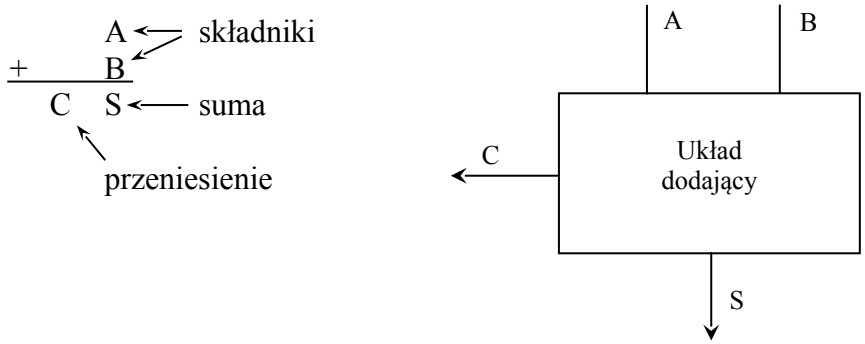
UKŁADY DODAJĄCE

1. Zasada działania dwójkowych układów dodających

Półsumator

Dodawanie liczb dwójkowych wykonuje się według tych samych zasad, jakimi posługujemy się przy dodawaniu liczb dziesiętnych. Rozpatrzmy operację dodawania jednobitowych liczb dwójkowych. Operację tę ilustruje rysunek 1.1.



rys. 1.1 **Półsumator**

Tablicę prawdy oraz tablice Karnaugha przedstawia rysunek 1.2.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

C

B \ A	0	1
0	0	0
1	0	1

S

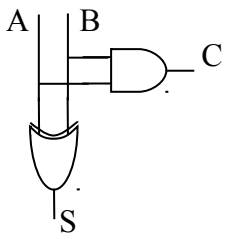
B \ A	0	1
0	0	1
1	1	0

rys. 1.2

Na podstawie tablic Karnaugha wyznaczamy funkcje opisujące sumę S i przeniesienie C:

$$S = A \cdot \overline{B} + \overline{A} \cdot B = A \oplus B$$
$$C = A \cdot B$$

Przykładowa implementacja tych funkcji za pomocą bramek przedstawiona jest na rys. 1.3.



rys. 1.3

W podobny sposób możemy otrzymać funkcje logiczne realizowane przez **półsubtraktor** (układ służący do odejmowania, realizujący A - B). Tablica prawdy oraz tablice Karnaugha dla tego układu są przedstawione na rys. 1.3a. Na ich podstawie znajdujemy funkcje opisujące różnicę D i pożyczkę V:

$$D = A \oplus B$$
$$V = \overline{A} \cdot B$$

A	B	V	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

V

B \ A	0	1
0	0	0
1	1	0

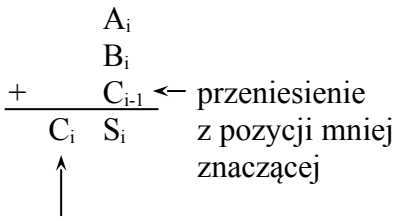
D

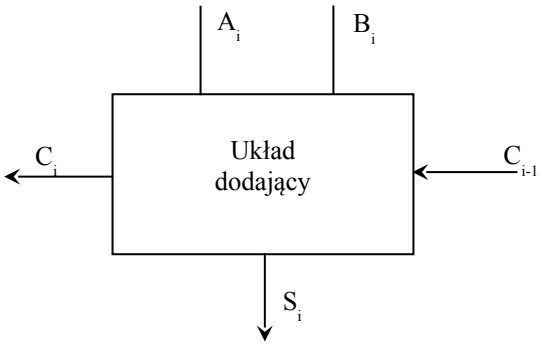
B \ A	0	1
0	0	1
1	1	0

rys. 1.3a

Pełny Sumator

W przypadku dodawania wielobitowych liczb dwójkowych należy uwzględnić przeniesienie z pozycji sąsiedniej, mniej znaczącej od rozpatrywanej. Operację dodawania bitów z i-tych pozycji dodawanych liczb wraz z tablicą prawdy i tablicami Karnaugha układu dodającego ilustruje rysunek 1.4.





A_i	B_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

C_i

Ci-1 \ AiBi	00	01	11	10
0	0	0	1	0
1	0	1	1	1

S_i

Ci-1 \ AiBi	00	01	11	10
0	0	1	0	1
1	1	0	1	0

rys. 1.4 Sumator pełny

Na podstawie tablic Karnaugha wyznaczamy funkcje S_i i C_i:

$$S_i = \overline{A_i} \cdot \overline{B_i} \cdot C_{i-1} + \overline{A_i} \cdot B_i \cdot \overline{C_{i-1}} + A_i \cdot \overline{B_i} \cdot \overline{C_{i-1}} + A_i \cdot B_i \cdot C_{i-1} = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = A_i \cdot B_i \cdot \overline{C_{i-1}} + A_i \cdot \overline{B_i} \cdot C_{i-1} + \overline{A_i} \cdot B_i \cdot C_{i-1} + A_i \cdot B_i \cdot C_{i-1} =$$
$$= A_i \cdot B_i \cdot (C_{i-1} + \overline{C_{i-1}}) + C_{i-1} \cdot (A_i \cdot \overline{B_i} + \overline{A_i} \cdot B_i) =$$
$$= A_i \cdot B_i + (A_i \oplus B_i) \cdot C_{i-1}$$

Warto zauważy, że zmienną C_i można otrzymać w postaci równania z wykorzystaniem multipleksera:
if $A_i=B_i$ then $C_i= A_i$
else $C_i= C_{i-1}$.
Przy czym do funkcji porównania można użyć operacji $A_i \oplus B_i$, która jest wykonywana do generacji S_i.

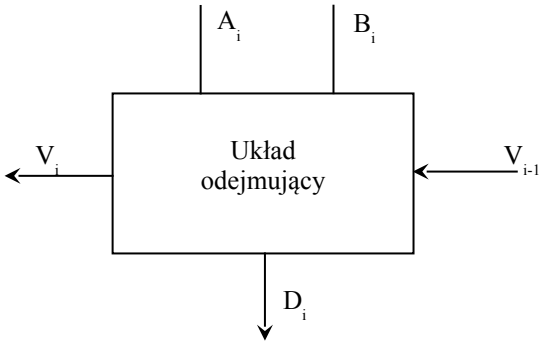
Układ realizujący takie funkcje nazywamy **jednopozycyjnym sumatorem pełnym**.
Podobnie jak dla sumatora, można wyznaczyć funkcje realizowane przez **subtraktor pełny**. Zasadę działania subtraktora ilustruje rysunek 1.5.

Funkcje D_i i V_i mają postać:

$$D_i = \overline{A_i} \cdot \overline{B_i} \cdot V_{i-1} + \overline{A_i} \cdot B_i \cdot \overline{V_{i-1}} + A_i \cdot \overline{B_i} \cdot \overline{V_{i-1}} + A_i \cdot B_i \cdot V_{i-1} = A_i \oplus B_i \oplus V_{i-1}$$

$$\begin{aligned} V_i &= \overline{A_i} \cdot \overline{B_i} \cdot V_{i-1} + \overline{A_i} \cdot B_i \cdot \overline{V_{i-1}} + \overline{A_i} \cdot B_i \cdot V_{i-1} + A_i \cdot B_i \cdot V_{i-1} = \\ &= \overline{A_i} \cdot B_i \cdot (\overline{V_{i-1}} + V_{i-1}) + (\overline{A_i} \cdot \overline{B_i} + A_i \cdot B_i) \cdot V_{i-1} = \\ &= \overline{A_i} \cdot B_i + (\overline{A_i} \oplus B_i) \cdot V_{i-1} \end{aligned}$$

$$\begin{array}{ll} & A_i \\ - & B_i \\ - & V_{i-1} \\ \hline & V_i \quad D_i \end{array}$$



A_i	B_i	V_{i-1}	D_i	V_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

V_i

V_{i-1} \ A_iB_i	00	01	11	10
0	0	1	0	0
1	1	1	1	0

D_i

V_{i-1} \ A_iB_i	00	01	11	10
0	0	1	0	1
1	1	0	1	0

rys. 1.5 Subtractor pełny

Subtractor można również otrzymać wykorzystując właściwości układów dodająco-odejmujących, które zostaną opisane na końcu tego opracowania. Odejmowanie można wtedy zastąpić dodawaniem po uprzedniej konwersji odjemnika do kodu U2, czyli zanegowanie odjemnika (każdego bitu osobno) i dodanie 1 do najmniej znaczącego bitu (ang. LSB). Dodanie 1 do LSB można przeprowadzić bezpośrednio w układzie dodającym poprzez wymuszenie 1 na C_0. Podsumowując otrzymujemy dla odejmowania:

$S_i = \overline{A_i} \oplus B_i \oplus C_{i-1}$
$C_i = \overline{A_i} \cdot B_i + (\overline{A_i} \oplus B_i) \cdot C_{i-1}$
$C_0 = 1.$

Sumatory pełne stosuje się w układach dodających szeregowych i równoległych, które będą omówione w punkcie 2, natomiast półsumatory stosuje się w układach **inkrementujących** (zwiększający o 1). Układ taki składa się z n półsumatorów połączonych kaskadowo. Na wejścia A_i podajemy liczbę, którą chcemy inkrementować. Wejścia B_i nie są potrzebne, więc używamy ich do przeniesienia z pozycji mniej znaczącej (półsumator nie ma wejścia C_{i-1}). Aby wykonać inkrementację, czyli dodać 1 do najmniej

znaczącego bitu, podajemy stan ‘1’ na wejście B pierwszego sumatora, które odegra rolę wejścia C_0. Schemat ideowy takiego układu dla liczb 4-bitowych przedstawiono na rys. 1.6. Funkcje logiczne realizowane przez układ inkrementujący otrzymujemy z funkcji półsumatora, zastępując B przez C_{i-1}, A przez A_i, S przez S_i oraz C przez C_i:

$S_i = A_i \oplus C_{i-1}$

$C_i = A_i \cdot C_{i-1}$

$C_0 = 1$

Inkrementator równoległy

Inkrementator równoległy działa podobnie jak szeregowy ale nie ma szeregowej propagacji przeniesień. Logikę inkrementatora równoległego można otrzymać bezpośrednio z logiki inkrementatora szeregowego (podstawiając zmienne do równań: $S_i = A_i \oplus C_{i-1}$; $C_i = A_i \cdot C_{i-1}$; $C_0 = 1$):

$S_1 = A_1 \oplus C_0 = \overline{A_i}$

zmienna pomocnicza: $C_1 = A_1$

$S_2 = A_2 \oplus C_1 = A_2 \oplus A_1$

zmienna pomocnicza: $C_2 = A_1 \cdot A_2$

$S_3 = A_3 \oplus (A_1 \cdot A_2)$

zmienna pomocnicza $C_3 = A_1 \cdot A_2 \cdot A_3$

$S_4 = A_4 \oplus (A_1 \cdot A_2 \cdot A_3)$

itd.

Inkrementator ten ma krótszy czas propagacji jednakże okupiony bardziej skomplikowanym układem.

Stosując podobną metodę można otrzymać układ pełnego sumatora równoległego.

$S_1 = A_1 \oplus B_1$

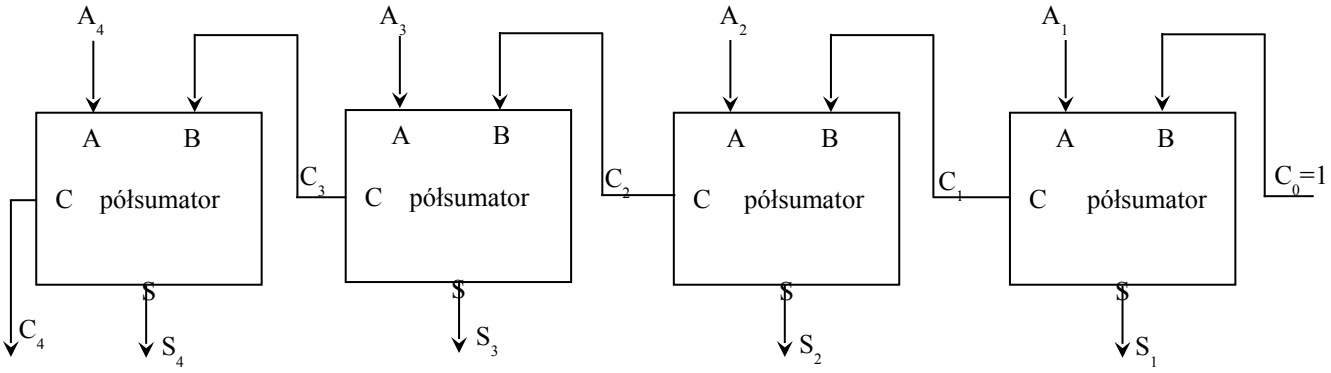
Zmienna pomocnicza $C_1 = A_1 \cdot B_1$

$S_2 = A_2 \oplus B_2 \oplus (A_1 \cdot B_1)$

$C_2 = A_2 \cdot B_2 + (A_2 \oplus B_2) \cdot A_1 \cdot B_1$

itd.

Układ dodający powstały w ten sposób (po dodatkowym wprowadzeniu zmiennych pomocniczych) jest nazywany Carry-Look-Ahead i zostanie omówiony w końcowej części tego opracowania



rys. 1.6 Układ inkrementujący (szeregowy)

Podobnie jak układ inkrementujący, można stworzyć układ **dekrementujący**. Funkcje realizowane przez taki układ otrzymujemy z funkcji półsubtraktora, zastępując odpowiednie zmienne :

$D_i = A_i \oplus V_{i-1}$

$V_i = \overline{A}_i \cdot V_{i-1}$

$V_0 = 1$

Układ dekrementujący można zaimplementować za pomocą półsubtraktorów – w taki sam sposób jak układ inkrementujący za pomocą półsumatorów. Innym rozwiązaniem jest dekrementacja przy użyciu półsumatorów – należy wtedy **na wejścia A_i inkrementera podać negację liczby** (za pomocą bramek NOT), którą chcemy dekrementować, a na wyjściu otrzymamy również zaprzeczenie wyniku. Taki

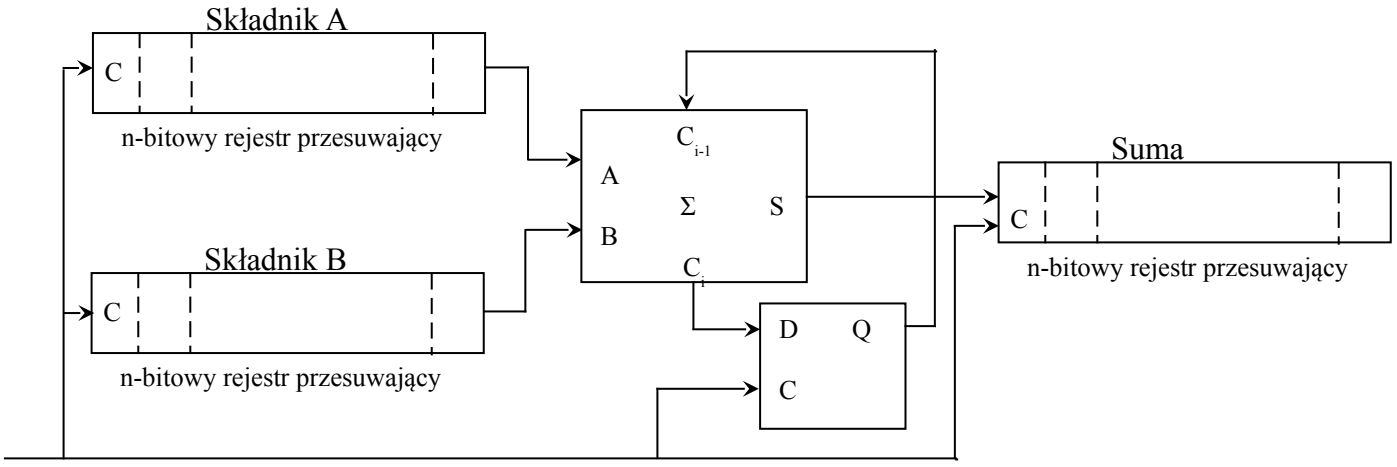
sposób postępowania wynika z porównania funkcji logicznych realizowanych przez inkrementer i dekrementer.

2. Układy dodające szeregowe i równoległe

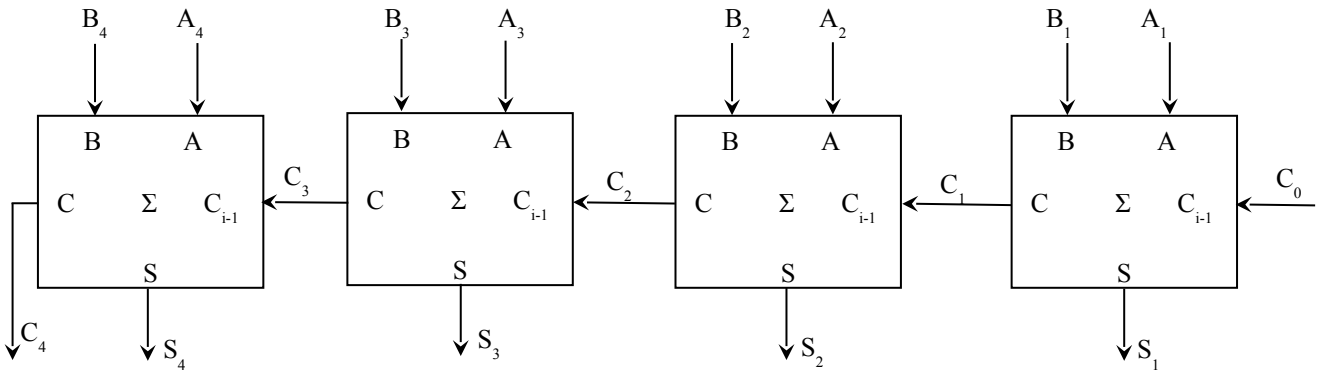
Dodawanie liczb dwójkowych można zrealizować szeregowo lub równolegle, stąd podział sumatorów na szeregowe i równoległe.

W sumatorze **szeregowym** (ang. S*erial Adder)* w każdym kroku dodawane są, poczynając od pozycji najmniej znaczącej, dwa bity składników oraz bit przeniesienia z poprzedniej pozycji. Przeniesienie z poprzedniej pozycji jest zapamiętywane, np. za pomocą przerzutnika D. Schemat ideowy takiego sumatora jest przedstawiony na rysunku 2.1. Zaletami sumatorów szeregowych są ich prostota i mała liczba układów potrzebnych do realizacji. Natomiast wadą jest mała szybkość działania – czas trwania dodawania składników n-bitowych wynosi nT (T - takt zegara). Sumatory szeregowe i ich zastosowania zostaną dokładnie omówione w ramach laboratorium, którego tematem będą rejestry przesuwne.

Do realizacji dodawania **równoległego** potrzebny jest sumator wielopozycyjny, który można otrzymać łącząc ze sobą odpowiednio szereg sumatorów jednopozycyjnych. Najprostszym przykładem sumatora równoległego jest sumator kaskadowy (sumator równoległy z przeniesieniami szeregowymi – ang. *Ripple -Carry Adder*), którego schemat ideowy jest przedstawiony na rysunku 2.2. Poszczególne pary bitów są dodawane za pomocą osobnych sumatorów, przy czym generowane na danej pozycji przeniesienie jest kierowane do sumatora pozycji następnej. Jeśli przeniesienie pojawiające się na dowolnej pozycji oddziaływałoby tylko na następną pozycję, wówczas dodawanie takie trwałoby bardzo krótko. Niestety przeniesienie może propagować nawet na całe słowo (np. dla 11111+11111), więc dodawanie równoległe z przeniesieniami szeregowymi nie będzie tak szybkie, jakby się to mogło wydawać.



rys. 2.1 Sumator szeregowy (S*erial Adder*)



rys. 2.2 Sumator równoległy z przeniesieniami szeregowymi (*Ripple - Carry Adder*)

Istnieją specjalne konstrukcje sumatorów równoległych umożliwiające przyspieszenie dodawania. Do najczęściej obecnie wykorzystywanych należy sumator z przeniesieniami jednoczesnymi (równoległymi – ang. *Carry-Lookahead Adder*). W sumatorze takim wszystkie przeniesienia są wytwarzane jednocześnie, na podstawie wartości bitów sumowanych składników i przeniesienia początkowego.
Oznaczmy przez G_i warunek generacji przeniesienia przez i-tą pozycję:

$$G_i = A_i \cdot B_i$$

Warunek $G_i = 1$ oznacza, że przeniesienie C_i wychodzące z tej pozycji jest równe 1, bez względu na wartość przeniesienia C_{i-1} przychodzącego na tę pozycję.
Oznaczmy przez P_i warunek propagacji przeniesienia przez i-tą pozycję:

$$P_i = A_i \oplus B_i$$

Warunek $P_i = 1$ oznacza, że $C_i = C_{i-1}$
Przeniesienie z i-tej pozycji sumatora można więc przedstawić w postaci :

$$C_i = G_i + P_i \cdot C_{i-1}$$

Funkcja sumy generowana przez i-tą pozycję może być przedstawiona w postaci :

$$S_i = P_i \oplus C_{i-1}$$

W celu zilustrowania wyników dotychczasowych rozważań, rozpatrzmy implementację czteropozycyjnego sumatora z przeniesieniami jednoczesnymi. Funkcje sumy i przeniesienia takiego sumatora przyjmą postać:

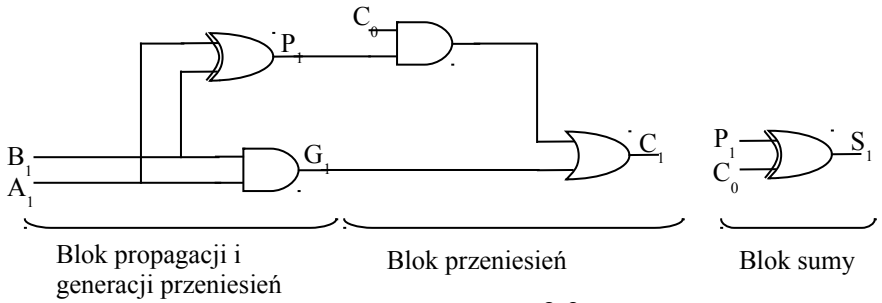
$$S_1 = P_1 \oplus C_0$$
$$S_2 = P_2 \oplus C_1$$
$$S_3 = P_3 \oplus C_2$$
$$S_4 = P_4 \oplus C_3$$

$$C_1 = G_1 + P_1 \cdot C_0$$
$$C_2 = G_2 + P_2 \cdot C_1 = G_2 + P_2 \cdot (G_1 + P_1 \cdot C_0) = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot C_0$$
$$C_3 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot C_0$$
$$C_4 = G_4 + P_4 \cdot C_3 = G_4 + P_4 \cdot G_3 + P_4 \cdot P_3 \cdot G_2 + P_4 \cdot P_3 \cdot P_2 \cdot G_1 + P_4 \cdot P_3 \cdot P_2 \cdot P_1 \cdot C_0$$

Z powyższego wynika, że sumator składa się z trzech bloków:

- bloku realizującego funkcje $A_i \cdot B_i$ i $A_i \oplus B_i$
- bloku sumy
- bloku przeniesień

Realizację fragmentu takiego układu (części obliczającej S_1 i C_1) przedstawiono na rysunku 2.3.

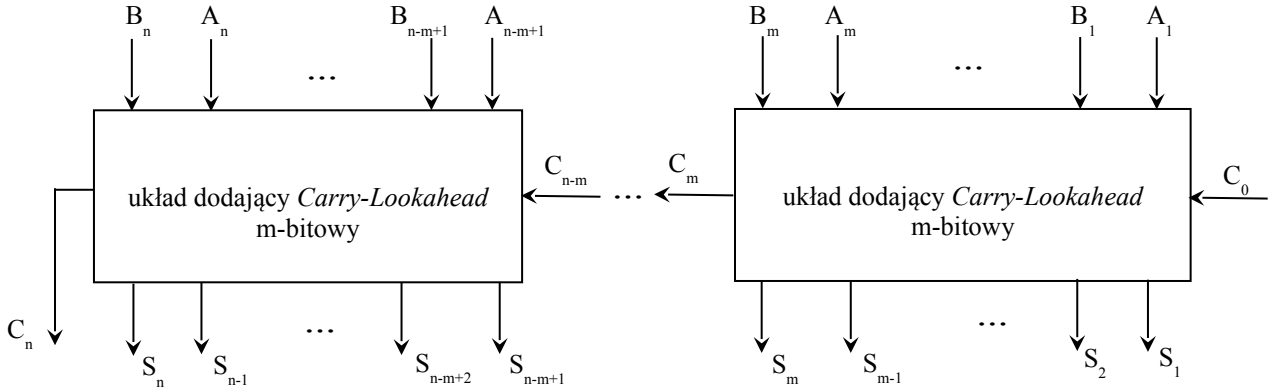


rys. 2.3

Sumatory równoległe z przeniesieniami równoległymi są szybkie, ale ich wadą jest znaczne skomplikowanie. Układy takie są kosztowne i trudne w realizacji, gdyż wszystkie sygnały wejściowe i generowane w trakcie dodawania muszą być podawane na wejścia wielu bramek, a bramki te z kolei mają wraz ze wzrostem liczby bitów coraz więcej wejść. Z praktycznego punktu widzenia nie da się więc stosować czystych układów typu *Carry - Lookahead* do dodawania dużych liczb.

Pewnym rozwiązaniem problemu układów równoległych z przeniesieniami równoległymi są sumatory blokowe szeregowo-równoległe (ang. *Ripple-Block Carry-Lookahead Adder*). Układ taki składa się z

kilku lub kilkunastu bloków, z których każdy jest niezależnym sumatorem typu *Carry – Lookahead.* Każdy blok wykonuje dodawanie określonej, zazwyczaj niewielkiej liczby bitów składników. Przeniesienia generowane między poszczególnymi blokami są propagowane szeregowo jak w zwykłym sumatorze typu *Ripple-Carry*. Sumator blokowy przedstawiono na rysunku 2.4.



rys. 2.4 Sumator blokowy (*Ripple-Block Carry-Lookahead Adder*)

Istnieje również wiele innych typów układów blokowych (np. na bazie bloków typu szeregowego połączonych jak w sumatorze z przeniesieniami równoległymi itp.). Wszystkie one realizują generalną ideę łączenia szybkości przeniesień równoległych z prostotą przeniesień szeregowych. Pamiętajmy jednak, że żaden układ szeregowo-równoległy nie może być tak szybki jak czysty układ równoległy *Carry-Lookahead.*

3. Układy dodająco – odejmujące w kodzie U1 i U2

Odejmowanie i dodawanie można zaimplementować w jednym układzie dodającym, pod warunkiem, że układ ten będzie miał możliwość dodawania liczb ze znakiem (odejmowanie jest dodawaniem liczby przeciwnej).

Liczby dwójkowe ze znakiem są przedstawiane na trzy sposoby:

- podstawowy: znak, moduł
- kod U1: (znak), uzupełnienie do 1
- kod U2: (znak) uzupełnienie do 2 (najczęściej stosowany w układach dodających)

W każdym z wymienionych zapisów znak liczby reprezentuje pierwszy bit : ‘0’ reprezentuje znak plus, natomiast ’1’ znak minus. Postać liczb dodatnich jest w każdym zapisie taka sama, natomiast postać liczb ujemnych jest różna:

- w zapisie podstawowym wartość bezwzględna liczby ujemnej jest przedstawiana w naturalnym kodzie dwójkowym
- w kodzie U1 uzupełniamy wartość bezwzględną liczby ujemnej do 1, tzn. w naturalnym zapisie dwójkowym zamieniamy zera na jedynki, a jedynki na zera
- w kodzie U2 uzupełniamy wartość bezwzględną liczby ujemnej do 2, tzn. w naturalnym zapisie dwójkowym zamieniamy zera na jedynki, jedynki na zera i dodajemy 1 do najmniej znaczącego bitu (jeśli w wyniku dodawania otrzymamy przeniesienie propagujące aż do najbardziej znaczącej pozycji, to jego wartość na tej pozycji jest ignorowana)

Poniższy przykład jest ilustracją wszystkich trzech sposobów zapisu liczb ujemnych (pierwszy bit za każdym razem jest bitem znaku, w tym przypadku ‘1’ pokazuje, że mamy do czynienia z liczbą ujemną):

-13 = 1 1 1 0 1 „znak, moduł”
= 1 0 0 1 0 „znak, uzupełnienie do 1” – kod U1
\+ 1
= 1 0 0 1 1 „znak, uzupełnienie do 2” – kod U2

Realizacja układu dodająco-odejmującego dla liczb w zapisie „znak, moduł” jest złożona i dlatego nie jest stosowana. Jeśli natomiast wykorzystamy kod uzupełnieniowy, to odejmowanie można zastąpić dodawaniem uzupełnienia. Unikniemy dzięki temu stosowania subtraktorów, ale trzeba użyć układów uzupełniających, czyli **komplementerów**.

Rozważmy komplementer słów czterobitowych. Oznaczmy przez *W, X, Y, Z* bity słowa wyjściowego, a przez *w, x, y, z* bity słowa wejściowego i zastosujmy dodatkowe wejście programujące *s* (jeżeli *s* jest w stanie ‘1’ układ wykonuje uzupełnienie do 1, a jeżeli *s* jest w stanie ‘0’ - na wyjściach otrzymujemy słowo wejściowe bez żadnej zmiany). Komplementer taki jest opisany następującymi funkcjami:

$W = w \oplus s$
$X = x \oplus s$
$Y = y \oplus s$
$Z = z \oplus s$

Układ uzupełniający do 2 najlepiej zrealizować używając komplementera do 1 i sumatora (inkrementatora, w celu dodania 1). Warto zwrócić uwagę, że operację inkrementacji bardzo często można przeprowadzić w następnym sumatorze poprzez wymuszenie 1 na C_0. Sumator ten może być równocześnie wykorzystany do wykonania operacji dodawania (odejmowania).

Poniższy przykład ilustruje, w jaki sposób można zastąpić odejmowanie dodawaniem uzupełnienia.

Wykonajmy klasyczne odejmowanie:

```
  1 0 0
- 0 1 1
-------
  0 0 1
```

A teraz zastąpmy odjemnik uzupełnieniem do 1 i przeprowadźmy dodawanie:

```
  1 0 0
+ 1 0 0
-------
(1) 0 0 0
+       1
-------
  0 0 1
```

Aby otrzymać prawidłowy wynik, przeniesienie otrzymane na najstarszej pozycji należy dodać do najmniej znaczącego bitu.
takie przeniesienie nazywamy przeniesieniem cyklicznym (ang. *End-Around Carry*).

Jeśli natomiast zastąpimy odjemnik uzupełnieniem do 2, przeniesienie cykliczne należy pominąć:

```
  1 0 0
+ 1 0 1
-------
(1) 0 0 1
```

Zauważmy, że zastosowanie kodu uzupełnieniowego umożliwia również bezpośrednie dodawanie liczb ze znakiem (razem z bitem znaku), przy czym wynik otrzymujemy również w odpowiednim kodzie uzupełnieniowym. Sposób postępowania jest identyczny jak poprzednio – w przypadku kodu U1 uwzględniamy przeniesienie cykliczne, natomiast w przypadku kodu U2 pomijamy je.

Prześledźmy dla przykładu dodawanie –3 + (–4) = –7.

Aby zastosować kod U1 znajdujemy reprezentację liczby –3 w tym kodzie: 1 1 0 0 a także reprezentację liczby –4 : 1 0 1 1 (pierwszy bit jest bitem znaku).
Następnie przeprowadzamy dodawanie, pamiętając o przeniesieniu cyklicznym:

```
    1 1 0 0
+   1 0 1 1
  ① 0 1 1 1
+           1
  1 0 0 0
```

Liczba 1 0 0 0 jest reprezentacją w kodzie U1 liczby ujemnej o wartości bezwzględnej 7 czyli liczby –7.

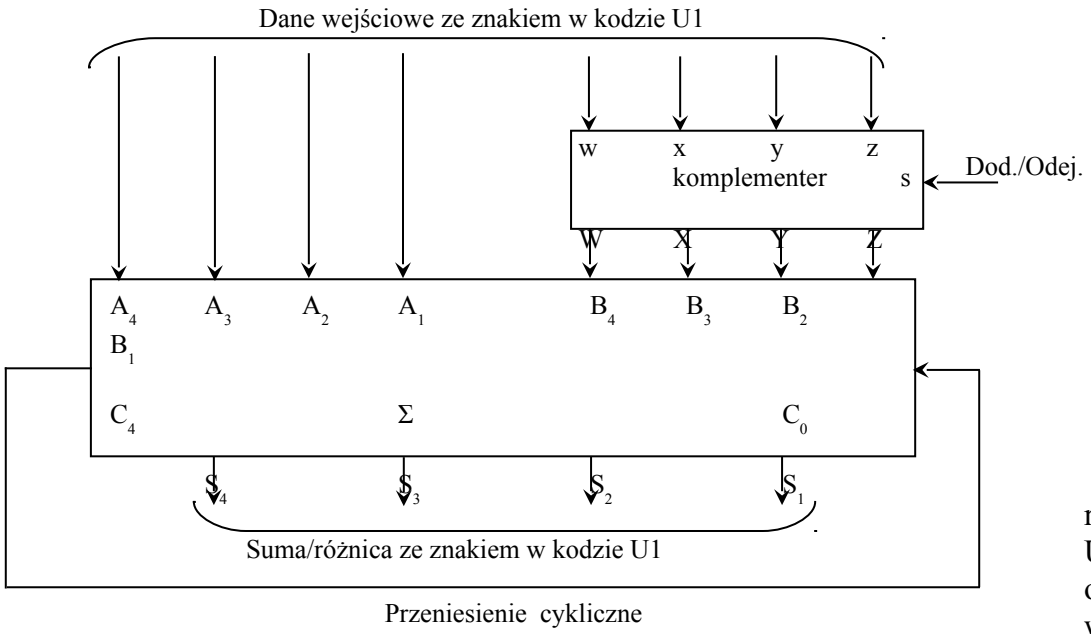
Aby zastosować kod U2, postępujemy analogicznie. Reprezentacje liczb –3 i –4 to 1 1 0 1 i 1 1 0 0.
Przeprowadzamy dodawanie:

```
    1 1 0 1
+   1 1 0 0
  ✕ 1 0 0 1
```

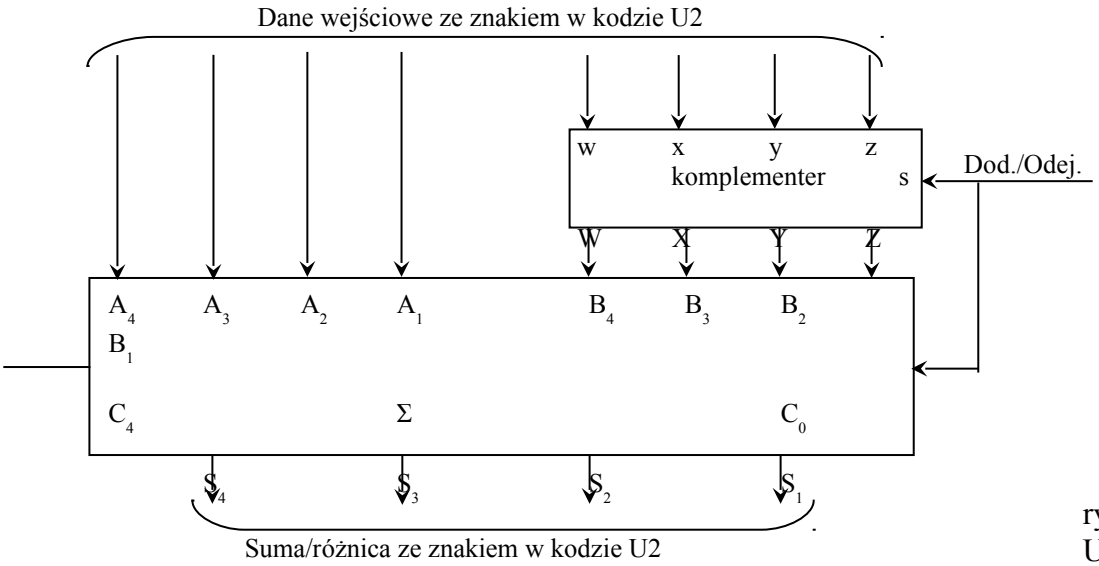
Otrzymany wynik jest reprezentacją liczby –7 w kodzie U2.

W analogiczny sposób możemy przeprowadzić odejmowanie liczb ze znakiem w kodzie U1 lub U2, pamiętając, że odejmowanie sprowadza się do dodania odpowiedniego uzupełnienia. Proces uzupełniania wykonujemy na wszystkich bitach liczby (łącznie z bitem znaku).

Możemy więc zaimplementować w jednym układzie dodawanie oraz odejmowanie liczb ze znakiem. Układ taki dla liczb czterobitowych ma 9 wejść: po cztery wejścia dla każdej liczby (razem z bitem znaku) oraz dodatkowe wejście *Dod./Odej.* Jeśli to dodatkowe wejście jest w stanie ‘0’ układ wykonuje dodawanie, a jeśli w stanie ‘1’ – odejmowanie. Na wejścia układu podajemy liczby w kodzie U1 lub U2, na wyjściu otrzymujemy sumę lub różnicę, też w odpowiednim kodzie. Schematy ideowe takich układów przedstawione są na rysunkach 3.1 i 3.2.



rys. 3.1
Układ dodająco – odejmujący w kodzie U1



rys. 3.2
Układ dodająco – odejmujący w kodzie U2

W obydwóch układach, wejście *Dod./Odej.* jest używane do uaktywnienia (lub nie) komplementera. Jeśli wejście to jest w stanie '0', to zgodnie z implementacją komplementera przedstawioną powyżej, układ zwraca liczbę na wyjścia bez zmian, czyli wykonujemy dodawanie liczb ze znakiem. Jeżeli natomiast wejście *Dod./Odej.* jest w stanie '1', następuje uzupełnienie do 1, a potem dodawanie, czyli wykonujemy odejmowanie. Warto również zwrócić uwagę na wykorzystanie wejścia C_0 układu dodającego. W przypadku układu pracującego w kodzie U1, wejście to zostało użyte do przeniesienia cyklicznego, a w przypadku układu pracującego w kodzie U2 – do wykonania uzupełnienia do 2. W obydwóch wypadkach podanie '1' na wejście C_0 sprowadza się do dodania 1 do najmniej znaczącego bitu.

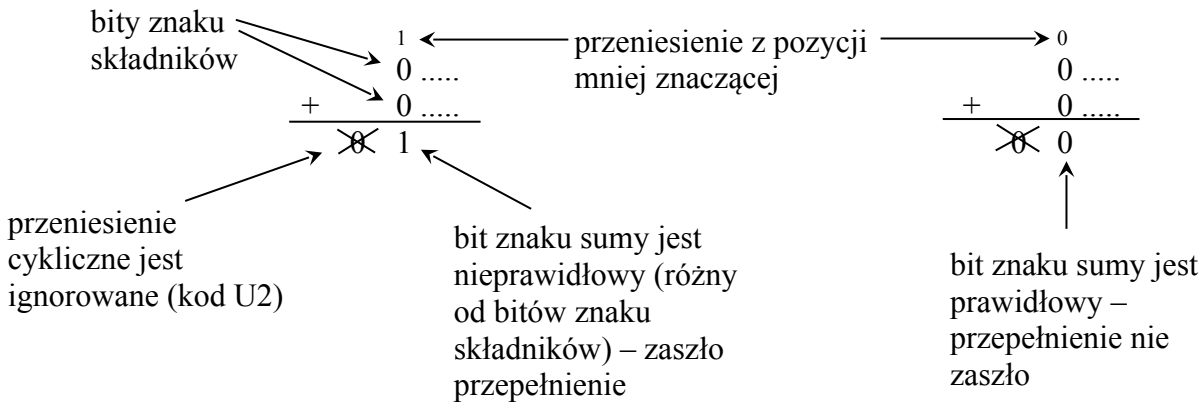
Warto na zakończenie zauważyć, że nasz układ dodający lub odejmujący liczby ze znakiem może zadziałać niepoprawnie. Jeśli np. wykonujemy dodawanie liczb dodatnich i na wejścia podamy zbyt duże liczby, to generowane przeniesienie może propagować aż na pozycję znaku i otrzymamy liczbę ujemną czyli błędny wynik. Zjawisko to nazywamy **przepełnieniem** (ang. *Overflow*). Wynik dodawania dwóch liczb dodatnich A i B będzie poprawny tylko wtedy gdy $A + B < 2^{n-1}$ (n – liczba bitów składników i wyniku razem z bitem znaku). Sytuacja będzie symetryczna w przypadku dodawania dwóch liczb ujemnych zbyt dużych co do wartości bezwzględnej. Otrzymamy wtedy w wyniku liczbę dodatnią. Dodawanie dwóch liczb ujemnych –C i –D (C, D > 0) da poprawny wynik wtedy, gdy $C + D < 2^{n-1}$. Poniżej przedstawiono dwa przykłady dodawania z przepełnieniem w przypadku składników dodatnich i ujemnych (dodawanie przeprowadzono w kodzie U1).

```
 7 =        0 0 1 1 1          –11=        1 0 1 0 0
11=   +     0 1 0 1 1          –6 =   +    1 1 0 0 1
        (0) 1 0 0 1 0                  (1) 0 1 1 0 1
                    0                              1
–13=        1 0 0 1 0          14 =        0 1 1 1 0
```

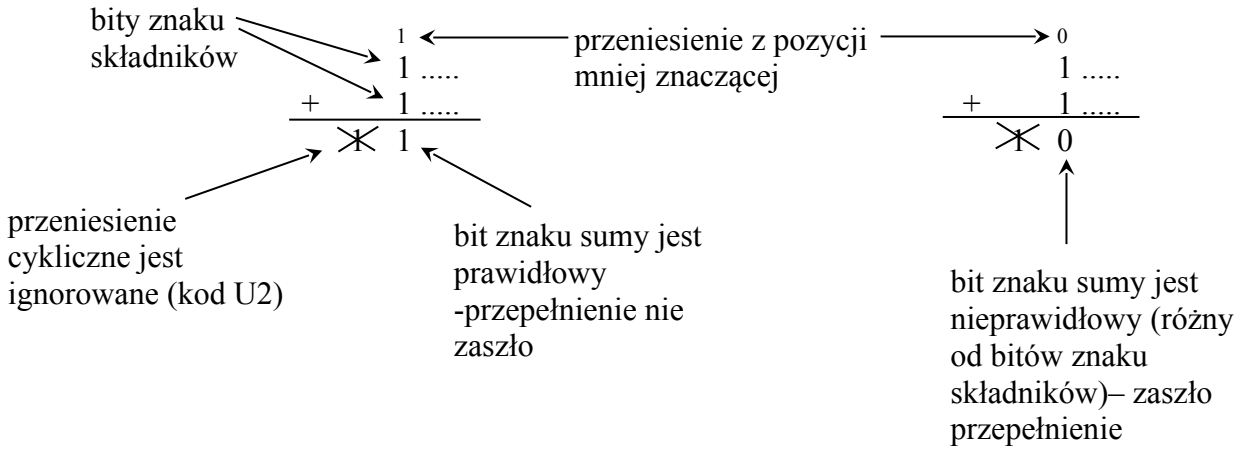
Przepełnienie można łatwo wykryć – wystarczy skontrolować bit znaku wyniku. Jeżeli otrzymamy błędny znak (np. ujemny w przypadku dodawania liczb dodatnich) – doszło do przepełnienia. Jeżeli bit znaku jest prawidłowy – dodawanie przebiegło poprawnie. W przypadku dodawania liczb różnych znaków przepełnienie nie może wystąpić (niemożliwe jest, aby wynik był za duży co do wartości bezwzględnej).

Na poniższym przykładzie prześledzimy warunek zaistnienia przepełnienia dla dodawania w kodzie U2. Przedstawiono tylko wartości najstarszych bitów (bitów znaku) dodawanych liczb, bit przeniesienia z pozycji mniej znaczącej niż pozycja znaku oraz bit przeniesienia generowanego na pozycji znaku. Mamy następujące sytuacje, w których przepełnienie mogłoby zaistnieć:

dla liczb dodatnich:



dla liczb ujemnych:



Możemy zauważyć, że warunek zaistnienia przepełnienia jest następujący:
Overflow = $C_n \oplus C_{n-1}$
gdzie C_n jest przeniesieniem generowanym na najstarszej pozycji, a C_{n-1} – przeniesieniem przychodzącym na tę pozycję.

Porównanie dwóch liczb

Porównanie dwóch liczb z wartością stałą można przeprowadzić bezpośrednio z tabeli prawdy. W szczególności jeżeli mamy znaleźć równości: poprzez bramkę AND, w której bity danej porównywanej są w postaci prostej lub zanegowanej w zależności od stanu bitu składowej stałej. Dla przykładu dla liczby 6= 0110_2 otrzymujemy: R= $\overline{A_3} A_2 A_1 \overline{A_0}$. W przypadku wykrywania operacji >, < (operacja ≤ jest równoważna zanegowanej operacji >) można wykorzystać funkcje logiczne i tabelę Karnaugh. Alternatywną metodą (stosowaną przy większej niż 4-6 bitów szerokości danej wejściowej) jest zastosowanie układu odejmującego i odczytaniu znaku wyniku – odejmowanie można uprościć poprzez zastosowanie uproszczeń logiki ponieważ jedno wejście ma wartość stałą.

Dla porównywania dwóch danych operację równości można przeprowadzić za pomocą iloczynu:

$$R= NOT[(A_i \oplus B_i) + ... + (A_0 \oplus B_0)]$$

Operację <, > przeprowadza się za pomocą układów odejmujących i sprawdzenia wyniku operacji. Warto zauważyć że łatwo daje się sprawdzić wynik, który jest mniejszy od zera – bit znaku równa się 1. W przypadku wyniku większego od zera trzeba sprawdzić czy wynik jest różny od zero (sumować logicznie wszystkie bity) i dodatkowo bit znaku=0. Dlatego podczas porównań należy dążyć do porównywania z wartościami mniejszymi od zera. Dla przykładu dla operacji A≥B należy dokonać operacji A-B - wynikiem jest zanegowany bit znaku. Dla operacji A>B należy dokonać operacji B-A – wynikiem jest bit znaku.