieujemnych z zakresu od 0 do $2^{n-1}-1$.

Jeśli bit znaku ma wartość 1, to na pozostałych $n-1$ bitach zapisanych jest 2^{n-1} wartości ujemnych z zakresu od -2^{n-1} do -1.

Kod U2 dla czterech bitów podano w tablicy 3.1.

Liczby w kodzie U2 można wyobrażać sobie, że leżą na okręgu. Idąc od wartości 0, zgodnie ze wskazówkami zegara, znajdują się kolejno liczby dodatnie $1, 2, \ldots, 2^{n-1}-1$. Robiąc jeden krok dalej wpadamy na najmniejszą wartość -2^{n-1} i dalej idąc zgodnie ze wskazówką zegara, mijamy kolejne coraz większe wartości aż do -1, za którą powracamy do początkowej wartości 0.

Poruszając się w przeciwną stronę, po minięciu najmniejszej wartości -2^{n-1} wpadniemy na największą wartość dodatnią $2^{n-1}-1$.

Co do długości dostępnych typów całkowitych ze znakiem, standard języka C mówi nam tylko tyle:

$$1 = \text{sizeof(signed char)} \leq \text{sizeof(short int)} \leq \text{sizeof(int)} \leq \text{sizeof(long int)}.$$

Na komputerze jaki mam w pracy na biurku wartości są następujące:

$$\begin{aligned} \text{sizeof(signed char)} &= 1 \\ \text{sizeof(short int)} &= 2 \\ \text{sizeof(int)} &= 4 \\ \text{sizeof(long int)} &= 8 \end{aligned}$$

Tablica 2.1: Czterobitowy kodu U2.

bity				wartość
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Zatem dopuszczalne zakresy wartości dla powyższych typów są następujące:

$$\begin{aligned} \text{signed char} &= -128 \,..\, 127 \\ \text{short int} &= -32768 \,..\, 32767 \\ \text{int} &= -2147483648 \,..\, 2147483647 \\ \text{long int} &= -9223372036854775808 \,..\, 9223372036854775807 \end{aligned}$$

Ze sposobem kodowania U2 wiąże się pewna ciekawostka. Otóż o wartości bezwzględnej $|x|$ wiemy, że jest ona nieujemna dla dowolnej wartości liczbowej x. Funkcją, która w C/C++ oblicza wartość bezwzględną dla liczby z typu `int` jest standardowa funkcja `abs(x)`, znajdująca się w bibliotece `glibc`.

Okazuje się, że poniższa formuła:

$$\forall_{x \in \text{int}} \; \text{abs(x)} \geq 0$$

nie jest prawdziwa (!).

Gdy zajrzymy do źródeł funkcji `abs`, przekonamy się, że jest ona zdefiniowana następująco:

```
int
abs (int i)
{
```

```
  return i < 0 ? -i : i;
}
```

Wyrażenie `i < 0 ? -i : i` wylicza się następująco: jeśli wartość parametru `i` jest ujemna, to wartością funkcji `abs` jest `-i`. W przeciwnym przypadku wartością jest `i`.

Stała `INT_MIN` oznacza w C najmniejszą wartość w typie `int`, natomiast stała `INT_MAX` oznacza największą wartość w typie `int`.

Pamiętając, że wartości typu `int` nie leżą symetrycznie względem zera (wartości ujemnych jest o jedna więcej niż dodatnich), wyliczyć możemy, że

$$-\texttt{INT\_MIN} = \texttt{INT\_MAX} + 1 = \texttt{INT\_MIN}.$$

Zatem `abs (INT_MIN) = INT_MIN < 0`.

Zapamiętajmy następujące zależności:

$$\begin{aligned} \texttt{INT\_MAX} + 1 &= \texttt{INT\_MIN} \\ \texttt{INT\_MIN} - 1 &= \texttt{INT\_MAX} \end{aligned}$$

Operacje bitowe w C/C++

W poniższej tabelce podano jakie operatory języka C/C++ odpowiadają operacją na bitach z logiki boolowskiej:

logika boolowska	język C/C++	operacja
$+$	`|`	suma
$\cdot$	`&`	iloczyn
$\overline{\square}$	`~`	dopełnienie
$\oplus$	`^`	różnica symetryczna

Dostępne są również w C/C++ następujące operacje:

język C/C++	operacja
`x >> n`	przesunięcie wartości x o n bitów w prawo
`x << n`	przesunięcie wartości x o n bitów w lewo

2.3.2 Ustawianie bitu

Aby ustawić na 1 w wartości x reprezentowanej n bitami $(b_{n-1}, b_{n-2}, \ldots, b_0)$ bit na pozycji $k \in \{0, 1, \ldots, n-1\}$ należy policzyć maskę złożoną z n bitów, która na pozycji k ma 1 a na pozostałych 0.

Maską taką jest wartość `1 << k`, czyli liczba 1 przesunięta w lewo o k bitów.

Wartość `x | (1 << k)` na pozycjach innych niż k ma te same bity co wartość x, natomiast na pozycji k ma ustawioną 1 (bez względu na to jaka była wcześniej wartość tego bitu).

Zatem w wyniku wykonania instrukcji `x |= 1 << k` zostanie na pozycji k-tej w wartości x wpisana jedynka.

2.3.3 Testowanie bitu

Aby sprawdzić czy w wartości x reprezentowanej n bitami $(b_{n-1}, b_{n-2}, \ldots, b_0)$ bit na pozycji $k \in \{0, 1, \ldots, n-1\}$ ma wartość 1, należy policzyć maskę złożoną z n bitów, która na pozycji k ma 1 a na pozostałych 0.

Wartość `x & (1 << k)` jest różna od zera jedynie wtedy, kiedy w wartości x na k-tej pozycji była jedynka.

2.3.4 Negowanie bitu

Aby zanegować w wartości x reprezentowanej n bitami $(b_{n-1}, b_{n-2}, \ldots, b_0)$ bit na pozycji $k \in \{0, 1, \ldots, n-1\}$ należy policzyć maskę złożoną z n bitów, która na pozycji k ma 1 a na pozostałych 0.

Maską taką jest wartość `1 << k`, czyli liczba 1 przesunięta w lewo o k bitów.

Wartość `x ^ (1 << k)` różni się od wartości `x` na pozycji k-tej, natomiast pozostałe bity są takie same.

Zatem w wyniku wykonania instrukcji `x ^= 1 << k` zostanie w wartości x zanegowany dokładnie jeden bit na na pozycji k-tej.

2.3.5 Kasowanie bitu

Aby skasować w wartości x reprezentowanej n bitami $(b_{n-1}, b_{n-2}, \ldots, b_0)$ bit na pozycji $k \in \{0, 1, \ldots, n-1\}$ należy policzyć maskę złożoną z n bitów, która na pozycji k ma 0 a na pozostałych 1.

Maską taką jest wartość `~(1 << k)`, czyli negacja wszystkich bitów w liczbie 1 przesuniętej w lewo o k bitów.

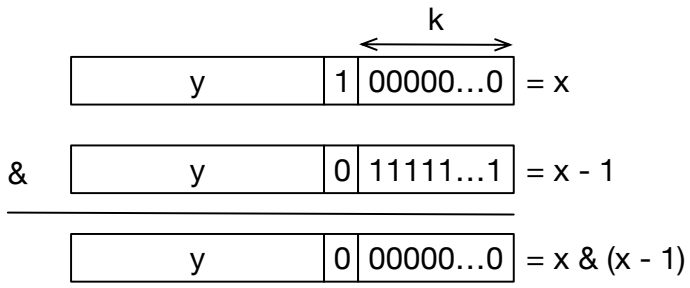
Wartość `x & ~(1 << k)` różni się od wartości `x` na pozycji k-tej, mając tam 0 (pozostałem bity są takie same).

Zatem w wyniku wykonania instrukcji `x &= ~(1 << k)` zostanie w wartości x wyzerowany dokładnie jeden bit na na pozycji k-tej.

2.3.6 Przykład zastosowania

Rozpatrzmy następujący przykład, w którym chcemy policzyć ile jest jedynek w binarnej reprezentacji wartości typu `unsigned long int`. Wartości tego typu mogą służyć do reprezentowania podzbiorów 64-elementowego zbioru. Ich zapis binarny można traktować jako funkcję charakterystyczną, która dla danej wartości określa czy wartość ta należy do podzbioru (bit równy 1) czy nie należy (but równy 0). Zliczanie jedynek w binarnej reprezentacji podzbioru odpowiada zatem zliczaniu liczby jego elementów (ang. cardinality).

Pierwsza wersja funkcji `int cardinality(unsigned long int subset)` wygląda następująco:



Rysunek 2.1: Kasowanie najmniej znaczącej jedynki.

```
int
cardinality (unsigned long int subset)
{
  int counter = 0;
  while(subset)
  {
    counter += subset & 1;
    subset >>= 1;
  }
  return counter;
}
```


Pętla `while` wykonywana jest tak długo, dopóki wartość `subset` jest różna od zera[2] (co najmniej jedna jedynka pozostała w jej binarnej reprezentacji).

Instrukcja `counter += subset & 1` powiększa wartość licznika o wartość najmniej znaczącego bitu w wartości `subset`, natomiast instrukcja `subset >>= 1` przesuwa wartość `subset` o jeden bit w prawo, wypychając najmniej znaczący bit poza zakres.

Powyższa funkcja ma jedną wadę, w najgorszym przypadku wykonuje tyle iteracji jak długa jest reprezentacja wartości `subset` (maksymalnie 64 iteracje).

Można zapisać funkcję w takiej postaci aby liczba iteracji była dokładnie równa liczbie jedynek w binarnej reprezentacji wartości `subset`.

W tym celu należy zastosować instrukcję, która kasuje dokładnie jedną jedynkę w binarnej reprezentacji `subset`.

Taką instrukcją jest `subset &= subset - 1`. Aby przekonać się, że działa ona poprawnie, rozpatrzmy jak wygląda reprezentacja dowolnej wartości `subset`.

Na rysunku 1.2 przedstawiono binarną reprezentację wartości $x > 0$. Kończy się ona $k \geq 0$ bitami 0, przed którymi jest bit 1 a przed nim dowolny ciąg

[2] Przypominam, że w języku C/C++ każda wartość całkowita różna od zera oznacza, w kontekście warunku logicznego, prawdę.

(być może pusty) bitów y. Reprezentacja wartości $x-1$ różni się od reprezentacji wartości x tym, że kończy się ona k bitami 1, przed którymi jest bit 0 a przed nim dokładnie ten sam ciąg bitów y. Wykonując operację bitowej koniunkcji na tych dwóch wartościach zostaje skasowana najmniej znacząca jedynka a ciąg bitów y pozostaje niezmieniony.

Poniżej przedstawiono ulepszoną wersję funkcji `cardinality`, która wykonuje dokładnie tyle iteracji pętli ile było jedynek w binarnej reprezentacji argumentu `subset`:

```
int
cardinality (unsigned long int subset)
{
  int counter = 0;
  while(subset)
  {
    subset &= subset - 1;
    counter++;
  }
  return counter;
}
```


2.4 Bramki logiczne

Bramką logiczną nazywamy układ elektroniczny, który posiada pewną liczbę wejść $x_1, x_2, \ldots, x_n$, gdzie $n \geq 1$, i jedno wyjście y. Na wejściach mogą pojawiać się niskie ($0V$) albo wysokie ($5V$) stany. Stan niski oznacza wartość 0, natomiast stan wysoki oznacza wartość 1. Stan wyjścia zależy od wyliczanej funkcji boolowskiej $y = f(x_1, x_2, \ldots, x_n)$.

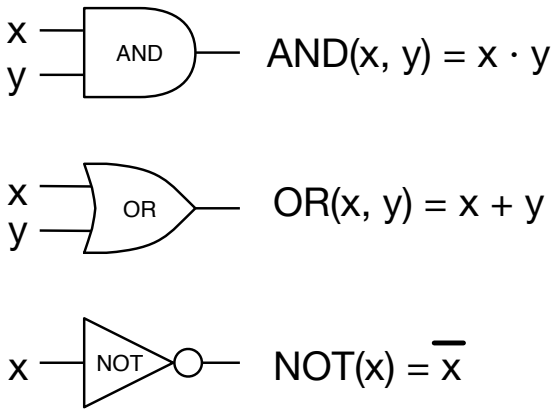
Na rysunku 2.2 przedstawiono trzy podstawowe bramki logiczne.

Aby wyliczyć różnicę symetryczną można skorzystać ze wzoru:

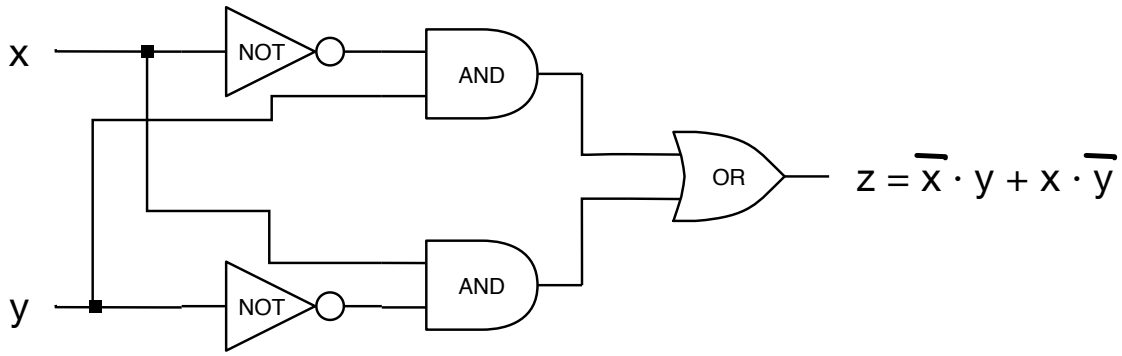
$$XOR(x, y) = \overline{x} \cdot y + x \cdot \overline{y}$$

i połączyć bramki logiczne jak na rysunku 2.3.

Operacja różnicy symetrycznej jest często używana i zaprojektowano dla niej bramkę XOR:



Rysunek 2.2: Podstawowe bramki logiczne AND, OR i NOT.



Rysunek 2.3: Układ wyliczający różnicę symetryczną.

Tablica 2.2: Zależność bitów wyjścia sumatora od bitów wejścia.

x	y	c_{in}	c_{out}	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2.5 Układy logiczne

Łącząc wyjścia jednych bramek z wejściami innych można budować układy logiczne wyliczające bardziej złożone operacje bitowe.

2.5.1 Sumator

Rozpoczniemy od prostego układu, który wylicza arytmetyczną sumę dwóch bitów x i y. Będzie on uwzględniał bit przeniesienia z poprzedniej pozycji c_{in}.

Arytmetyczna suma trzech bitów x, y i c_{in} może przyjąć wartość z zakresu od 0 do 3. Do wyrażenia jej będą użyte dwa bity z i c_{out}.

W poniższych wzorach operacja + będzie sumą arytmetyczną a nie logiczną.

Bity wyniku możemy zdefiniować następująco (operacja $\div$ to iloraz całkowity):

$$\begin{aligned} z &= (x + y + c_{in}) \mod 2, \\ c_{out} &= (x + y + c_{in}) \div 2. \end{aligned}$$

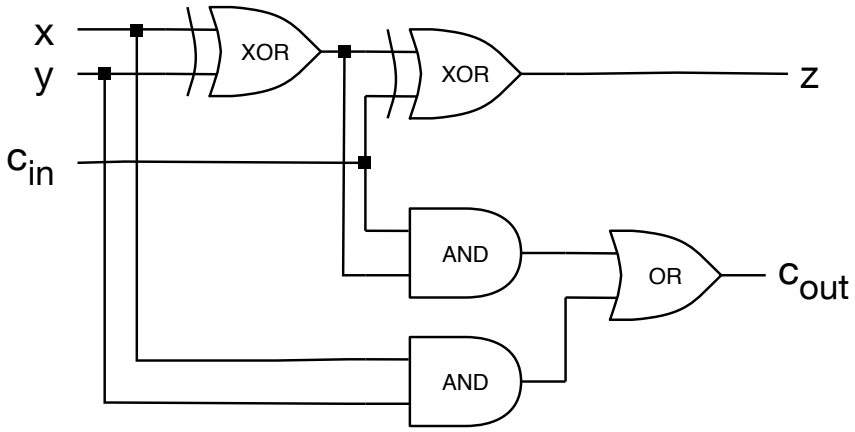
W tablicy 2.2 przedstawiono zależność bitów wyjściowych od wejściowych spełniających powyższe równania.

Jak łatwo zauważyć bity c_{out} i z w tablicy 2.2 są binarnym zapisem wartości sumy trzech bitów wejścia x, y i c_{in}.

Wyliczyć można je stosując operacje logiczne w następujący sposób:

$$\begin{aligned} z &= x \oplus y \oplus c_{in} \\ c_{out} &= x \cdot y + c_{in} \cdot (x \oplus y) \end{aligned}$$

Układ wyliczający bity z i c_{out} przedstawiono na rysunku 2.4.

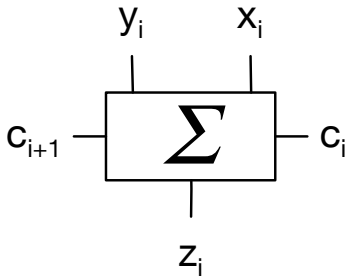


Rysunek 2.4: Układ wyliczający arytmetyczną sumę trzech bitów.

2.5.2 Sumator kaskadowy

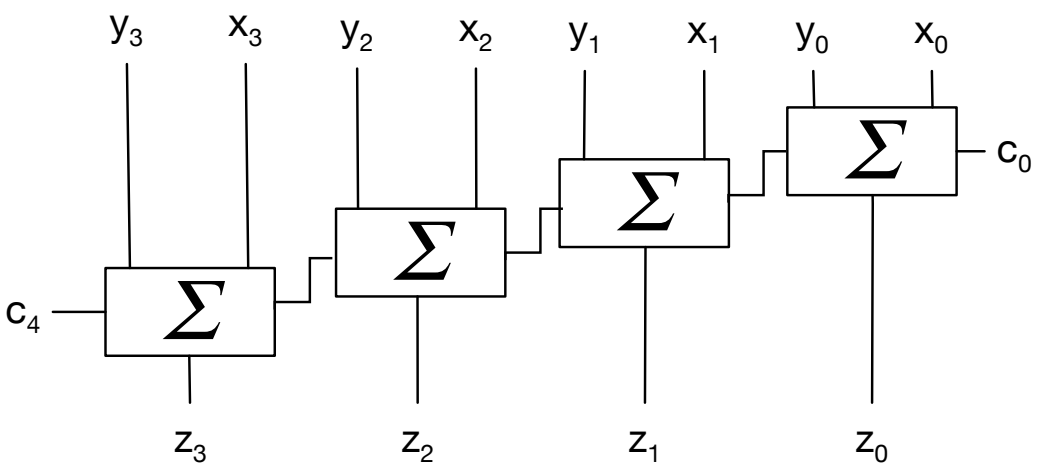
Sumator opisany w poprzednim punkcie użyjemy do sumowania dwóch wartości czterobitowych. Pierwsza wartość będzie zapisana bitami (x_3, x_2, x_1, x_0) a druga bitami (y_3, y_2, y_1, y_0). Czterobitowy wynik będzie zapisany za pomocą bitów (z_3, z_2, z_1, z_0) i bitu przeniesienia c_4, który będzie równy 1 gdy suma wartości (x_3, x_2, x_1, x_0) i (y_3, y_2, y_1, y_0) przekracza wartość 15 (maksymalna jaką można zapisać na czterech bitach (z_3, z_2, z_1, z_0)).

Układ z rysunku 2.4 będziemy dla uproszczenia rysować jako pojedyncza bramka o trzech wejściach x_i, y_i i c_i oraz dwóch wyjściach z_i i c_{i+1}:



Na rysunku 2.5 przedstawiono schemat czterobitowego sumatora kaskadowego. Nazwa jego podkreśla, że bity wyniku wyliczane są kolejno od prawej do lewej i wyliczenie kolejnego bitu z_i, dla $i > 0$, wymaga wcześniejszego policzenia przeniesienia c_i.

Na kolejnym wykładzie o architekturze mikroprocesora i programowaniu niskopoziomowym, zostanie zaprezentowane jak korzystać z bitu przeniesienia.



Rysunek 2.5: Czterobitowy sumator kaskadowy.