

Sławomir Kulesza

Technika cyfrowa

Układy arytmetyczne

Wykład dla studentów III roku Informatyki

Wersja 1.0, 05/10/2010

Układy arytmetyczne

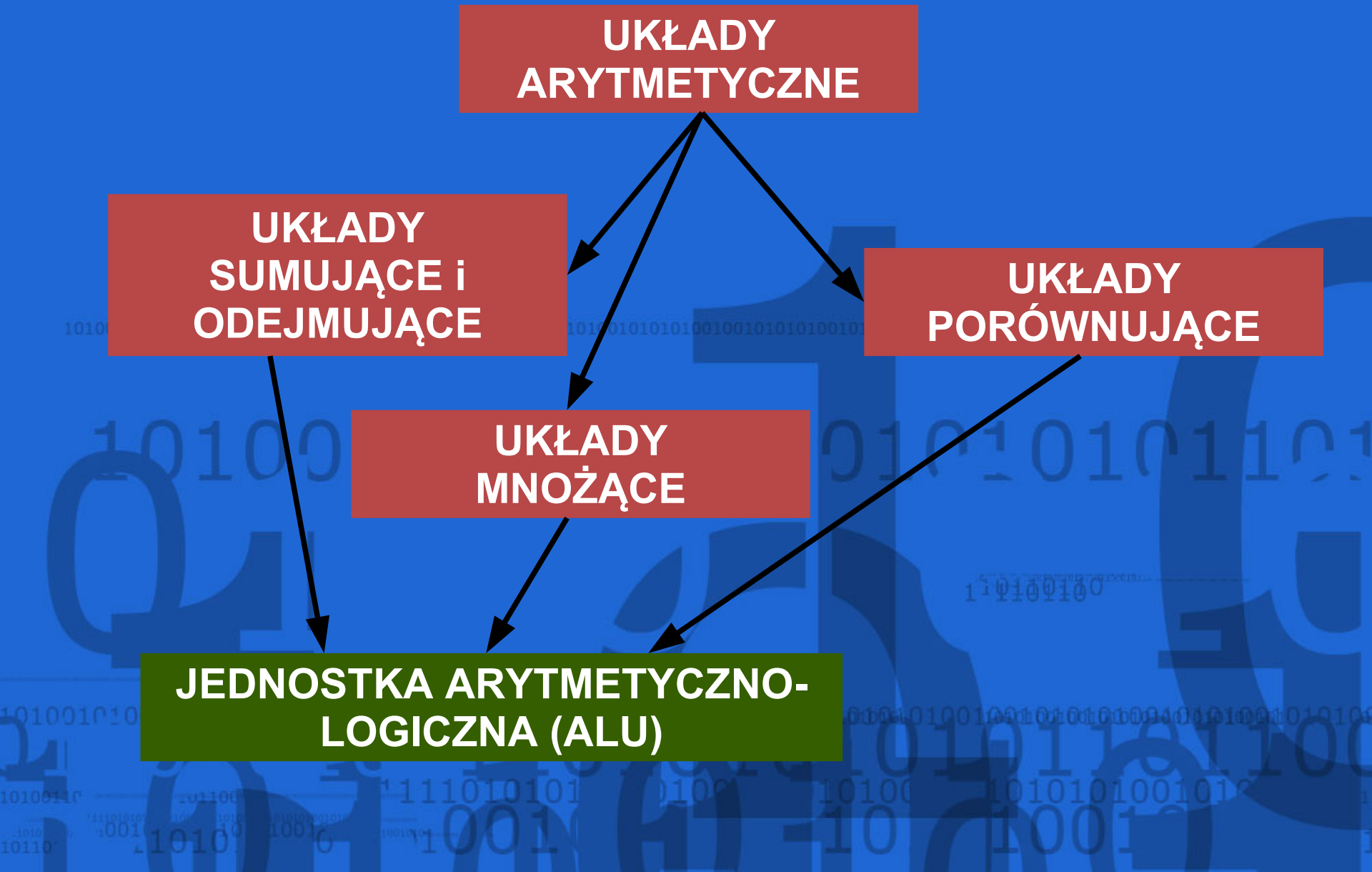
**UKŁADY
ARYTMETYCZNE**

**UKŁADY
SUMUJĄCE i
ODEJMUJĄCE**

**UKŁADY
PORÓWNUJĄCE**

**UKŁADY
MNOŻĄCE**

**JEDNOSTKA ARYTMETYCZNO-
LOGICZNA (ALU)**



Dodawanie liczb binarnych (bez znaku)

Reguły dodawania liczb dwójkowych na pozycji i ($i = 0, 1, \dots, n-1$)

| c_i | p_i | q_i | s_i | c_{i+1} |
|-------|-------|-------|-------|-----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

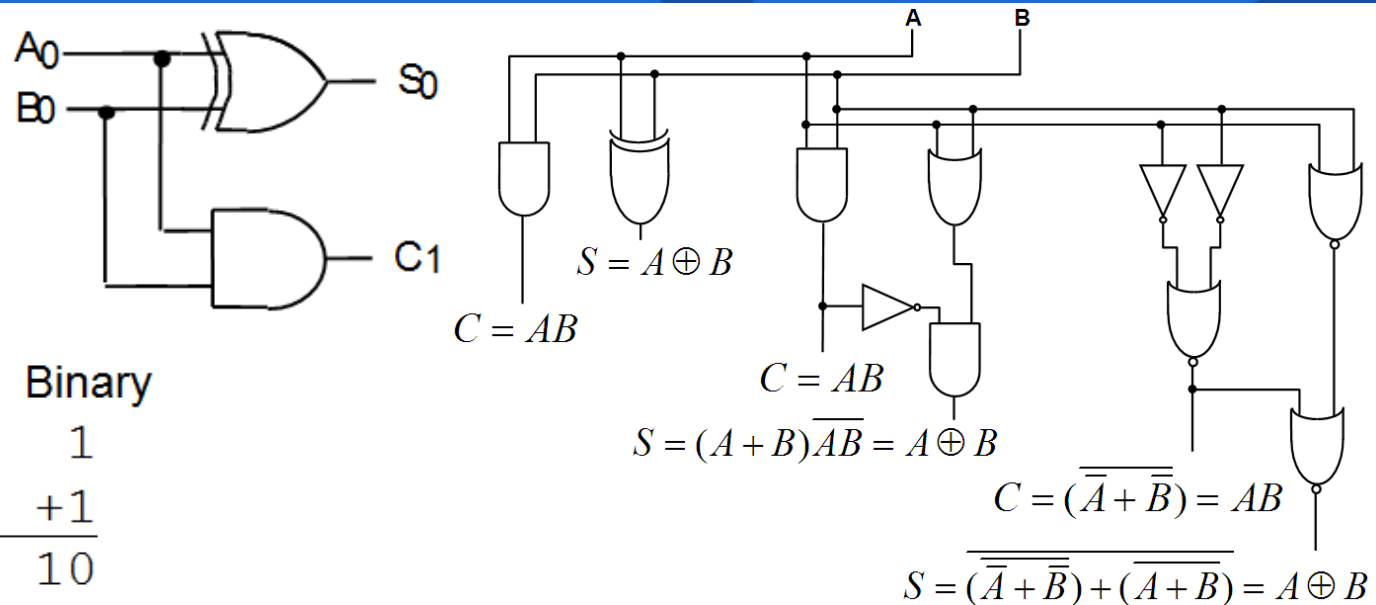
◀ suma modulo 2: $s_i = p_i \oplus q_i$

Półsumator 1-bitowy

Półsumator jest układem sumującym nie uwzględniającym bitu przeniesienia wstępnego (z pozycji niższej).

| A ₀ | B ₀ | S ₀ | C ₁ |
|----------------|----------------|----------------|----------------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

| Dec | Binary |
|-----|--------|
| 1 | 1 |
| +1 | +1 |
| 2 | 10 |



Pełny sumator 1-bitowy

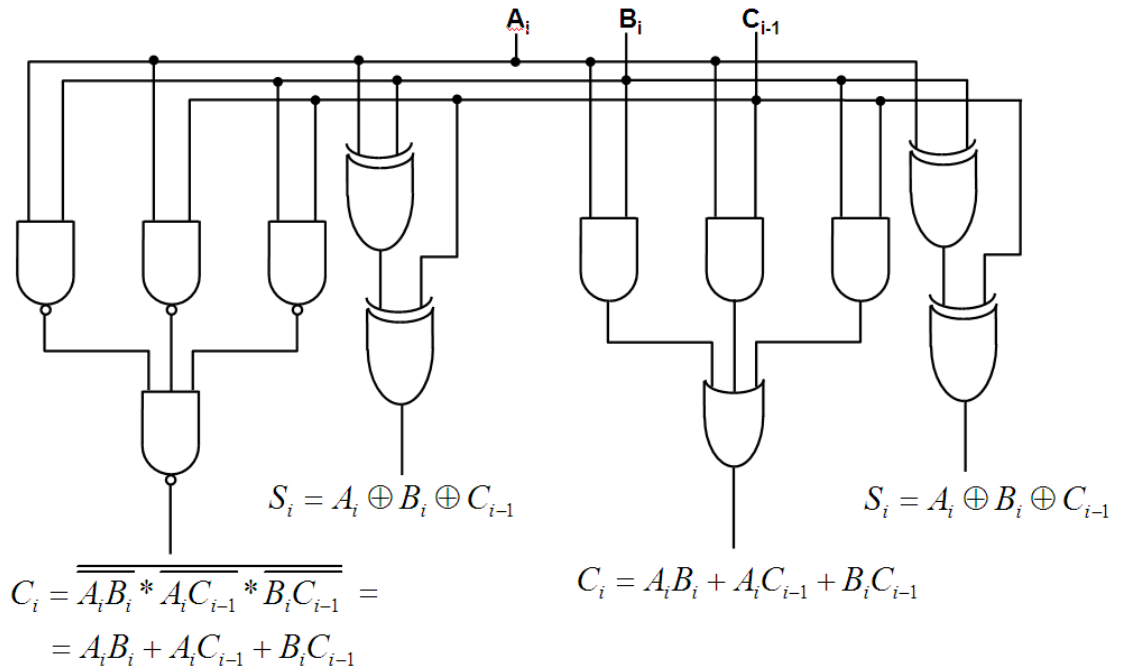
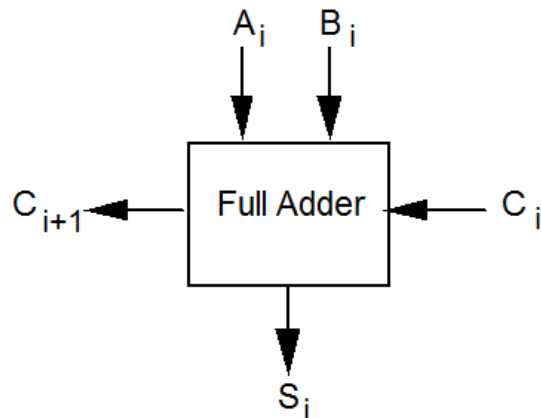
Pełny sumator jest układem uwzględniającym bit przeniesienia wstępnego c_i (z pozycji niższej) i bit przeniesienia na pozycję wyższą c_{i+1} .

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

| C_i | $A_i B_i$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| 0 | | | 1 | | 1 |
| 1 | | 1 | | 1 | |

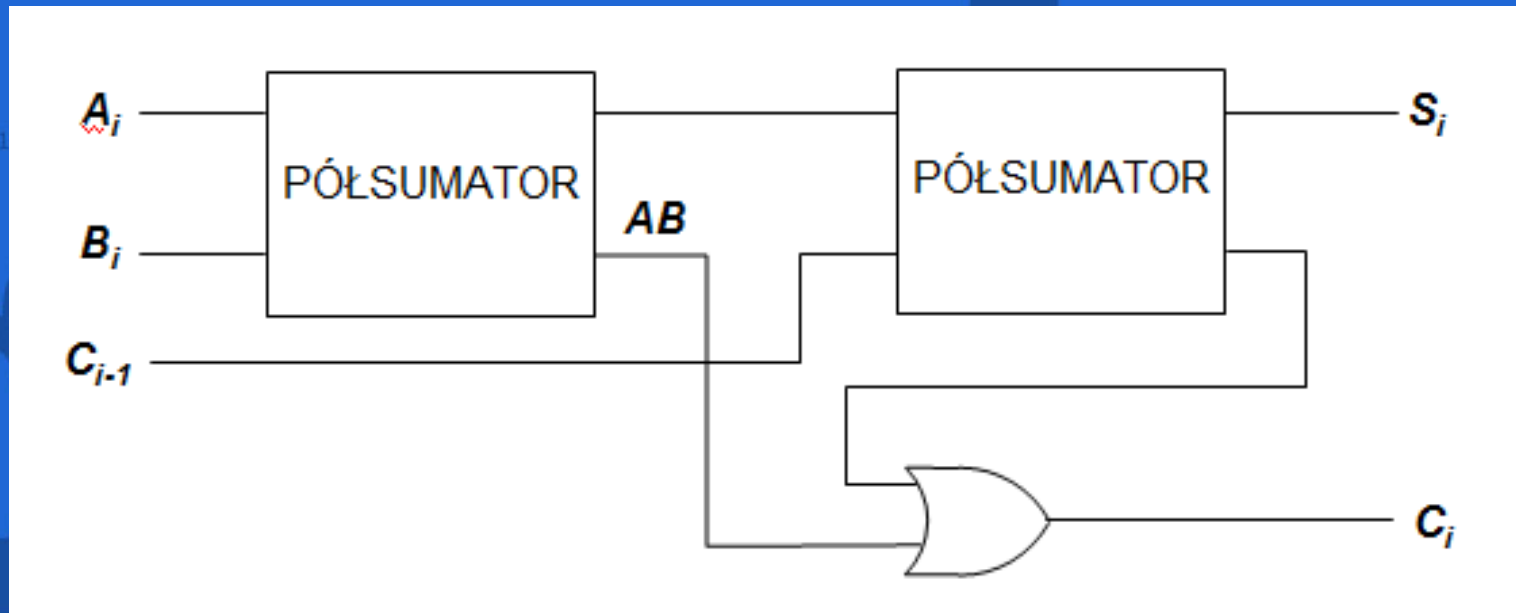
| C_i | $A_i B_i$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| 0 | | | | 1 | |
| 1 | | | 1 | 1 | 1 |

C_{i+1}

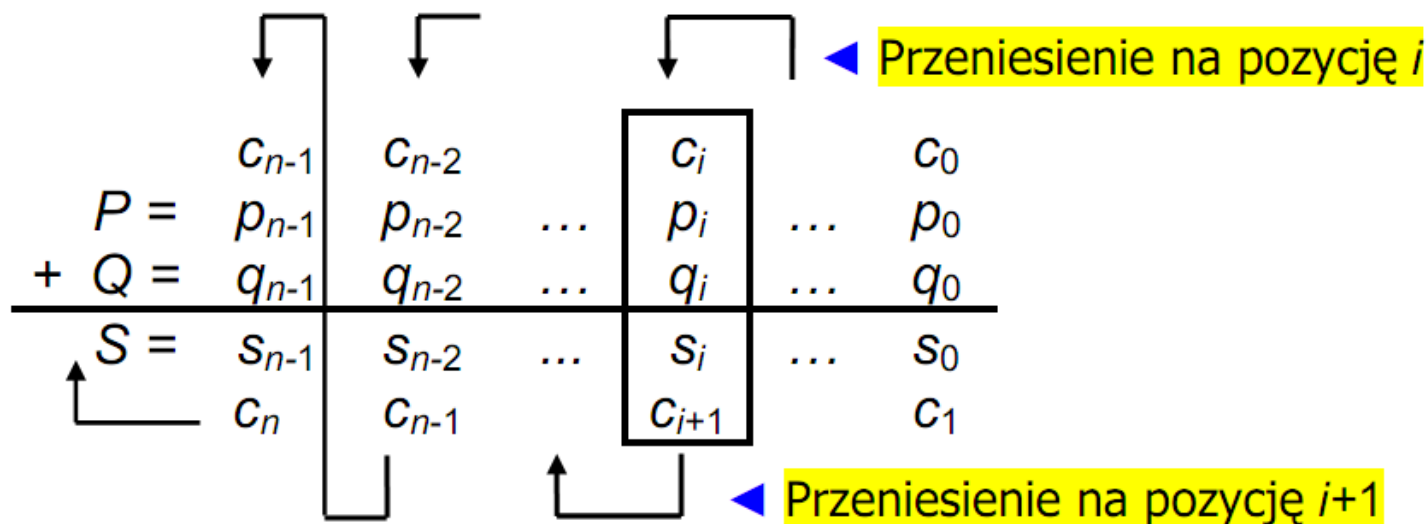


Pełny sumator 1-bitowy

Pełny sumator powstaje z połączenia dwóch półsumatorów.



Pełny sumator wielobitowy



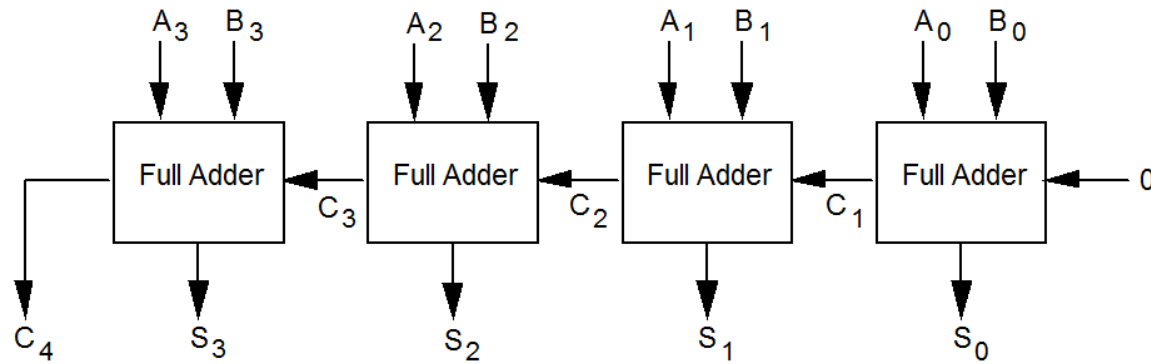
Wynik Y dodawania liczb P i Q oraz przeniesienia wstępnego c_0 :

$$P + Q + c_0 = Y = c_n \circ S \quad \text{czyli} \quad L(Y) = 2^n c_n + L(S)$$

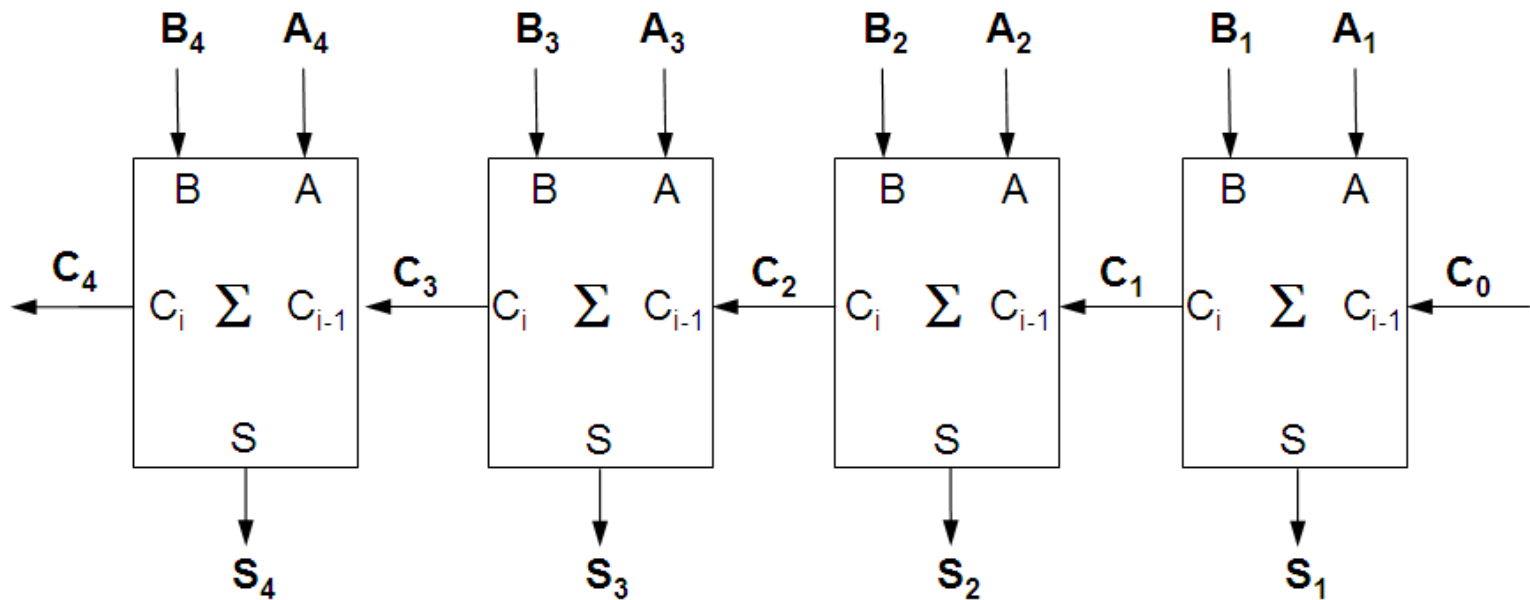
Przykład:

| Pozycja | 4 | 3 | 2 | 1 | 0 | |
|----------------|------------|---|---|------------|---|-------------|
| $P =$ | | 1 | 0 | 0 | 1 | $L(P) = 9$ |
| $Q =$ | | 1 | 1 | 0 | 1 | $L(Q) = 13$ |
| Wynik $Y =$ | 1 | 0 | 1 | 1 | 0 | $L(Y) = 22$ |
| Przeniesienia: | | 1 | | | 1 | |
| | \uparrow | | | \uparrow | | |
| | c_4 | | | c_1 | | |

Pełny sumator 4-bitowy



| | | | | |
|---|---|---|---|---|
| C | 1 | 1 | 1 | 0 |
| A | 0 | 1 | 0 | 1 |
| B | 0 | 1 | 1 | 1 |
| S | 1 | 1 | 0 | 0 |



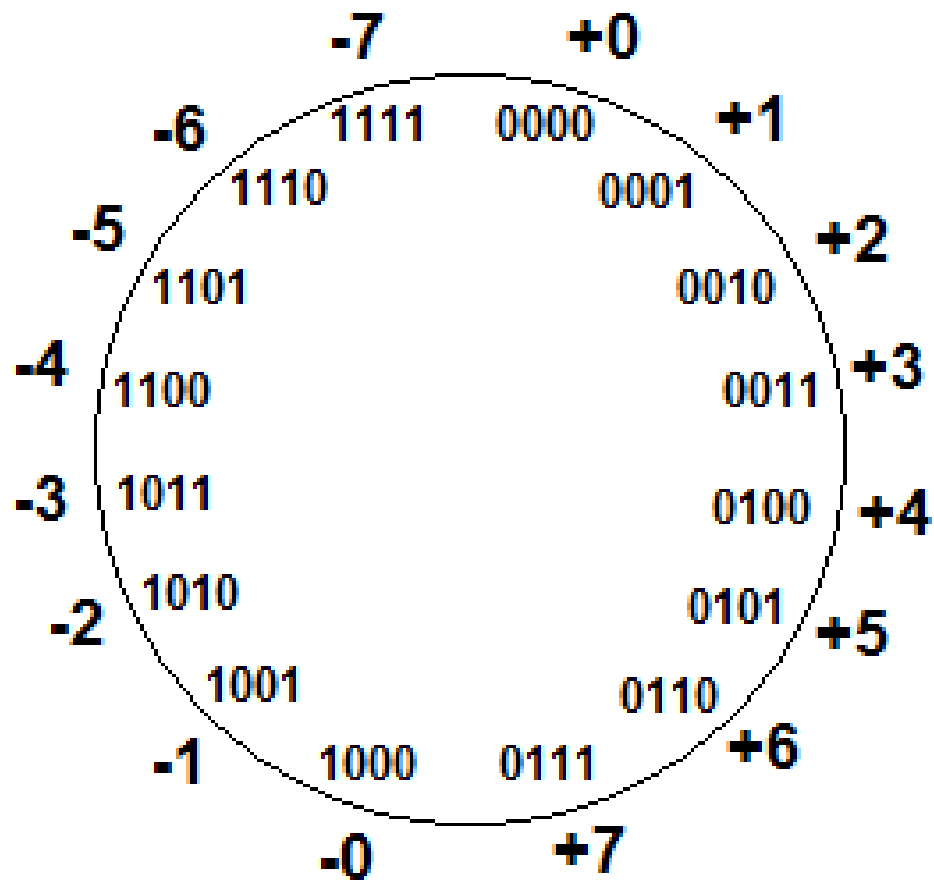
Zapis liczb ze znakiem

Znak liczby koduje się na bicie MSB: 0 (+), 1 (–).

- (1) kod znak-moduł (ZM),
- (2) kod uzupełnienie do 1 (U1),
- (3) kod uzupełnienie do 2 (U2).

| Liczba dziesiętna | ZM | ZU1 | ZU2 |
|-------------------|-------|-------|-------|
| -8 (-1) | | | 1.000 |
| -7 (-0.875) | 1.111 | 1.000 | 1.001 |
| -6 (-0.75) | 1.110 | 1.001 | 1.010 |
| -5 (-0.625) | 1.101 | 1.010 | 1.011 |
| -4 (-0.5) | 1.100 | 1.011 | 1.100 |
| -3 (-0.375) | 1.011 | 1.100 | 1.101 |
| -2 (-0.25) | 1.010 | 1.101 | 1.110 |
| -1 (-0.125) | 1.001 | 1.110 | 1.111 |
| -0 | 1.000 | 1.111 | |
| 0 | | | 0.000 |
| +0 | 0.000 | 0.000 | |
| 1 (0.125) | 0.001 | 0.001 | 0.001 |
| ... | ... | ... | ... |
| 7 (0.875) | 0.111 | 0.111 | 0.111 |

Kod znak-moduł ZM



+

0 100 = + 4

1 100 = - 4

-

Dodawanie/odejmowanie w kodzie ZM

| | | | | |
|-------------------------------------------|------------|-------------|---------------|-------------|
| | 4 | 0100 | -4 | 1100 |
| znak wyniku taki sam jak obu operandów | <u>+ 3</u> | <u>0011</u> | <u>+ (-3)</u> | <u>1011</u> |
| | 7 | 0111 | -7 | 1111 |

| | | | | |
|---------------------------------------------------------------------------|------------|-------------|------------|-------------|
| | 4 | 0100 | -4 | 1100 |
| znak wyniku taki jak znak operandu o większej wartości bezwzględnej | <u>- 3</u> | <u>1011</u> | <u>+ 3</u> | <u>0011</u> |
| | 1 | 0001 | -1 | 1001 |

Różne sposoby realizacji dodawania i odejmowania, połączone ponadto z koniecznością porównywania wartości bezwzględnych liczb.

Zapis liczb ze znakiem

Znak liczby koduje się na bicie MSB: 0 (+), 1 (−).

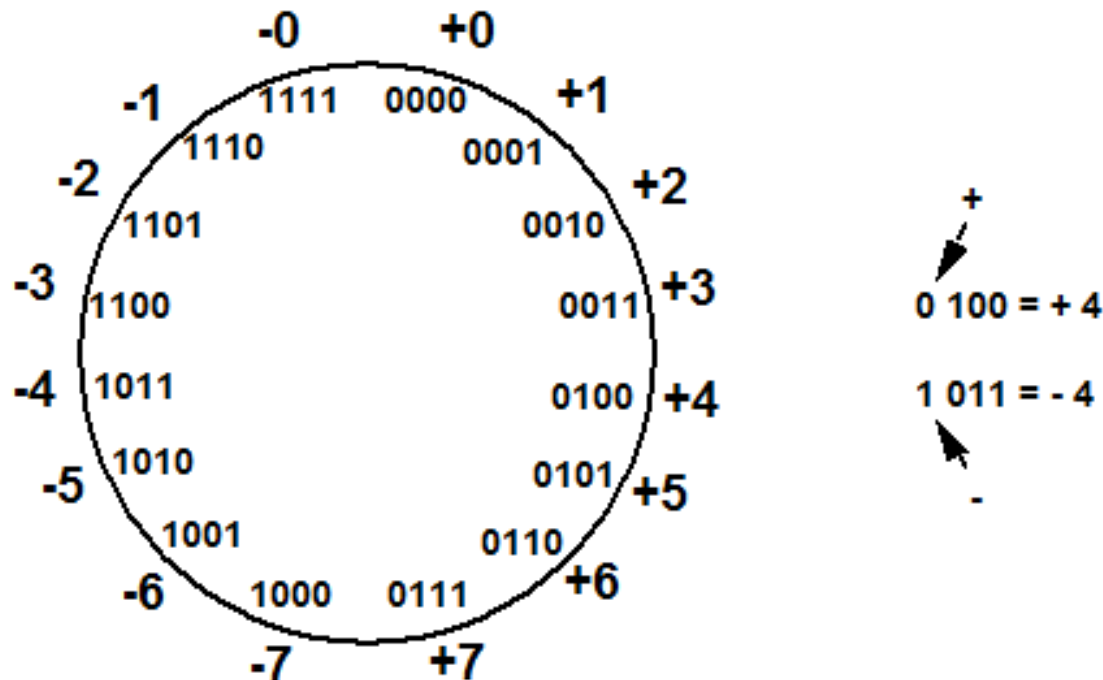
- (1) kod znak-moduł (ZM),
- (2) kod uzupełnienie do 1 (U1),
- (3) kod uzupełnienie do 2 (U2).

| Liczba dziesiętna | ZM | ZU1 | ZU2 |
|-------------------|-------|-------|-------|
| -8 (-1) | | | 1.000 |
| -7 (-0.875) | 1.111 | 1.000 | 1.001 |
| -6 (-0.75) | 1.110 | 1.001 | 1.010 |
| -5 (-0.625) | 1.101 | 1.010 | 1.011 |
| -4 (-0.5) | 1.100 | 1.011 | 1.100 |
| -3 (-0.375) | 1.011 | 1.100 | 1.101 |
| -2 (-0.25) | 1.010 | 1.101 | 1.110 |
| -1 (-0.125) | 1.001 | 1.110 | 1.111 |
| -0 | 1.000 | 1.111 | |
| 0 | | | 0.000 |
| +0 | 0.000 | 0.000 | |
| 1 (0.125) | 0.001 | 0.001 | 0.001 |
| ... | ... | ... | ... |
| 7 (0.875) | 0.111 | 0.111 | 0.111 |

Uzupełnienia liczb binarnych – U1

Uzupełnienie do 1 powstaje przez zanegowanie wszystkich bitów liczby wyjściowej:

Ex.: $P = 1011100_{\text{ZM}} = 0100011_{\text{U1}}$



Dodawanie/odejmowanie w kodzie U1

| | | | |
|------------|-------------|-----------------------|-------------|
| 4 | 0100 | -4 | 1011 |
| <u>+ 3</u> | <u>0011</u> | <u>+ (-3)</u> | <u>1100</u> |
| 7 | 0111 | -7 | 10111 |
| | | Przeniesienie zwrotne | → 1 |
| | | | 1000 |
| 4 | 0100 | -4 | 1011 |
| <u>- 3</u> | <u>1100</u> | <u>+ 3</u> | <u>0011</u> |
| 1 | 10000 | -1 | 1110 |
| | | Przeniesienie zwrotne | → 1 |
| | | | 0001 |

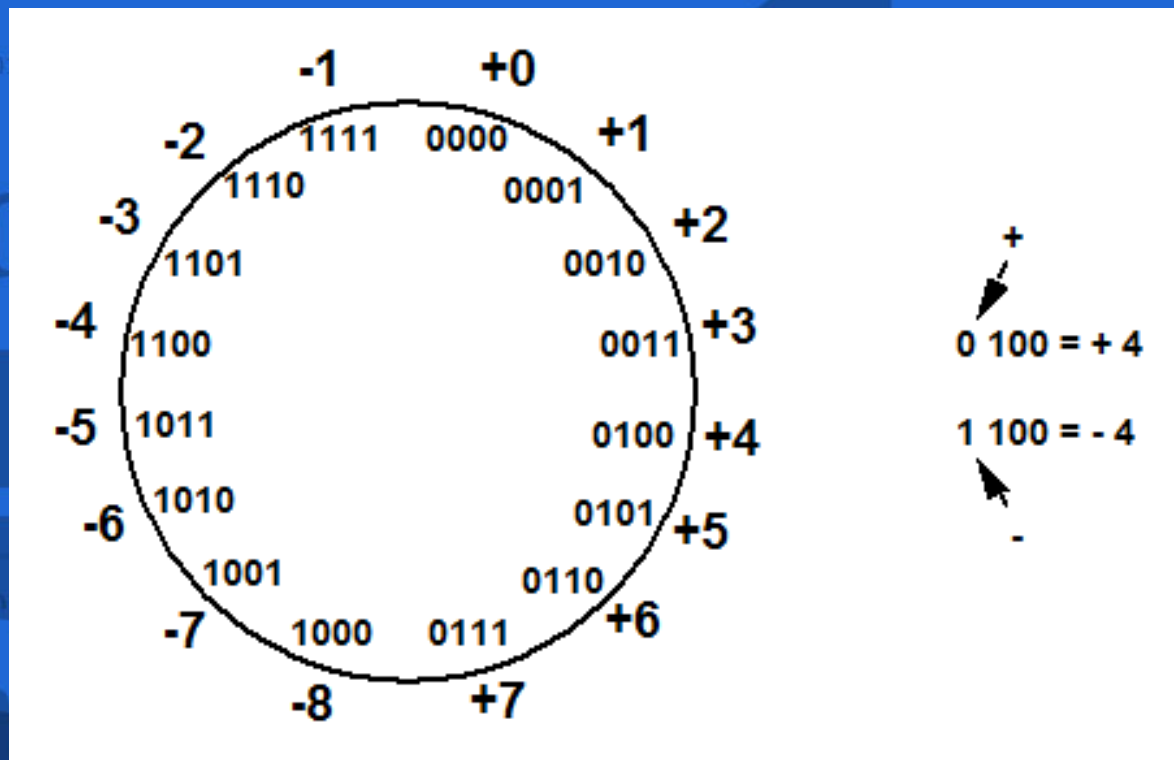
Konieczność obsługi bitu przeniesienia zwrotnego, ale za to dodawanie i odejmowanie realizowane identycznie.

Uzupełnienia liczb binarnych – U2

Uzupełnienie do 2 powstaje przez dodanie 1 do liczby $U1(P)$.

Reguła mnemotechniczna:

Przepisz wszystkie mniej znaczące zera i pierwszą najmniej znaczącą jedynkę słowa P , a następnie zaneguj wszystkie pozostałe bity: $P = 1011100_{ZM} = 0100100_{U2}$



Dodawanie/odejmowanie w kodzie U2

$$\begin{array}{r} 5 \quad \quad \quad \textcircled{0}111 \\ \quad \quad \quad \underline{0101} \\ \hline 3 \quad \quad \quad \underline{0011} \\ \hline -8 \quad \quad \quad 01000 \end{array}$$

Overflow

$$\begin{array}{r} 5 \quad \quad \quad \textcircled{0}000 \\ \quad \quad \quad \underline{0101} \\ \hline 2 \quad \quad \quad \underline{0010} \\ \hline 7 \quad \quad \quad 00111 \end{array}$$

No overflow

$$\begin{array}{r} -7 \quad \quad \quad \textcircled{1}000 \\ \quad \quad \quad \underline{1001} \\ \hline -2 \quad \quad \quad \underline{1100} \\ \hline 7 \quad \quad \quad 10111 \end{array}$$

Overflow

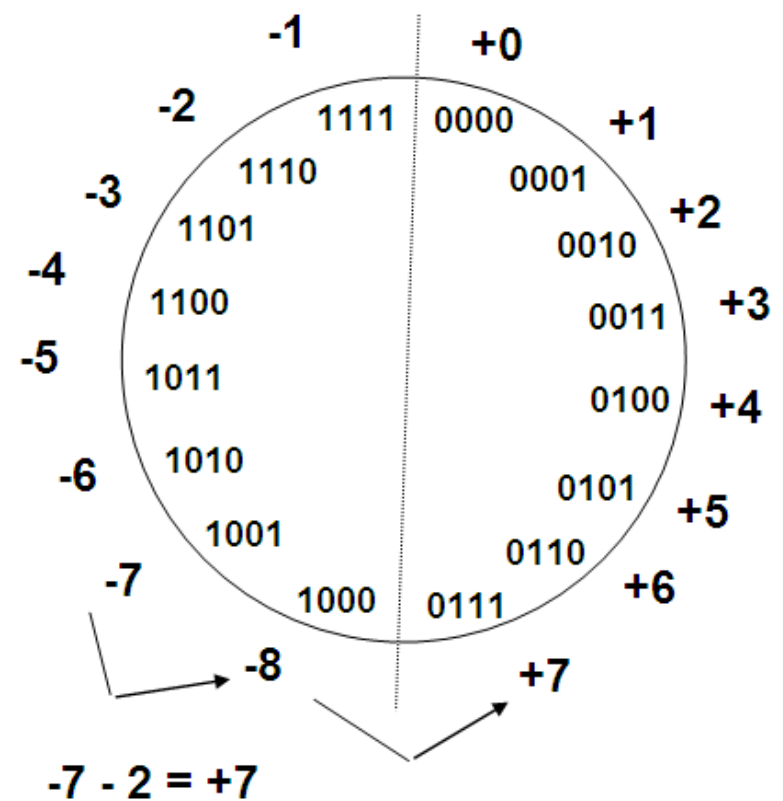
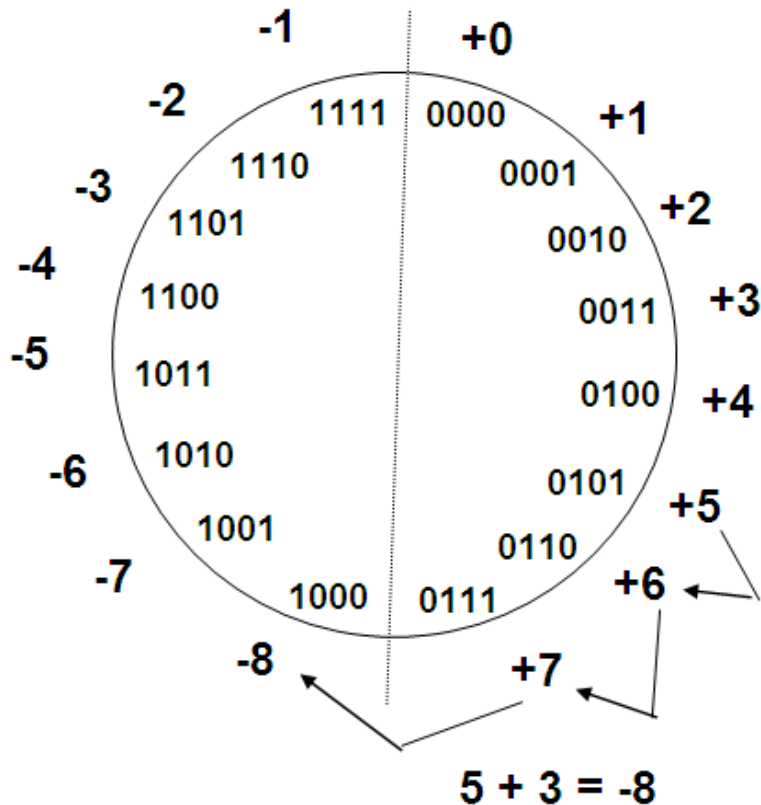
$$\begin{array}{r} -3 \quad \quad \quad \textcircled{1}111 \\ \quad \quad \quad \underline{1101} \\ \hline -5 \quad \quad \quad \underline{1011} \\ \hline -8 \quad \quad \quad 11000 \end{array}$$

No overflow

Jeśli przeniesienie na bit znaku i bit nadmiaru są takie same, ignoruj je. Jeśli są różne, występuje błąd przepełnienia (overflow).

Prostota algorytmu dodawania/odejmowania czyni kod U2 najczęściej wykorzystywanym do operacji arytmetycznych na liczbach binarnych.

Błąd przepełnienia (overflow)



Błąd przepełnienia powstaje, gdy znak wyniku różni się od znaków operandów (bit przeniesienia na pozycję znaku różni się od bitu nadmiaru).

Błąd przepełnienia (overflow)

Jeżeli **obydwie** dodawane liczby są **dodatnie albo ujemne**, to wystąpienie przeniesienia 1 z sumy modułów na pozycję znaku zmienia znak wyniku. Wówczas **wynik jest błędny!**

Przykład z użyciem kodu ZU2

Bit znaku
↓

| | |
|-------|-----------|
| +69 | 0.1000101 |
| +103 | 0.1100111 |
| <hr/> | |
| +172 | 1.0101100 |

– 84 w ZU2?

$c_m = 1$
 $c_z = 0$

c_m – przeniesienie z modułu sumy

c_z – przeniesienie z bitu znaku

TEST NADMIARU ►

$$c_z \oplus c_m = 1$$

czyli błąd występuje gdy te bity przeniesień mają **różne** wartości.

Błąd przepełnienia (overflow)

Aby wyeliminować możliwość wystąpienia błędu, należy **zwiększyć długość słowa danych przynajmniej o 1 bit**

$$\begin{array}{rcl} +69 & 0.01000101 & \\ +103 & 0.01100111 & \\ \hline +172 & 0.10101100 & +172 \blacktriangleright \text{wynik poprawny} \end{array}$$

↑
Bit znaku

Inny przykład:

$$\begin{array}{rcl} -39 & 1.11011001 & \\ -115 & 1.10001101 & \\ \hline -154 & 1.01100110 & -154 \blacktriangleright \text{wynik poprawny} \end{array}$$

↑
Bit znaku

dodatkowy bit

uzupełnienie formatu
39 = 0100111

► W porównaniu z kodem **U2** potrzebne są dwa dodatkowe bity!

Dodawanie/odejmowanie liczb w kodzie U2

(A) Dodawanie liczb o tych samych znakach w kodzie U2 wymaga wcześniejszego zwiększenia długości słowa modułu każdej liczby o 1 bit.

(B) Długość słowa argumentów operacji w kodzie U2 jest większa o 2 bity: bit znaku oraz bit nadmiaru.

Np. $-4 + (-15) = -19$

$$\text{NB}(4) = 100, \quad \text{NB}(15) = 1111$$

► wyrównujemy formaty: $\text{NB}(4) = 0100$

► obliczamy uzupełnienia: $\text{U2}(4) = 1100, \quad \text{U2}(15) = 0001$

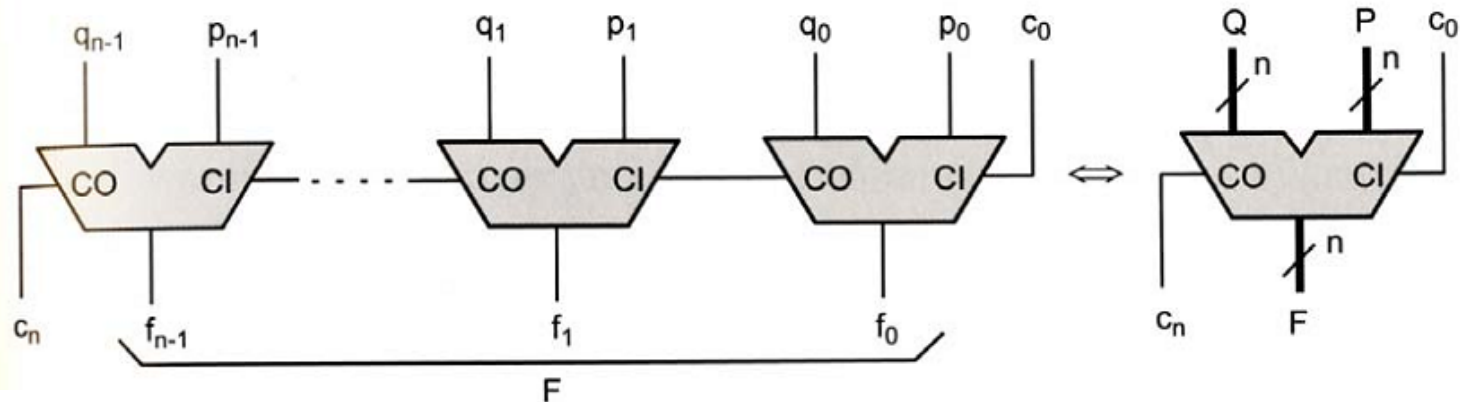
Spróbujemy dodać $-4 + (-15)$ w kodzie ZU2

$$\begin{array}{r} 1.1100 \\ 1.0001 \\ \hline \neq 0.1101 \rightarrow +13??? \end{array}$$

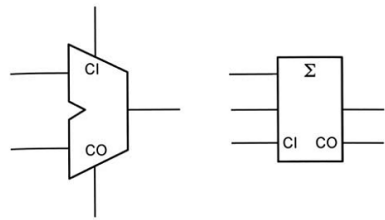
Przeniesienie z pozycji znaku $c_z = 1$ i z pozycji MSB modułu $c_m = 0$

Test nadmiaru: $c_z \oplus c_m = 1 \rightarrow \text{ERROR!!!}$

Sumator z przeniesieniami szeregowymi (sumator kaskadowy, ripple carry adder RCA)



Symbole graficzne sumatora

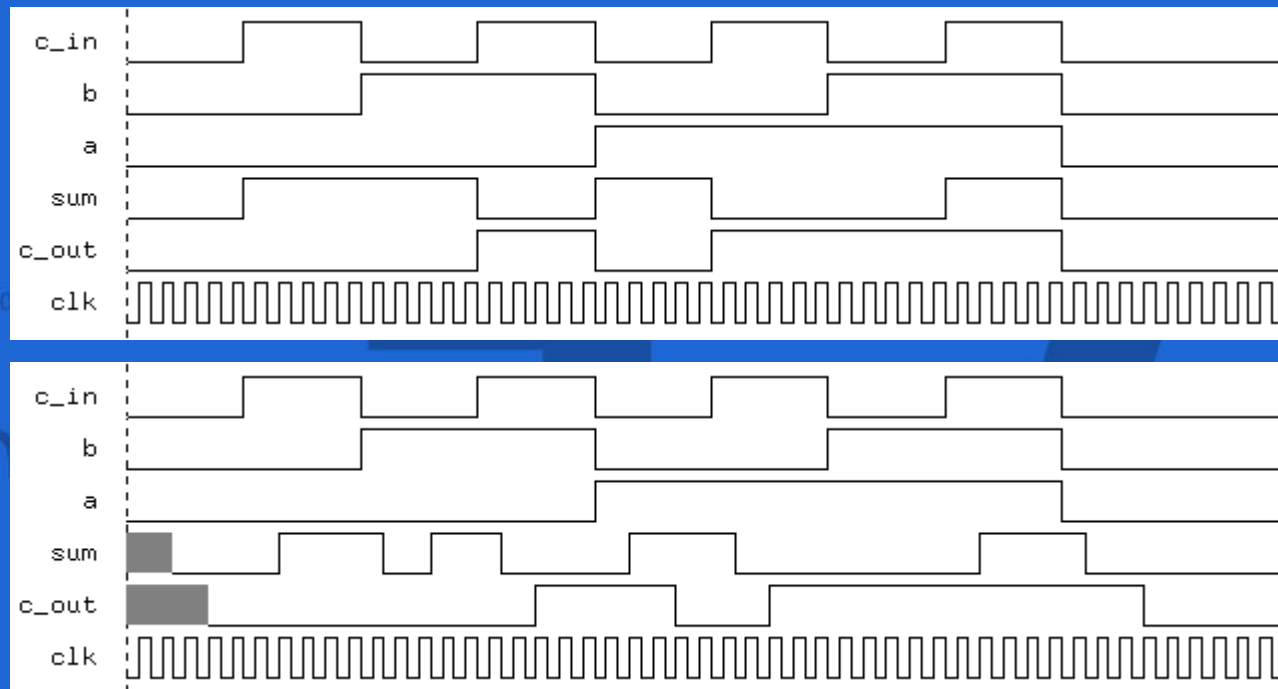
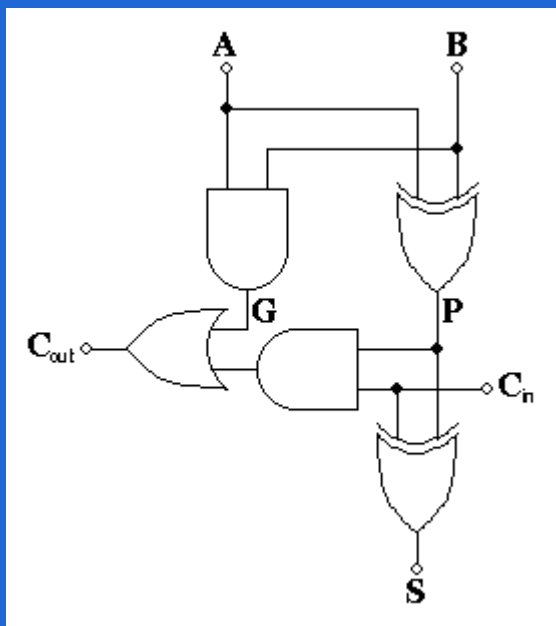


$$c_n \circ F = P + Q + c_0$$

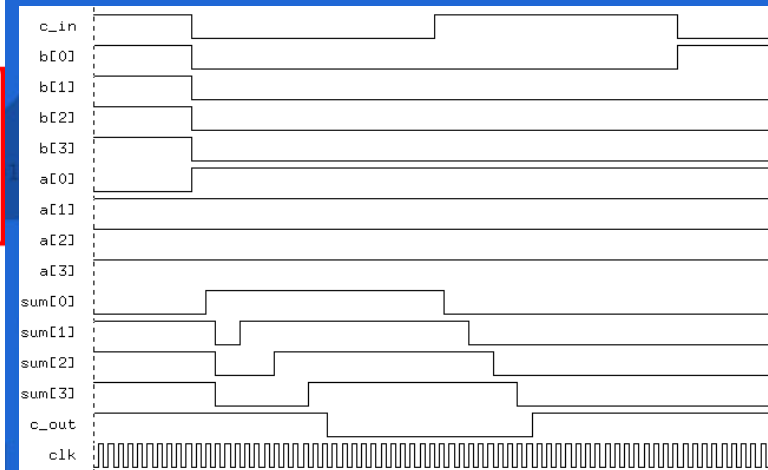
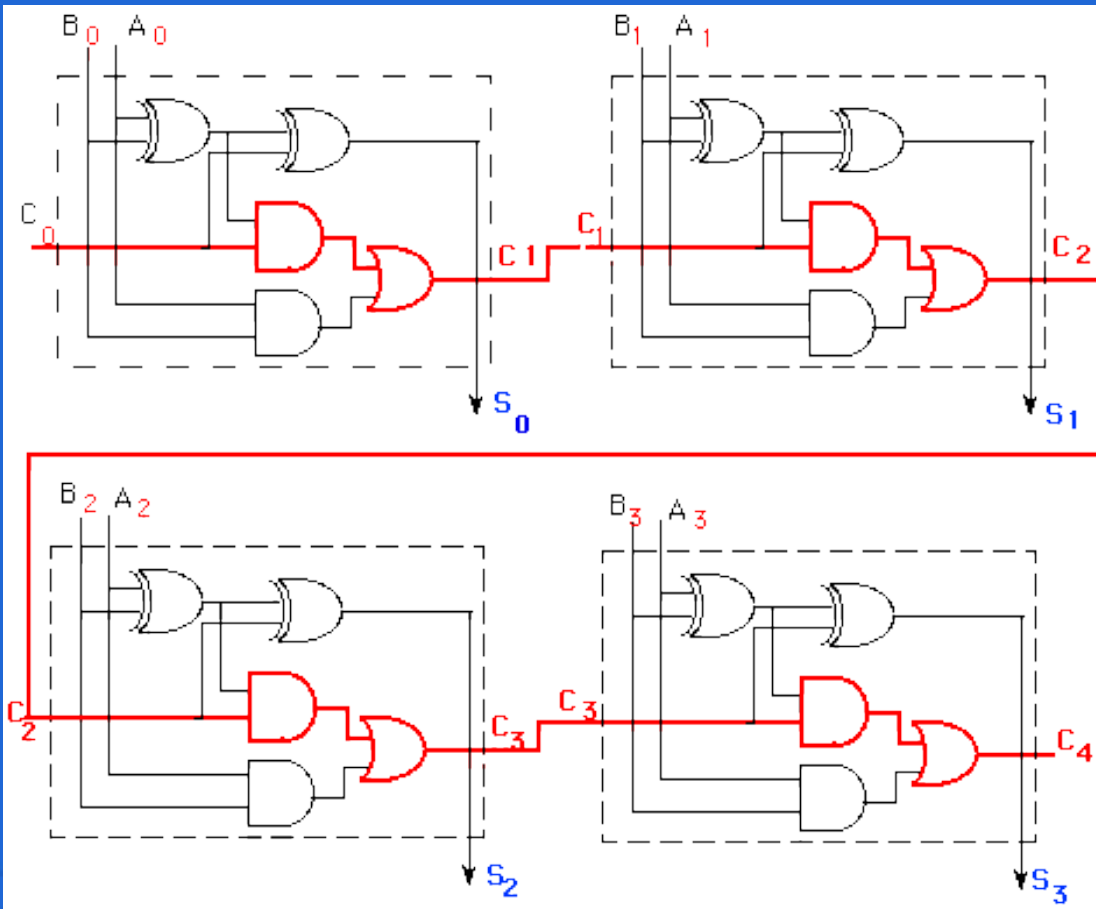
Wada: długi czas dodawania (transmisji przeniesień do uzyskania poprawnych bitów f_{n-1} i c_n)

Zaleta: prostota budowy

Opóźnienia w sumatorze 1-bitowym

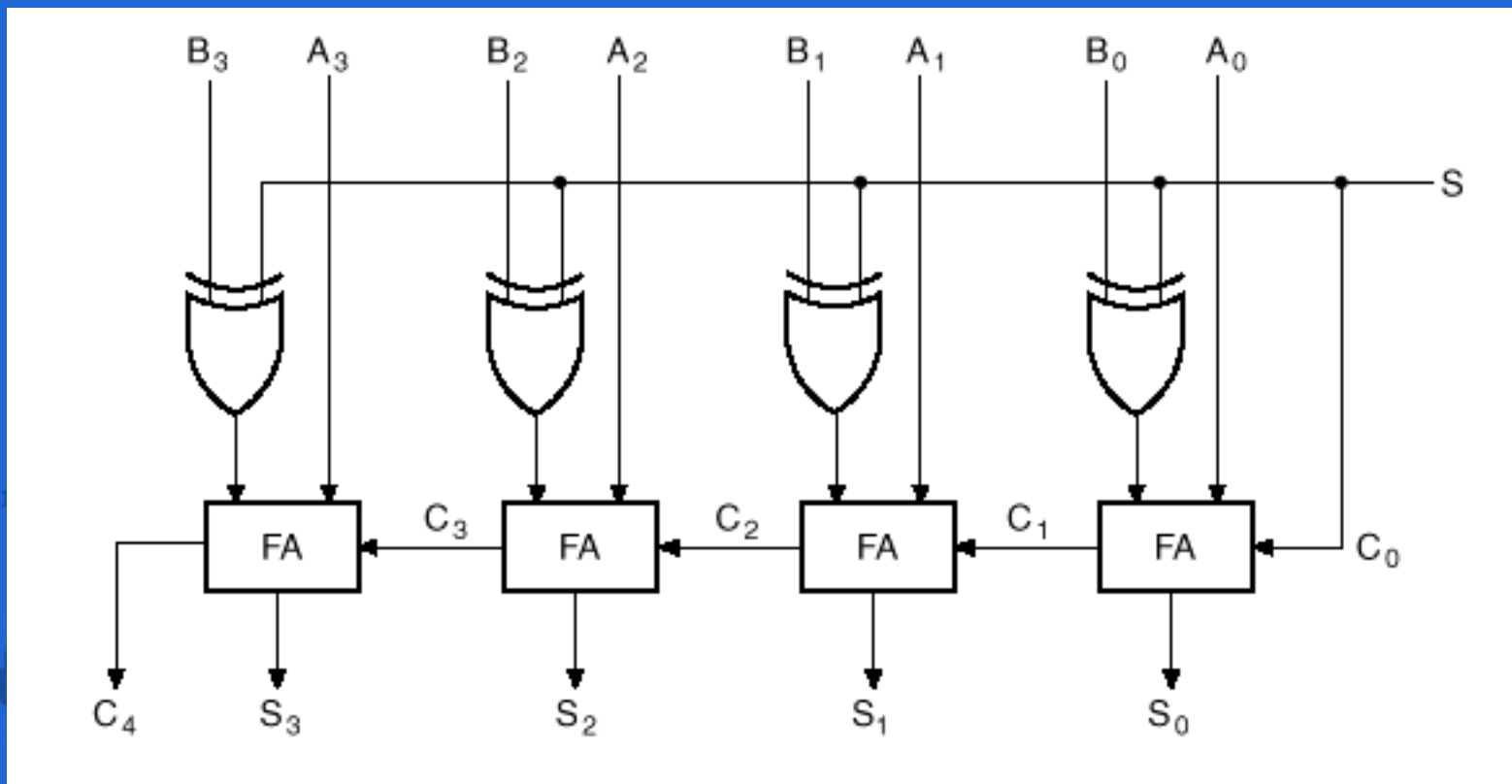


Opóźnienia w sumatorze kaskadowym 4-bitowym



110100100

Sumator-subtraktor kaskadowy 4-bitowy



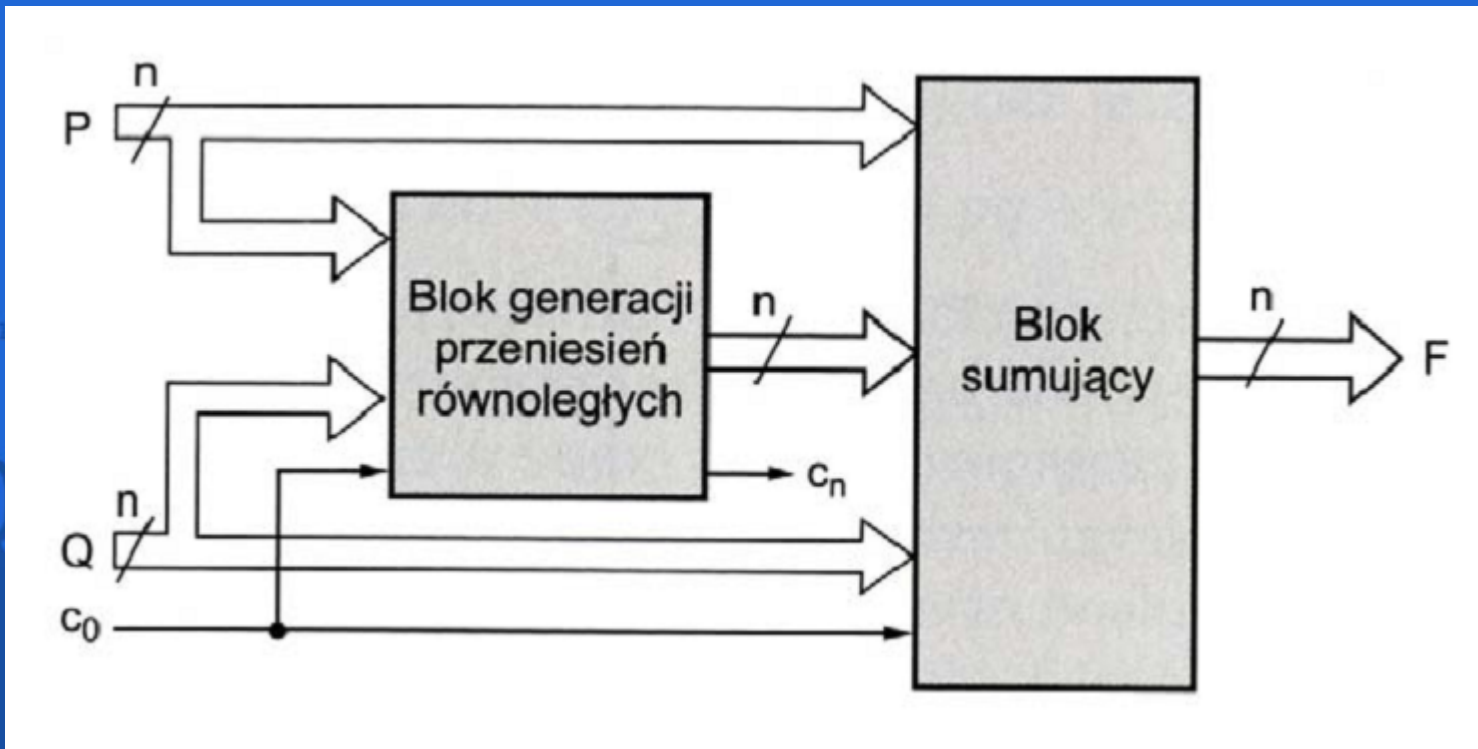
Dodawanie: $S = 0, C_0 = 0$:

$$S_3 S_2 S_1 S_0 = A_3 A_2 A_1 A_0 + B_3 B_2 B_1 B_0 + C_0 = A + B$$

Odejmowanie: $S = 1, C_0 = 1$:

$$S_3 S_2 S_1 S_0 = A_3 A_2 A_1 A_0 + \overline{B_3} \overline{B_2} \overline{B_1} \overline{B_0} + 1 = A + U2(B) = A - B$$

Sumator z przeniesieniami równoległymi (carry-lookahead adder CLA)

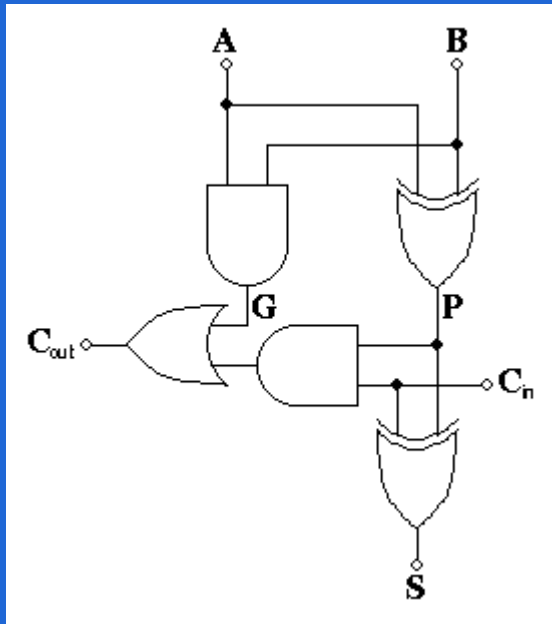


Generacja przeniesień równoległych

Zauważmy, że w przypadku FA:

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = A \cdot B + (A + B)C_{in} = A \cdot B + (A \oplus B)C_{in}$$



Wprowadźmy oznaczenia:

(A) Generacja przeniesienia:

$$G = A \cdot B$$

(B) Propagacja przeniesienia:

$$P = A \oplus B$$

Wówczas:

$$S = P \oplus C_{in}$$

$$C_{out} = G + P \cdot C_{in}$$

Generacja przeniesień równoległych

Aby wygenerować jednocześnie wszystkie bity przeniesień należy zbudować funkcję niezależną od przeniesień wstępnych c_i ($i \geq 1$). Korzystając z bitów generacji G_i i propagacji P_i przeniesień sumatora:

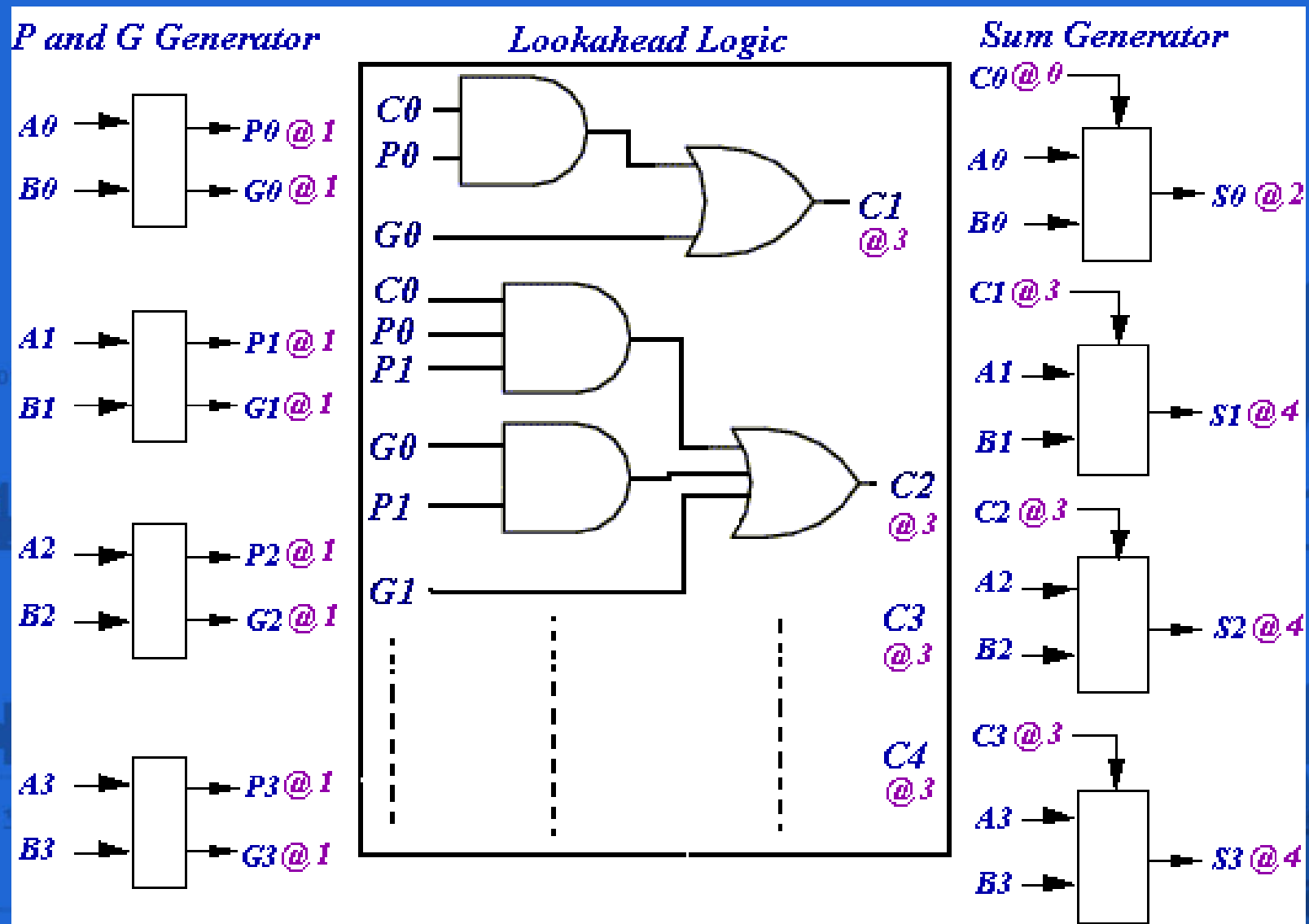
$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

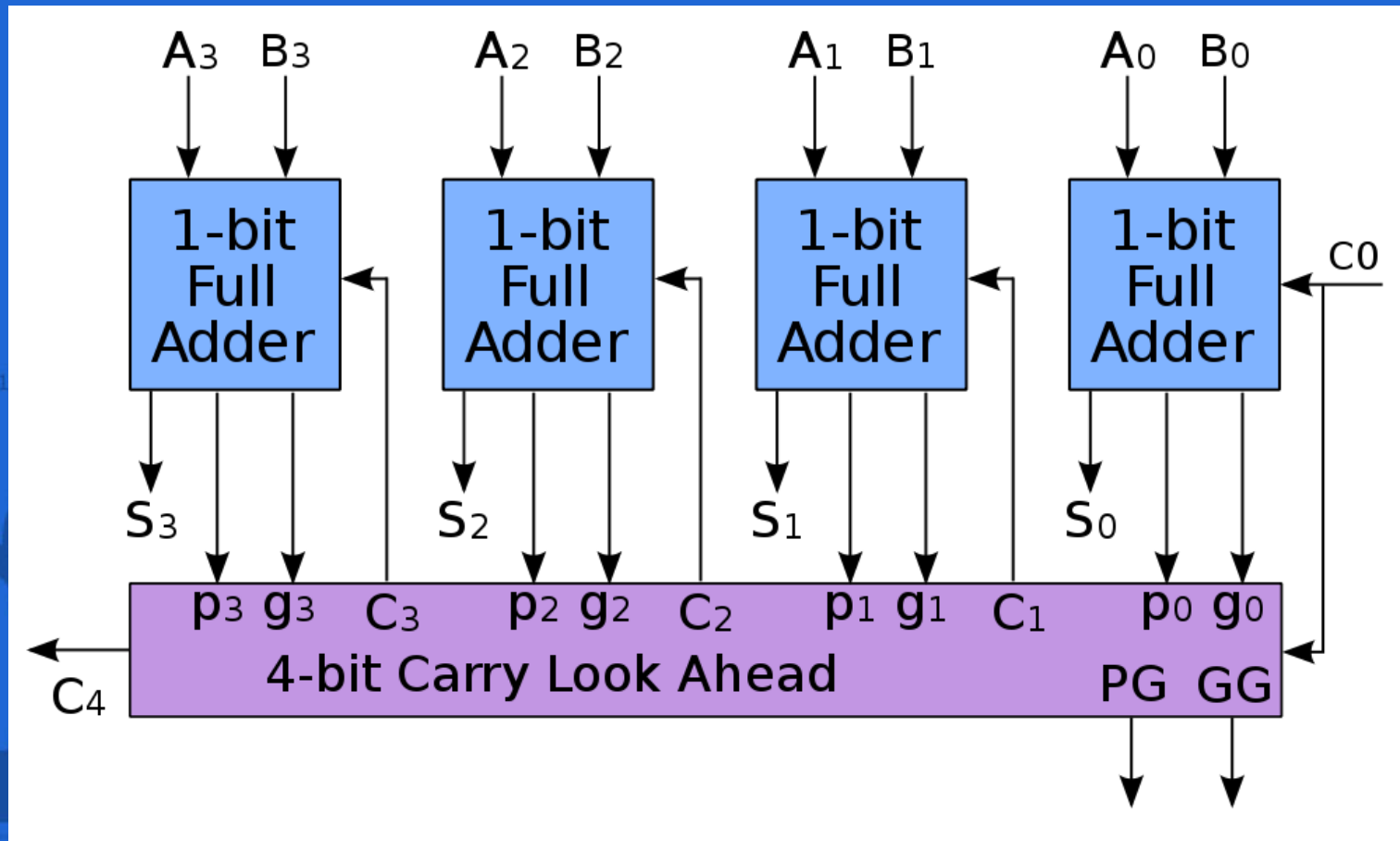
$$\dots$$
$$C_{i+1} = G_i + P_i \cdot C_i = G_i + f(A, B, C_0)$$

Przeniesienie na pozycję $(i+1)$ realizowane jest więc rekurencyjnie. Sumatory CLA są o ok. 40 % szybsze niż RCA, aczkolwiek nie buduje się układów większych niż 4-bitowe (komplikacja układu).

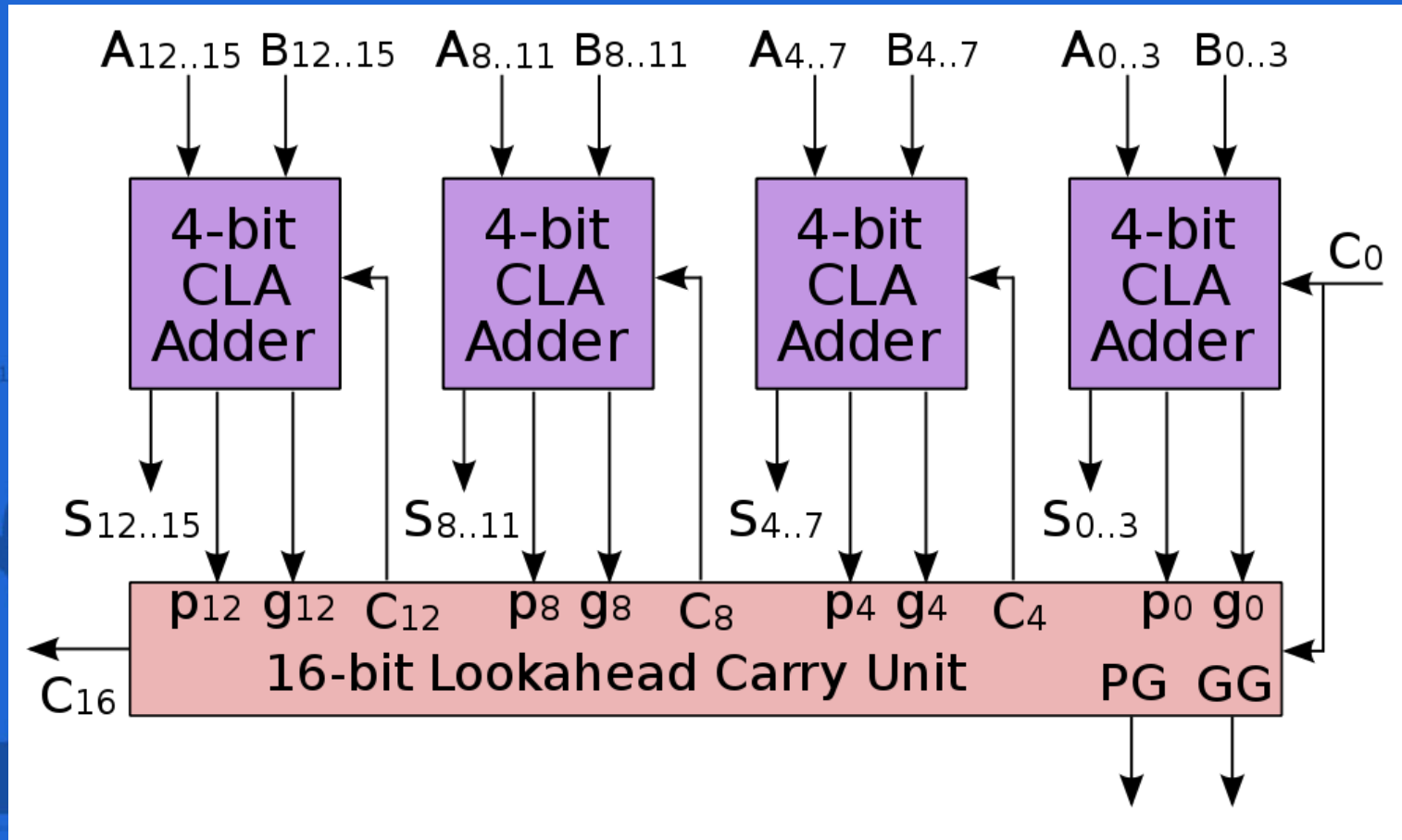
Generacja przeniesień równoległych



4-bitowy sumator CLA



16-bitowy sumator CLA



Generator przeniesień równoległych (Look Ahead Carry Generator) układ 74182 (Texas Instruments)

FUNCTION TABLE FOR \bar{G} OUTPUT

| INPUTS | | | | | | | OUTPUT |
|------------------------|-------------|-------------|-------------|-------------|-------------|-------------|--------|
| \bar{G}_3 | \bar{G}_2 | \bar{G}_1 | \bar{G}_0 | \bar{P}_3 | \bar{P}_2 | \bar{P}_1 | |
| L | X | X | X | X | X | X | L |
| X | L | X | X | L | X | X | L |
| X | X | L | X | L | L | X | L |
| X | X | X | L | L | L | L | L |
| All other combinations | | | | | | | H |

FUNCTION TABLE
FOR \bar{P} OUTPUT

| INPUTS | | | | OUTPUT |
|------------------------|-------------|-------------|-------------|--------|
| \bar{P}_3 | \bar{P}_2 | \bar{P}_1 | \bar{P}_0 | |
| L | L | L | L | L |
| All other combinations | | | | H |

FUNCTION TABLE
FOR C_{n+x} OUTPUT

| INPUTS | | | OUTPUT |
|------------------------|-------------|-------|--------|
| \bar{G}_0 | \bar{P}_0 | C_n | |
| L | X | X | H |
| X | L | H | H |
| All other combinations | | | L |

FUNCTION TABLE
FOR C_{n+y} OUTPUT

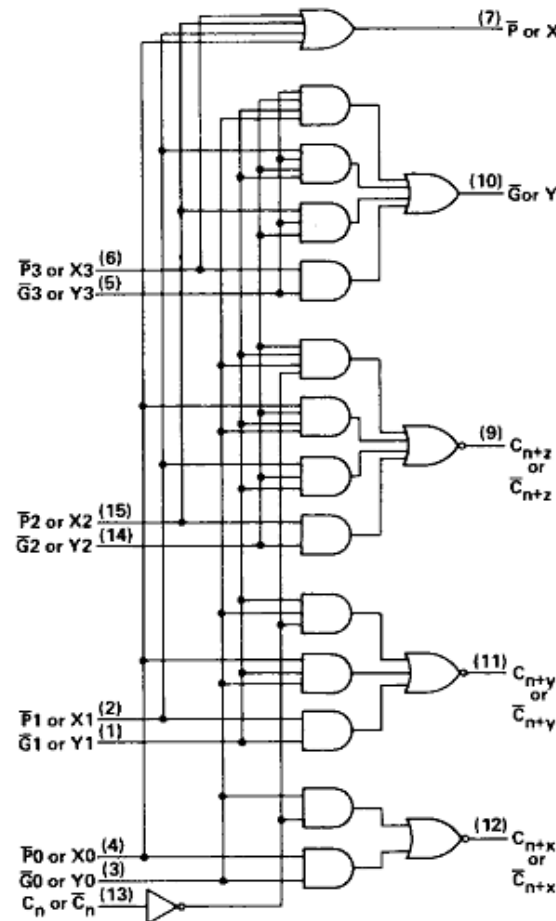
| INPUTS | | | | | | OUTPUT |
|------------------------|-------------|-------------|-------------|-------|--|--------|
| \bar{G}_1 | \bar{G}_0 | \bar{P}_1 | \bar{P}_0 | C_n | | |
| L | X | X | X | X | | H |
| X | L | L | X | X | | H |
| X | X | L | L | H | | H |
| All other combinations | | | | | | L |

FUNCTION TABLE FOR C_{n+z} OUTPUT

| INPUTS | | | | | | | OUTPUT |
|------------------------|-------------|-------------|-------------|-------------|-------------|-------|--------|
| \bar{G}_2 | \bar{G}_1 | \bar{G}_0 | \bar{P}_2 | \bar{P}_1 | \bar{P}_0 | C_n | |
| L | X | X | X | X | X | X | H |
| X | L | X | L | X | X | X | H |
| X | X | L | L | L | X | X | H |
| X | X | X | L | L | L | H | H |
| All other combinations | | | | | | | L |

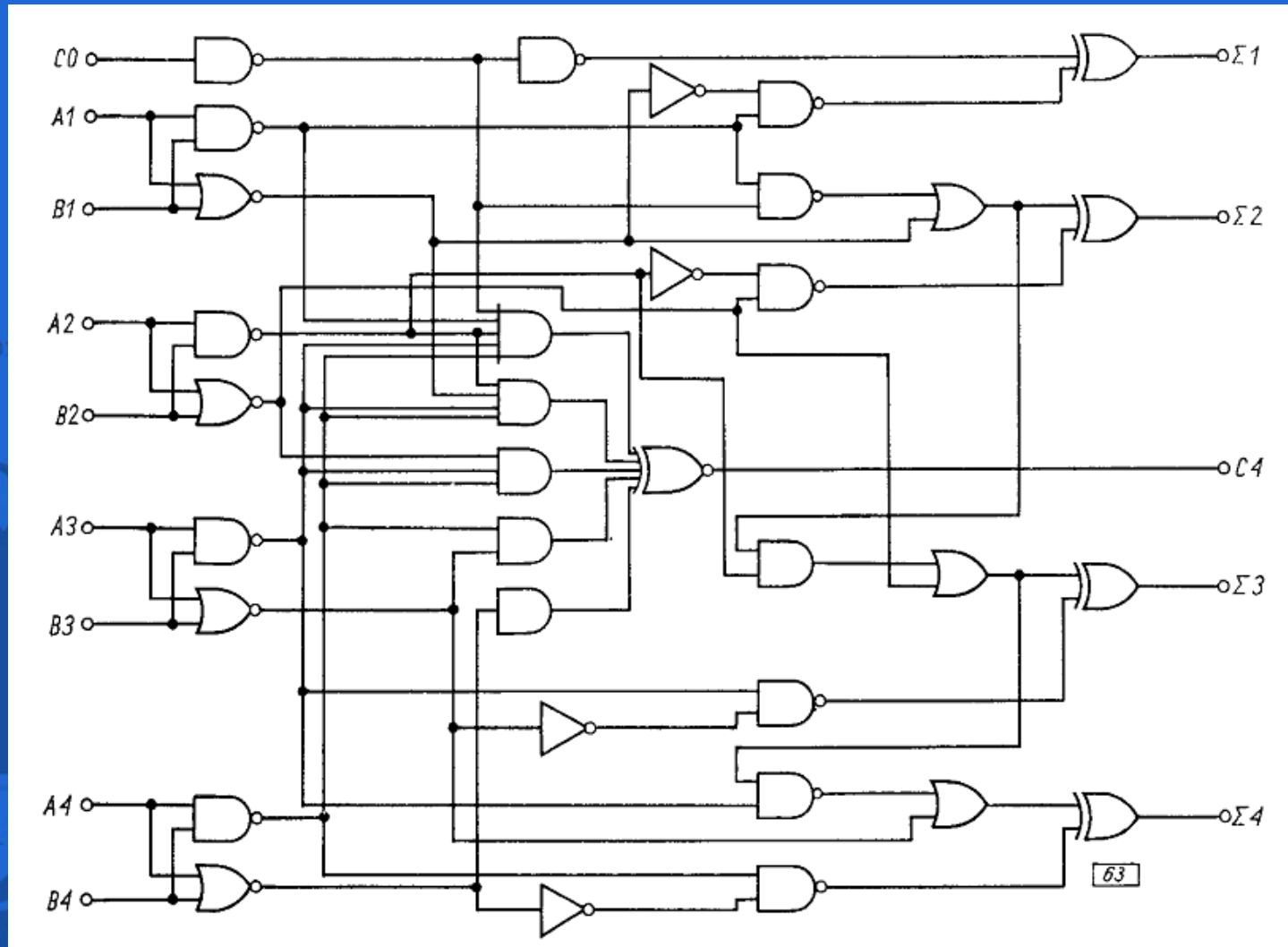
H = high level, L = low level, X = irrelevant

Any inputs not shown in a given table are irrelevant with respect to that output.



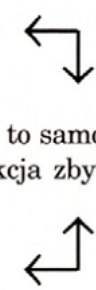
Pin numbers shown on logic notation are for D, J or N packages.

4-bitowy sumator binarny z przeniesieniem równoległym (CLA) – układ 7483 (CEMI)



Dodawanie liczb w kodzie BCD

Aby operować na liczbach binarnych w kodzie BCD należy korygować wynik generując bity przeniesienia w przypadku, gdy liczba jest większa niż $9_{10} = 1001_{NB}$ (przeniesienie dekadowe d). Uzyskuje się je poprzez dodanie do korygowanego wyniku liczby $6_{10} = 0110_{NB}$.

| Liczba dziesiętna | Suma nie skorygowana $cs_3s_2s_1s_0$ | Suma skorygowana $df_3f_2f_1f_0$ |
|-------------------|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| 0 | 0 0000 |  <p>to samo (korekcja zbyteczna)</p> |
| 1 | 0 0001 | |
| 2 | 0 0010 | |
| 3 | 0 0011 | |
| 4 | 0 0100 | |
| 5 | 0 0101 | |
| 6 | 0 0110 | |
| 7 | 0 0111 | |
| 8 | 0 1000 | |
| 9 | 0 1001 | |
| 10 | 0 $\overline{1010}$ | 10000 |
| 11 | 0 $\overline{1011}$ | 10001 |
| 12 | 0 $\overline{1100}$ | 10010 |
| 13 | 0 $\overline{1101}$ | 10011 |
| 14 | 0 $\overline{1110}$ | 10100 |
| 15 | 0 $\overline{1111}$ | 10101 |
| 16 | 4 implikanty funkcji d $\overline{1} 0000$ | 10110 |
| 17 | $\overline{1} 0001$ | 10111 |
| 18 | $\overline{1} 0010$ | 11000 |
| 19 | $\overline{1} 0011$ | 11001 |

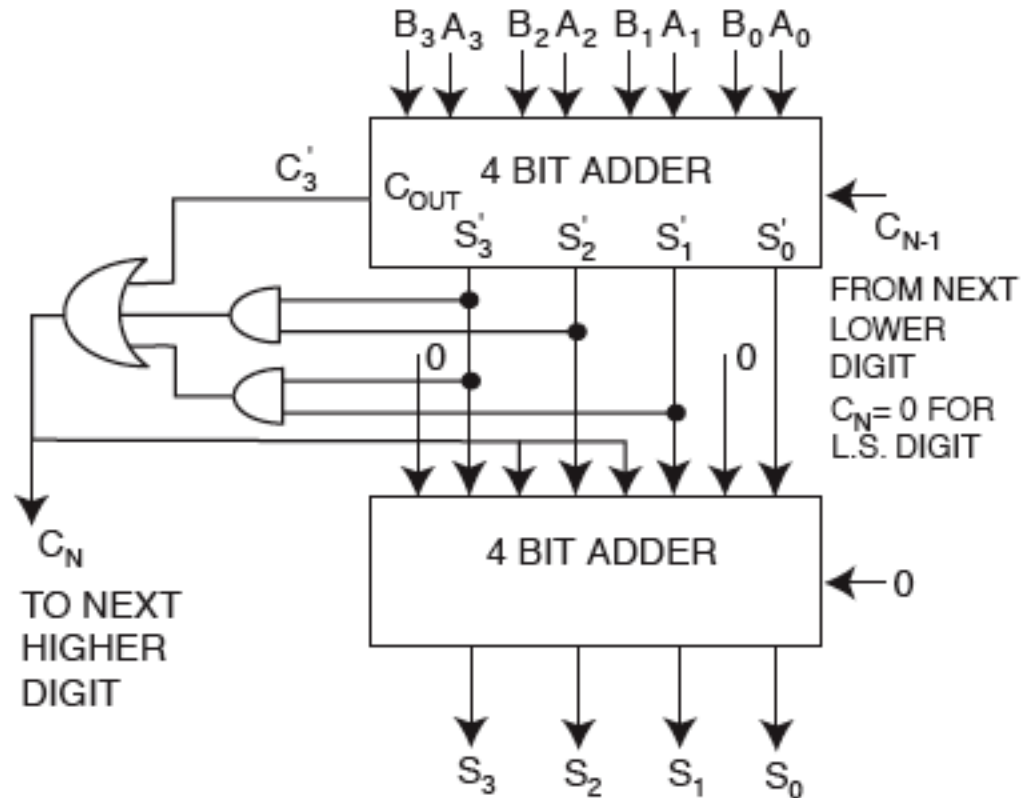
Sumator BCD

Przeniesienie dekadowe:

$$d = c + s_1s_3 + s_2s_3$$

| s_1s_0 \ s_3s_2 | | s_1s_0 | | | |
|---------------------|--|----------|----|----|----|
| | | 00 | 01 | 11 | 10 |
| 00 | | 0 | 0 | 0 | 0 |
| 01 | | 0 | 0 | 0 | 0 |
| 11 | | 1 | 1 | 1 | 1 |
| 10 | | 0 | 0 | 1 | 1 |

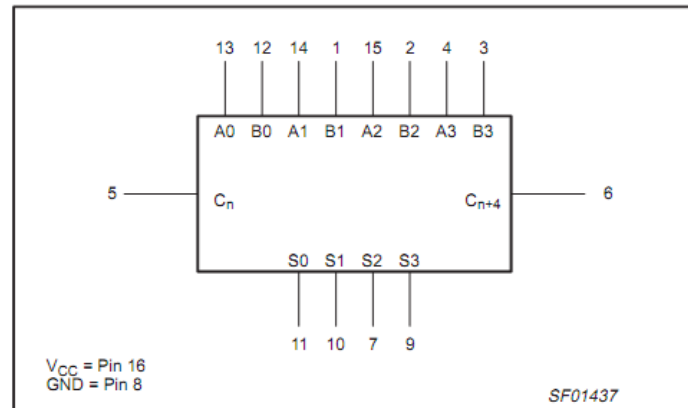
$s_1s_3 + s_2s_3$



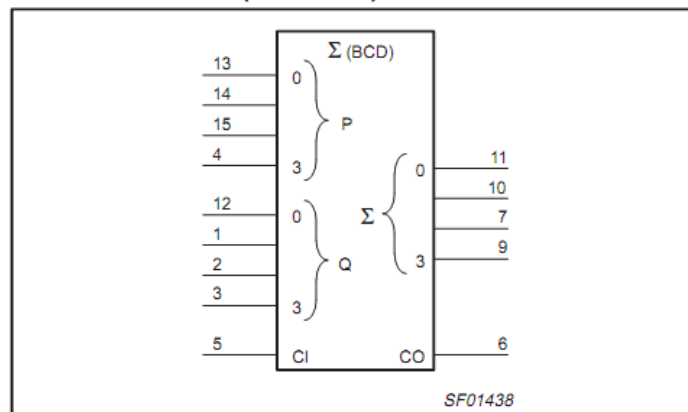
- ADD 0110 WHEN $C_N=1$
- ADD 0000 WHEN $C_N=0$

Sumator BCD – układ 74F583 (Philips)

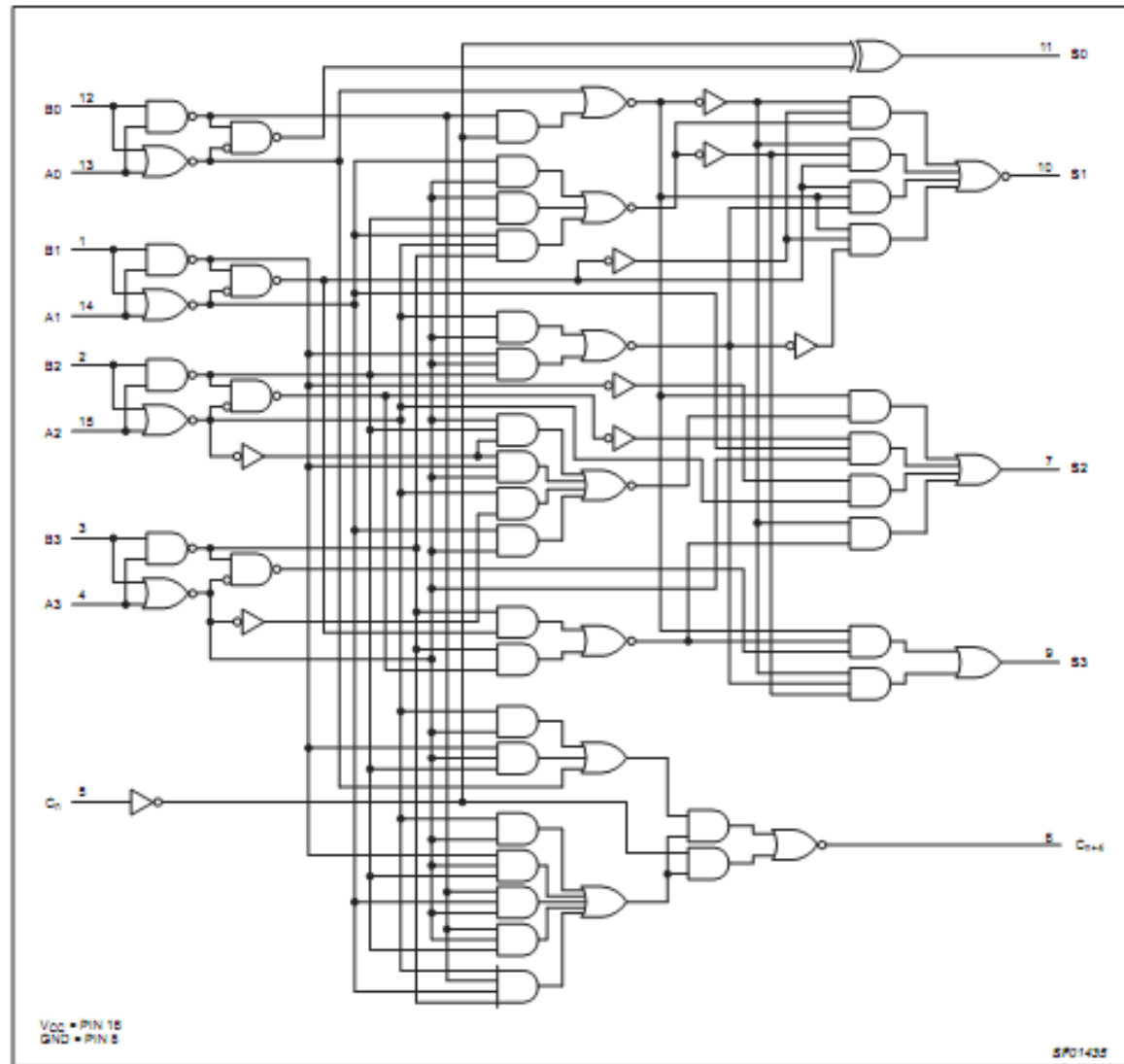
LOGIC SYMBOL



LOGIC SYMBOL (IEEE/IEC)

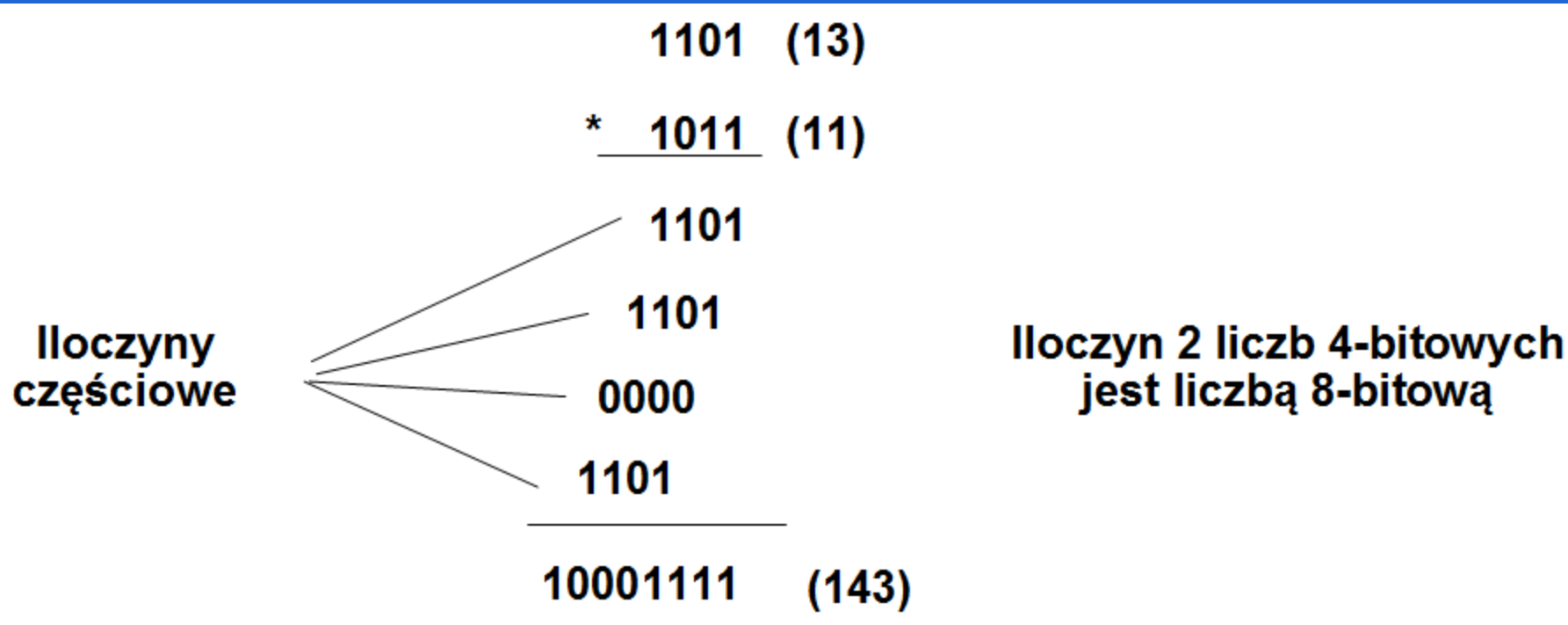


LOGIC DIAGRAM

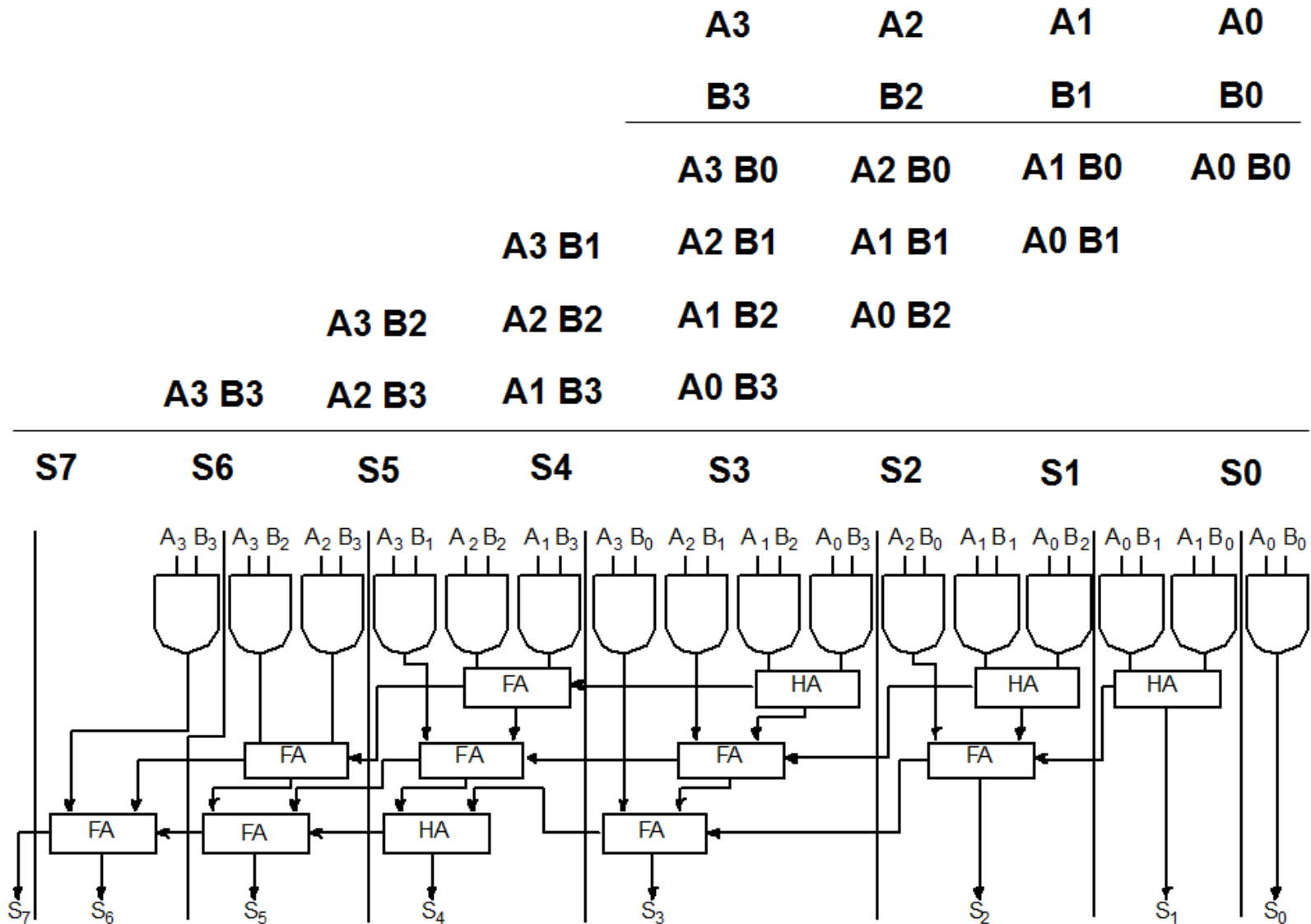


SF01435

Mnożenie liczb binarnych 4-bitowych

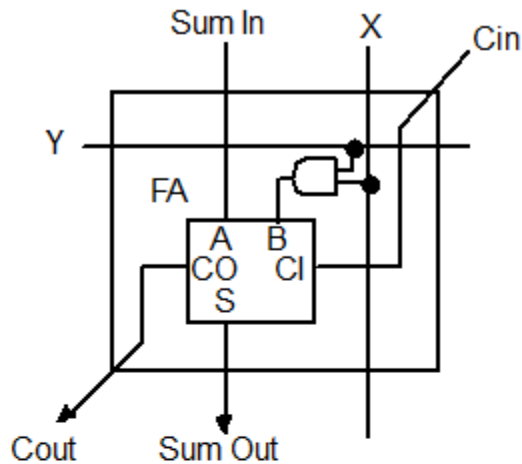


Akumulacja iloczynów częściowych

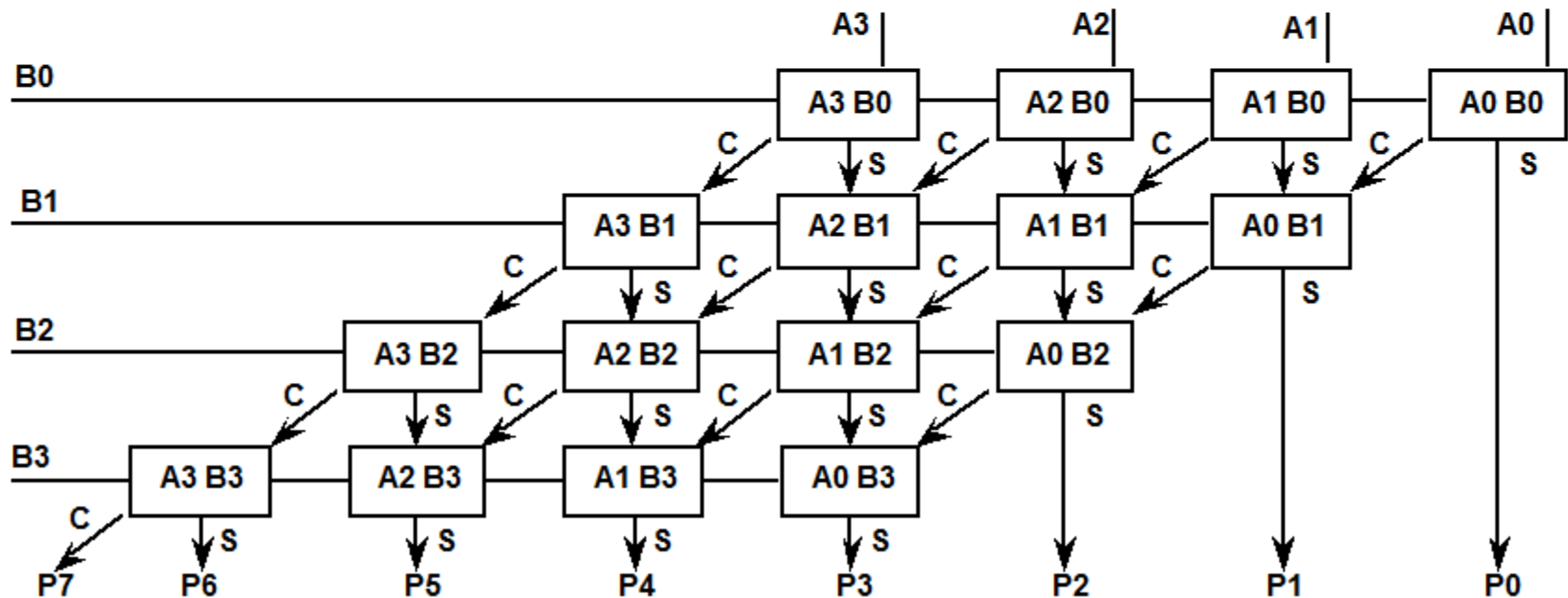


12 sumatorów x 6 bramek + 16 bramek = 88 bramek

Multiplikator matrycowy 4x4



1 komórka = pełny sumator + AND



Kombinacyjny multiplikator 4x4 (układy SN74284 + SN74285 Texas Instruments)

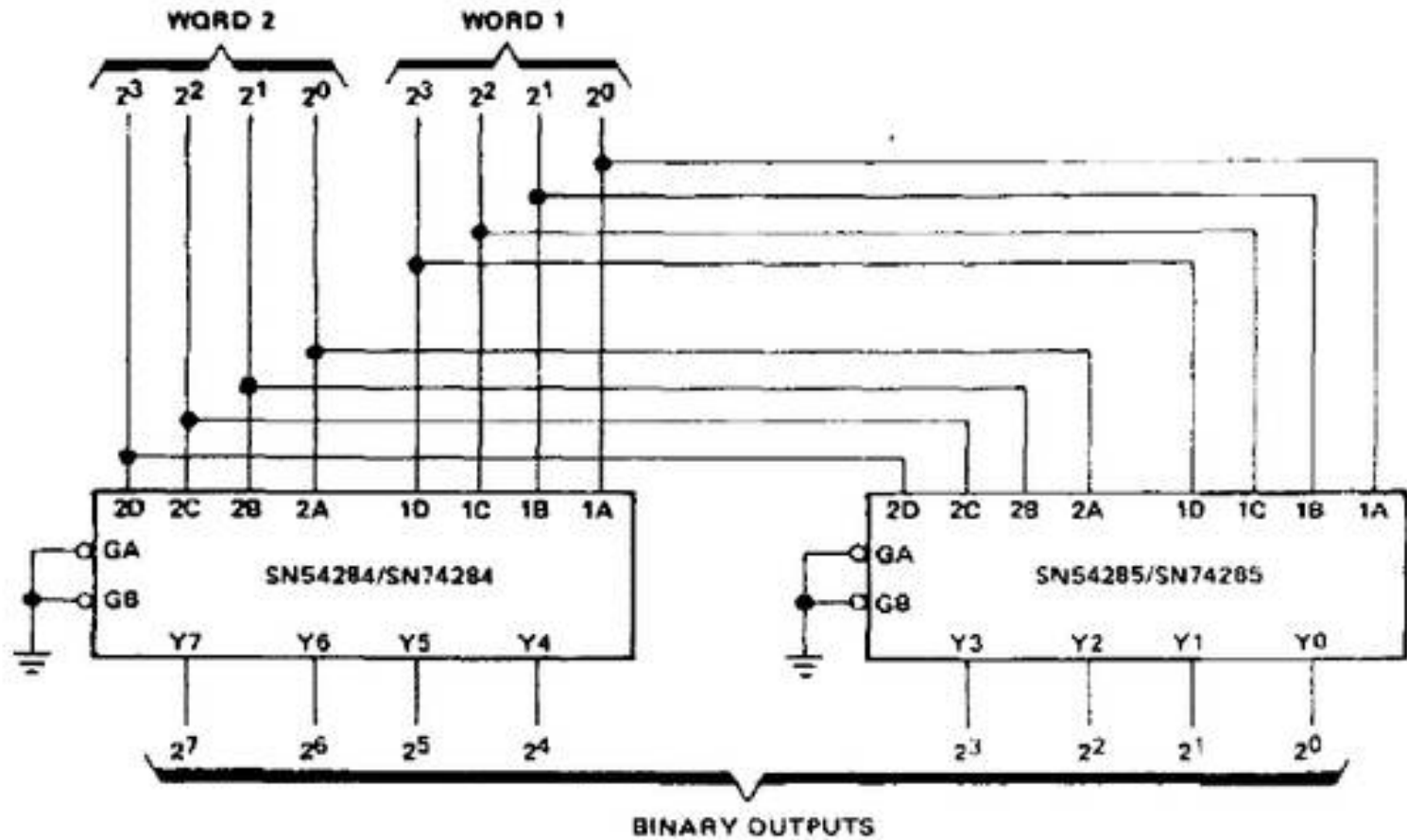


FIGURE A-4 X 4 MULTIPLIER

Splaszczanie bloków funkcjonalnych

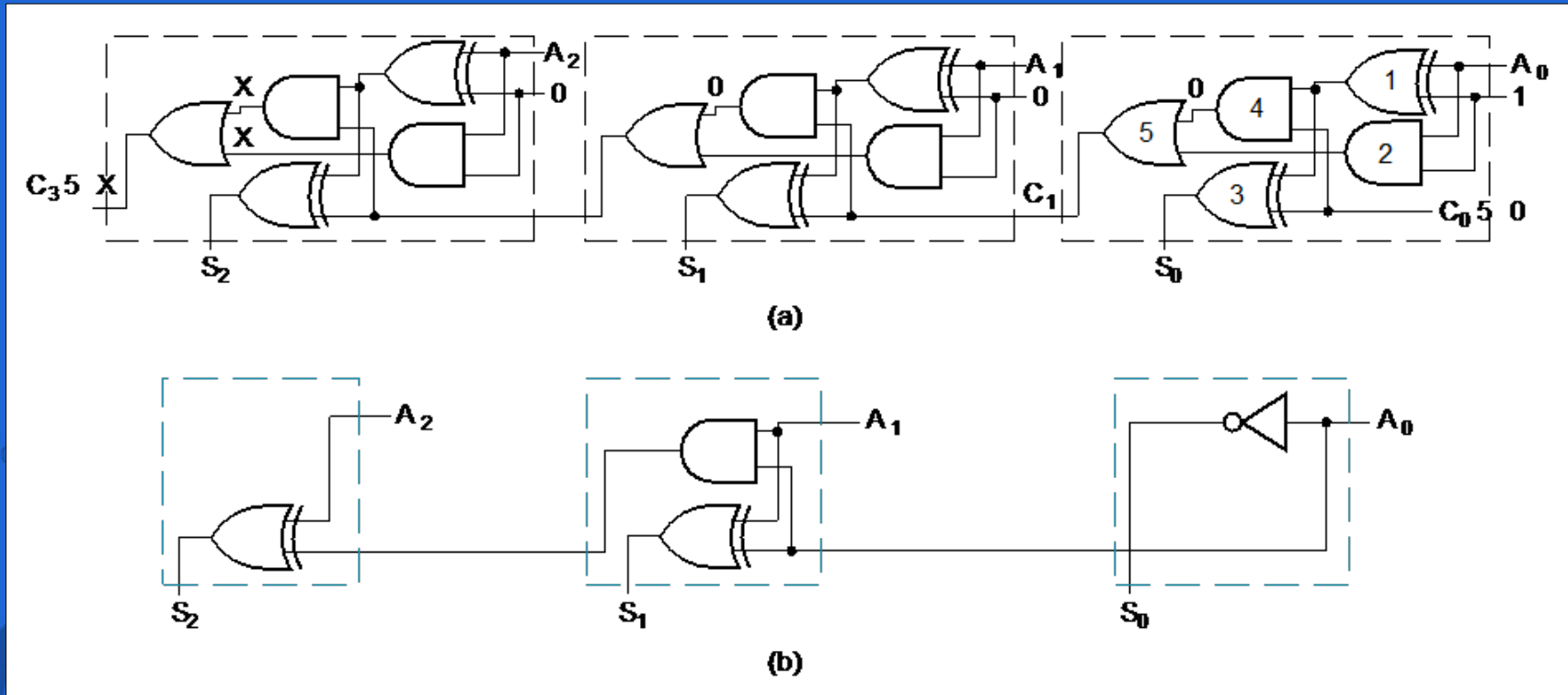
Wygodną techniką projektowania bloków funkcjonalnych jest ich **splaszczanie (kontrakcja)**, polegająca na usuwaniu nadmiarowości poprzez wprowadzanie określonych stałych wejściowych.

Można w ten sposób zrealizować następujące funkcje:

- inkrementację/dekrementację,
- mnożenie/dzielenie przez stałą,
- dopełnianie/uzupełnianie zerami i in.

Nowa funkcja musi być jednak realizowalna przez dany blok funkcjonalny po doprowadzeniu określonych sygnałów wejściowych – tu: stanów logicznych 0 lub 1, nie zaś X lub \bar{X} .

Inkrementer/dekrementer

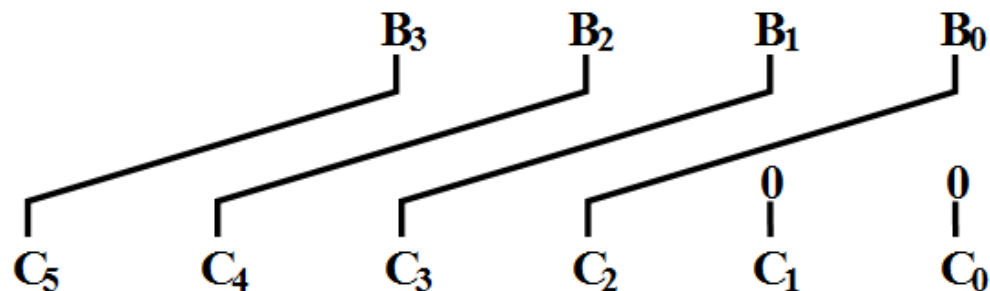


Spłaszczenie sumatora z przeniesieniem szeregowym i wprowadzenie słowa $B = 001$ daje inkrementer 3-bitowy. Wprowadzenie jako B liczby ujemnej daje dekrementer 3-bitowy. Powielenie środkowej komórki pozwala wydłużyć słowo inkrementowane/dekrementowane.

Mnożenie/dzielenie przez 2^n

(a) Mnożenie przez 100

Przesunięcie w lewo o 2

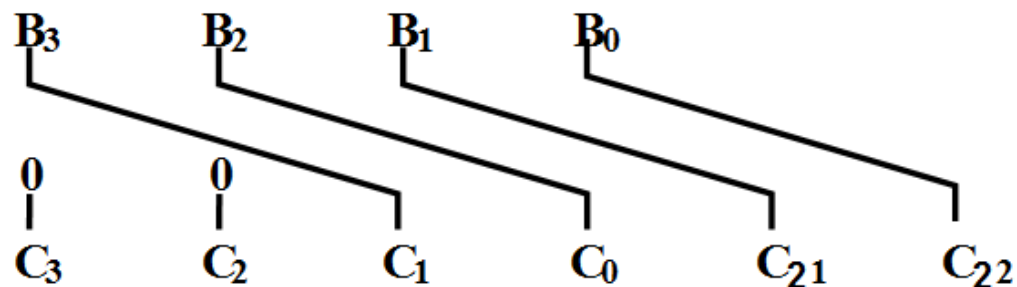


(b) Dzielenie

Przez 100

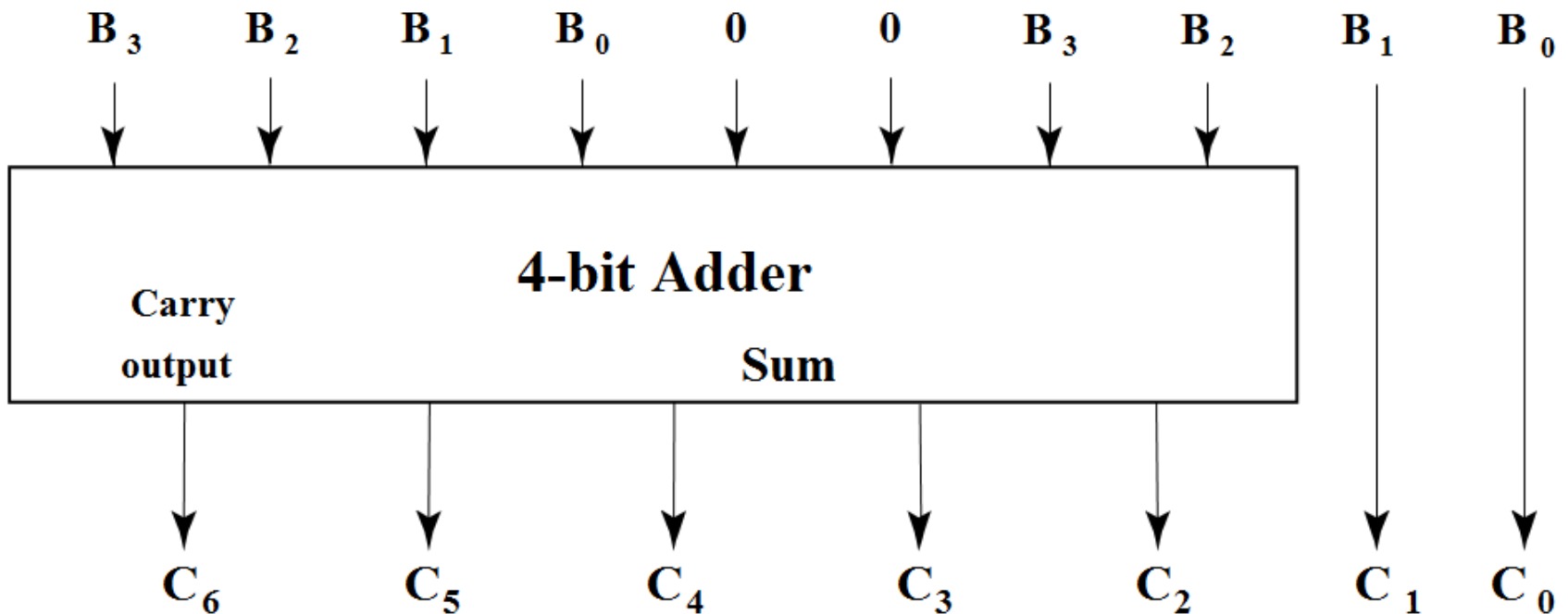
Przesunięcie w prawo o 2

Zachowana reszta



Mnożenie przez stałą

Mnożenie B przez 101



Każde mnożenie/dzielenie binarne można zrealizować jako złożenie przesunięcia i sumowania, tu:

$$B \cdot 101 = (B + 2^{-2} \cdot B) \cdot 2^2$$

Jednostka arytmetyczno-logiczna (ALU)

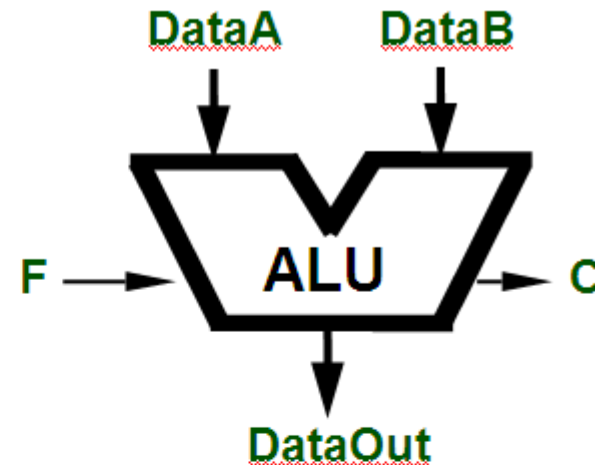
◦ Funkcje ALU

- Dwa argumenty wielobitowe (słowa wejściowe)
- Słowo sterujące **F** określa operację

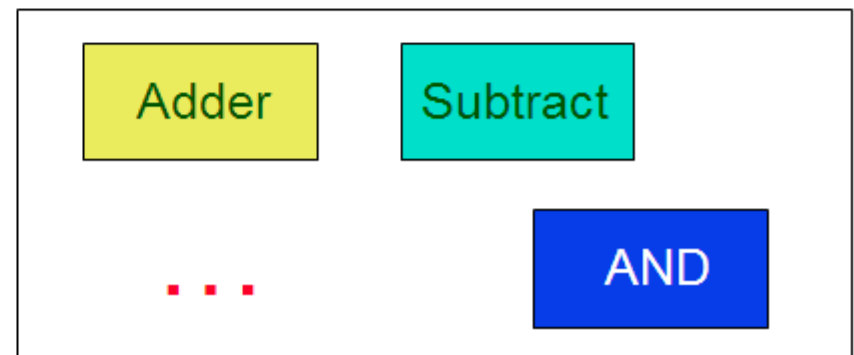
◦ **DataOut** ma tyle bitów co oba słowa wejściowe (**DataA** and **DataB**)

◦ ALU jest układem kombinacyjnym

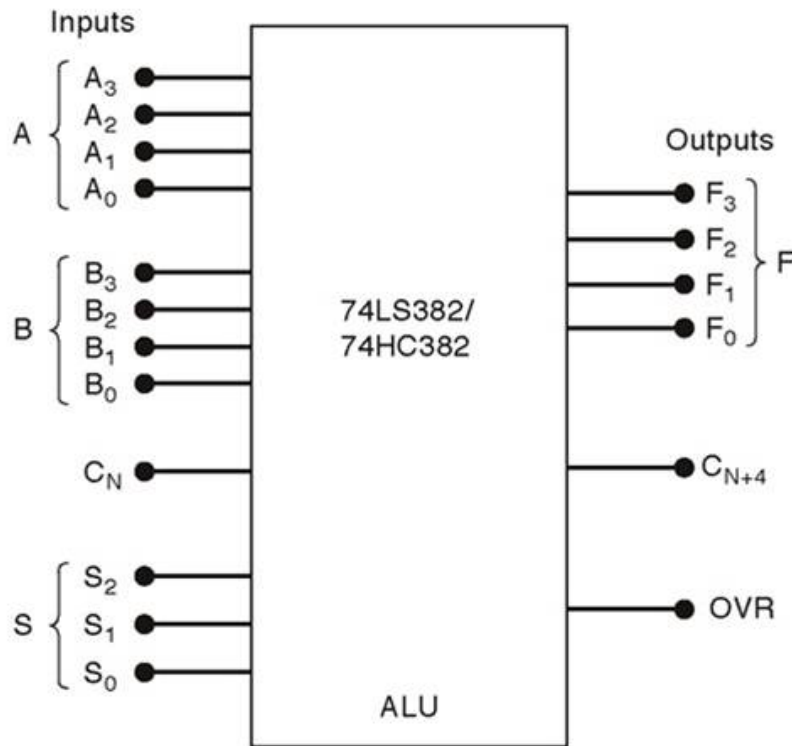
◦ Słowo **C** określa specyficzne wyniki wykonanych operacji (np. przepełnienie).



Traktuj ALU jak zbiór układów logicznych i arytmetycznych w jednym pudełku. Słowo **F** adresuje blok.



Jednostka arytmetyczno-logiczna (ALU)



A = 4-bit input number
 B = 4-bit input number
 C_N = carry into LSB position
 S = 3-bit operation select inputs
 F = 4-bit output number
 C_{N+4} = carry out of MSB position
 OVR = overflow indicator

(a)

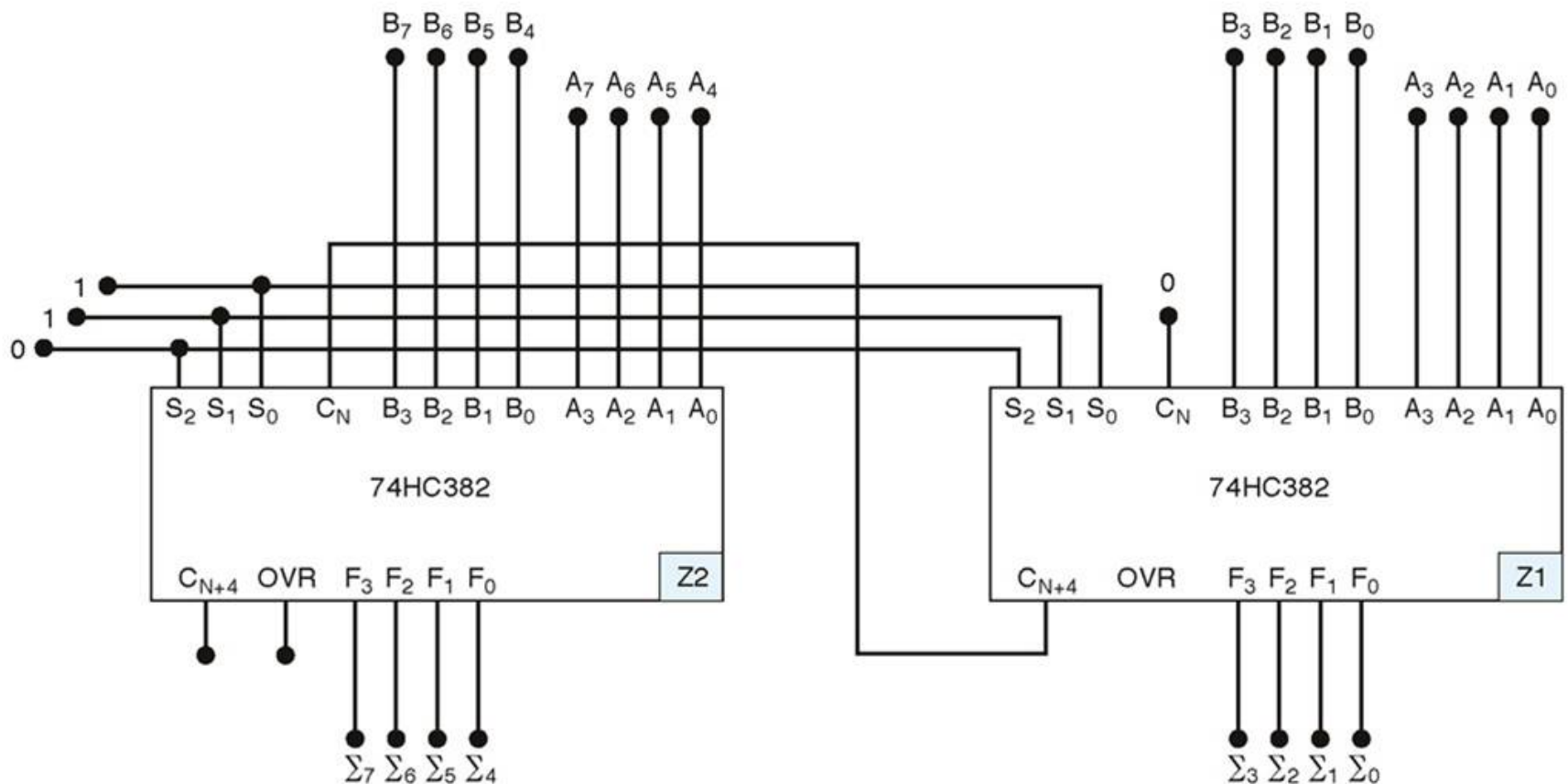
Function Table

| S ₂ | S ₁ | S ₀ | Operation | Comments |
|----------------|----------------|----------------|-----------|--------------------------------------------------------------------|
| 0 | 0 | 0 | CLEAR | F ₃ F ₂ F ₁ F ₀ = 0000 |
| 0 | 0 | 1 | B minus A | Needs C _N = 1 |
| 0 | 1 | 0 | A minus B | |
| 0 | 1 | 1 | A plus B | Needs C _N = 0 |
| 1 | 0 | 0 | A ⊕ B | Exclusive-OR |
| 1 | 0 | 1 | A + B | OR |
| 1 | 1 | 0 | AB | AND |
| 1 | 1 | 1 | PRESET | F ₃ F ₂ F ₁ F ₀ = 1111 |

Notes: S inputs select operation.
 OVR = 1 for signed-number overflow.

(b)

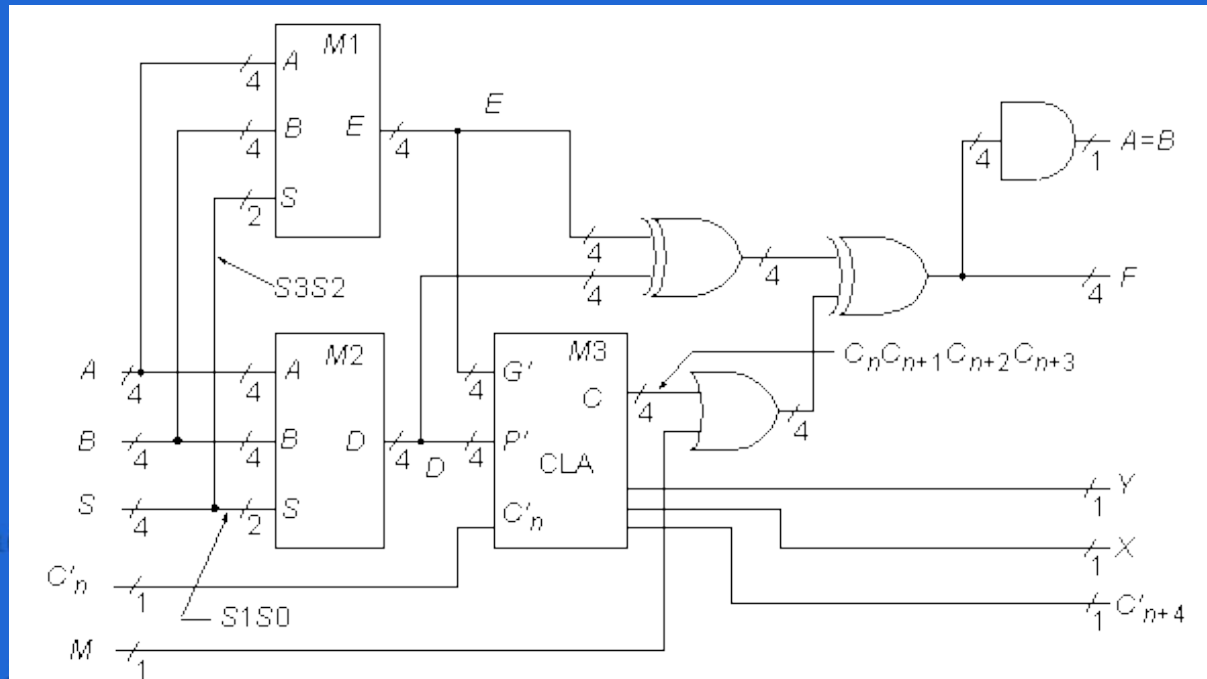
Rozszerzanie linii danych ALU



Notes:

- Z1 adds lower-order bits.
- Z2 adds higher-order bits.
- $\Sigma_7 - \Sigma_0 = 8\text{-bit sum}$.
- OVR of Z2 is 8-bit overflow indicator.

Jednostka arytmetyczno-logiczna (ALU)



| Selection | | | | M = 1 | M = 0, Arithmetic Functions | |
|-----------|----|----|----|---------------------------|-----------------------------------------------|---------------------------------------------------------------|
| S3 | S2 | S1 | S0 | Logic Function | Cn = 0 | Cn = 1 |
| 0 | 0 | 0 | 0 | $F = \text{not } A$ | $F = A \text{ minus } 1$ | $F = A$ |
| 0 | 0 | 0 | 1 | $F = A \text{ nand } B$ | $F = A \text{ B minus } 1$ | $F = A \text{ B}$ |
| 0 | 0 | 1 | 0 | $F = (\text{not } A) + B$ | $F = A (\text{not } B) \text{ minus } 1$ | $F = A (\text{not } B)$ |
| 0 | 0 | 1 | 1 | $F = 1$ | $F = \text{minus } 1$ | $F = \text{zero}$ |
| 0 | 1 | 0 | 0 | $F = A \text{ nor } B$ | $F = A \text{ plus } (A + \text{not } B)$ | $F = A \text{ plus } (A + \text{not } B) \text{ plus } 1$ |
| 0 | 1 | 0 | 1 | $F = \text{not } B$ | $F = A \text{ B plus } (A + \text{not } B)$ | $F = A \text{ B plus } (A + \text{not } B) \text{ plus } 1$ |
| 0 | 1 | 1 | 0 | $F = A \text{ xnor } B$ | $F = A \text{ minus } B \text{ minus } 1$ | $F = (A + \text{not } B) \text{ plus } 1$ |
| 0 | 1 | 1 | 1 | $F = A + \text{not } B$ | $F = A + \text{not } B$ | $F = A \text{ minus } B$ |
| 1 | 0 | 0 | 0 | $F = (\text{not } A) B$ | $F = A \text{ plus } (A + B)$ | $F = (A + \text{not } B) \text{ plus } 1$ |
| 1 | 0 | 0 | 1 | $F = A \text{ xor } B$ | $F = A \text{ plus } B$ | $F = A \text{ plus } (A + B) \text{ plus } 1$ |
| 1 | 0 | 1 | 0 | $F = B$ | $F = A (\text{not } B) \text{ plus } (A + B)$ | $F = A (\text{not } B) \text{ plus } (A + B) \text{ plus } 1$ |
| 1 | 0 | 1 | 1 | $F = A + B$ | $F = (A + B)$ | $F = (A + B) \text{ plus } 1$ |
| 1 | 1 | 0 | 0 | $F = 0$ | $F = A$ | $F = A \text{ plus } A \text{ plus } 1$ |
| 1 | 1 | 0 | 1 | $F = A (\text{not } B)$ | $F = A \text{ B plus } A$ | $F = AB \text{ plus } A \text{ plus } 1$ |
| 1 | 1 | 1 | 0 | $F = A \text{ B}$ | $F = A (\text{not } B) \text{ plus } A$ | $F = A (\text{not } B) \text{ plus } A \text{ plus } 1$ |
| 1 | 1 | 1 | 1 | $F = A$ | $F = A$ | $F = A \text{ plus } 1$ |