

# Logika cyfrowa

## Wykład 3: układy arytmetyczne

---

Marek Materzok

10 marca 2021

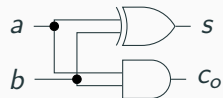
# Dodawanie liczb binarnych

---

Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

$a$	$b$	$c_o$	$s$
0	0		
0	1		
1	0		
1	1		

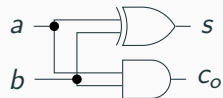


$$\begin{aligned} s &= a \oplus b \\ c_o &= ab \end{aligned}$$

Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + \quad b \\ \hline c \quad s \end{array} \qquad \begin{array}{r} 0 \\ + \quad 0 \\ \hline \end{array}$$

$a$	$b$	$c_o$	$s$
0	0		
0	1		
1	0		
1	1		



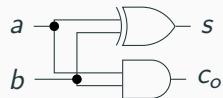
$$\begin{aligned} s &= a \oplus b \\ c_o &= ab \end{aligned}$$

# Półsumator

Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + \quad b \\ \hline c \quad s \end{array} \qquad \begin{array}{r} 0 \\ + \quad 0 \\ \hline 0 \quad 0 \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1		
1	0		
1	1		



$$\begin{aligned} s &= a \oplus b \\ c_o &= ab \end{aligned}$$

# Półsumator

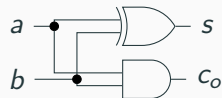
Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1		
1	0		
1	1		



$$s = a \oplus b$$

$$c_o = ab$$

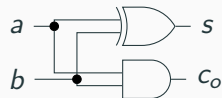
Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline 0 \quad 1 \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1	0	1
1	0		
1	1		



$$s = a \oplus b$$

$$c_o = ab$$

# Półsumator

Układ dodający dwie cyfry binarne:

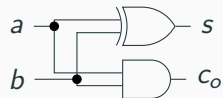
$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 0 \\ \hline \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1	0	1
1	0		
1	1		



$$\begin{aligned} s &= a \oplus b \\ c_o &= ab \end{aligned}$$



# Półsumator

Układ dodający dwie cyfry binarne:

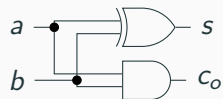
$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 0 \\ \hline 0 \quad 1 \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1		



$$s = a \oplus b$$

$$c_o = ab$$

# Półsumator

Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

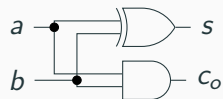
$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 0 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 1 \\ \hline \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1		



$$s = a \oplus b$$

$$c_o = ab$$

Układ dodający dwie cyfry binarne:

$$\begin{array}{r} a \\ + b \\ \hline c \quad s \end{array}$$

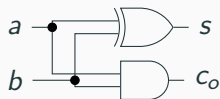
$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \quad 0 \end{array}$$

$$\begin{array}{r} 0 \\ + 1 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 0 \\ \hline 0 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \\ + 1 \\ \hline 1 \quad 0 \end{array}$$

$a$	$b$	$c_o$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



$$s = a \oplus b$$

$$c_o = ab$$

# Pełny sumator

$a$	$b$	$c$	$c_o$	$s$
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

		$yz$			
		00	01	11	10
$x$	0	0	1	0	1
	1	1	0	1	0

		$yz$			
		00	01	11	10
$x$	0	0	0	1	0
	1	0	1	1	1

# Pełny sumator

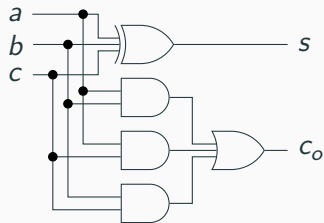
$a$	$b$	$c$	$c_o$	$s$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

		$yz$			
		00	01	11	10
$x$	0	0	1	0	1
	1	1	0	1	0

		$yz$			
		00	01	11	10
$x$	0	0	0	1	0
	1	0	1	1	1

# Pełny sumator

$a$	$b$	$c$	$c_o$	$s$
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

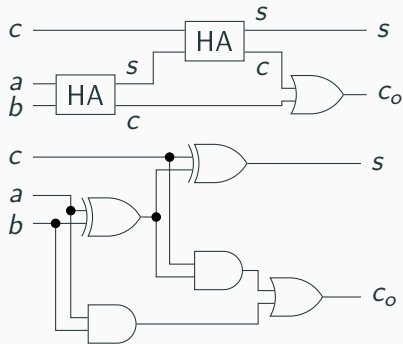


$$s = a \oplus b \oplus c$$

$$c_o = ab + ac + bc$$

## Pełny sumator z półsumatorów

$a$	$b$	$c$	$c_o$	$s$
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



# Sumator szeregowy

$$\begin{array}{rccccccccc} & & a_3 & a_2 & a_1 & a_0 & & & & \\ + & b_3 & b_2 & b_1 & b_0 & & & & & \\ \hline c_4 & s_3 & s_2 & s_1 & s_0 & & & & & \end{array}$$

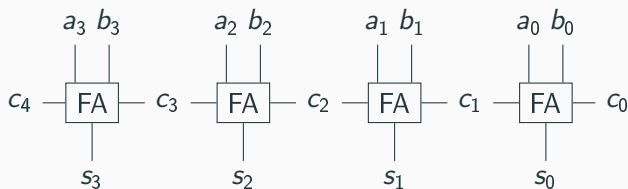


# Sumator szeregowy

$$\begin{array}{rccccccccc} & c_4 & & c_3 & & c_2 & & c_1 & & c_0 \\ & & & a_3 & & a_2 & & a_1 & & a_0 \\ + & & & b_3 & & b_2 & & b_1 & & b_0 \\ \hline c_4 & & s_3 & & s_2 & & s_1 & & s_0 \end{array}$$

# Sumator szeregowy

$$\begin{array}{r} c_4 \quad c_3 \quad c_2 \quad c_1 \quad c_0 \\ \quad a_3 \quad a_2 \quad a_1 \quad a_0 \\ + \quad b_3 \quad b_2 \quad b_1 \quad b_0 \\ \hline c_4 \quad s_3 \quad s_2 \quad s_1 \quad s_0 \end{array}$$



# Liczby ujemne – znak-moduł

- Najbardziej znaczący bit określa znak, pozostałe bity – moduł (wartość bezwzględna).

$s$	$b_2$	$b_1$	$b_0$
-----	-------	-------	-------

$$(2)_{10} = 0010, (-2)_{10} = 1010$$

- Łatwa zmiana znaku – negacja bitu znaku.
- Dwie reprezentacje zera –  $000 \dots 0$  i  $100 \dots 0$
- Trudne dodawanie:
  - Jeśli obie liczby mają ten sam znak, dodajemy moduły.
  - Jeśli mają różne znaki, wynikowy znak jest taki, jak znak liczby o **większym** module. Moduły odejmujemy „pisemnie” mniejszy od większego.

## Przykład – znak-moduł

-5 (1101) i 2 (0010)

odejmujemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 0 \\ \hline 0 \ 1 \ 1 \end{array}$$

Wynik: -3 (1011)

## Przykład – znak-moduł

-5 (1101) i 2 (0010)

odejmujemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 0 \\ \hline 0 \ 1 \ 1 \end{array}$$

Wynik: -3 (1011)

5 (0101) i -2 (1010)

odejmujemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 0 \\ \hline 0 \ 1 \ 1 \end{array}$$

Wynik: 3 (0011)

## Przykład – znak-moduł

-5 (1101) i 2 (0010)  
odejmujemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 0 \\ \hline 0 \ 1 \ 1 \end{array}$$

Wynik: -3 (1011)

5 (0101) i -2 (1010)  
odejmujemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ - \ 0 \ 1 \ 0 \\ \hline 0 \ 1 \ 1 \end{array}$$

Wynik: 3 (0011)

-5 (1101) i -2 (1010)  
dodajemy moduły:

$$\begin{array}{r} 1 \ 0 \ 1 \\ + \ 0 \ 1 \ 0 \\ \hline 1 \ 1 \ 1 \end{array}$$

Wynik: -7 (1111)

## Liczby ujemne – kod uzupełnień do jedności

- Najbardziej znaczący bit określa znak.

Reprezentacją liczby  $-n$  w  $k$  bitach jest  $2^k - 1 - n$ .

$$(2)_{10} = 0010, (-2)_{10} = 1101$$

- Łatwa zmiana znaku – negacja **wszystkich** bitów.

$$(2^k - 1)_{10} = (\underbrace{11 \dots 1}_k)_2$$

- Dwie reprezentacje zera –  $00 \dots 0$  i  $11 \dots 1$
- Komplikacja z dodawaniem – czasem potrzebna jest korekta wyniku

## Przykład – kod uzupełnień do jedności

-5 (1010) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Wynik: -3 (1100)



## Przykład – kod uzupełnień do jedności

-5 (1010) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Wynik: -3 (1100)

5 (0101) i -2 (1101)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \end{array}$$

## Przykład – kod uzupełnień do jedności

-5 (1010) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Wynik: -3 (1100)

5 (0101) i -2 (1101)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \\ \hookrightarrow & & & & & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array}$$

Wynik: 3 (0011)

## Przykład – kod uzupełnień do jedności

-5 (1010) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Wynik: -3 (1100)

5 (0101) i -2 (1101)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \\ \hookrightarrow & & & & & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array}$$

Wynik: 3 (0011)

-5 (1010) i -2 (1101)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 & 1 \end{array}$$

Wynik: -7 (1000)

## Przykład – kod uzupełnień do jedności

-5 (1010) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

Wynik: -3 (1100)

5 (0101) i -2 (1101)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 1 & 0 \\ \hookrightarrow & & & & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array}$$

Wynik: 3 (0011)

-5 (1010) i -2 (1101)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 & 1 \\ \hookrightarrow & & & & 1 \\ \hline & 1 & 0 & 0 & 0 \end{array}$$

Wynik: -7 (1000)

## Liczby ujemne – kod uzupełnień do dwóch

- Reprezentacją liczby  $-n$  w  $k$  bitach jest  $2^k - n$ .  
 $(2)_{10} = 0010$ ,  $(-2)_{10} = 1110$
- Zmiana znaku – negacja wszystkich bitów i **dodanie jedynki**.  
 $2^k - n = (2^k - 1 - n) + 1$
- Tylko jedna reprezentacja zera, ale
- Istnieje liczba bez przeciwnej,  $-2^{k-1}$ :  
 $(-2^{k-1})_{10} = (2^k - (2^{k-1}))_{10} = (2^{k-1})_1 0 = 10 \dots 0$
- Dodawanie takie samo, jak dodawanie bez znaku

## Przykład – kod uzupełnień do dwóch

-5 (1011) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

Wynik: -3 (1101)

## Przykład – kod uzupełnień do dwóch

-5 (1011) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

Wynik: -3 (1101)

5 (0101) i -2 (1110)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

Wynik: 3 (0011)

## Przykład – kod uzupełnień do dwóch

-5 (1011) i 2 (0010)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

Wynik: -3 (1101)

5 (0101) i -2 (1110)

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

Wynik: 3 (0011)

-5 (1011) i -2 (1110)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

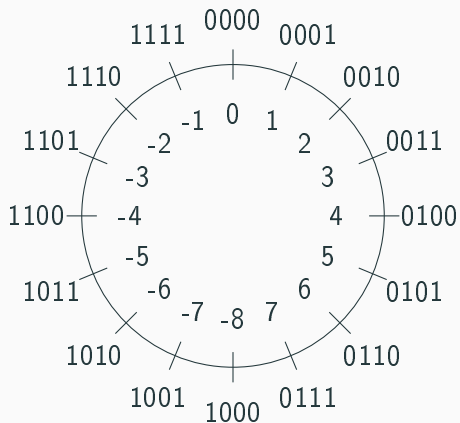
Wynik: -7 (1001)



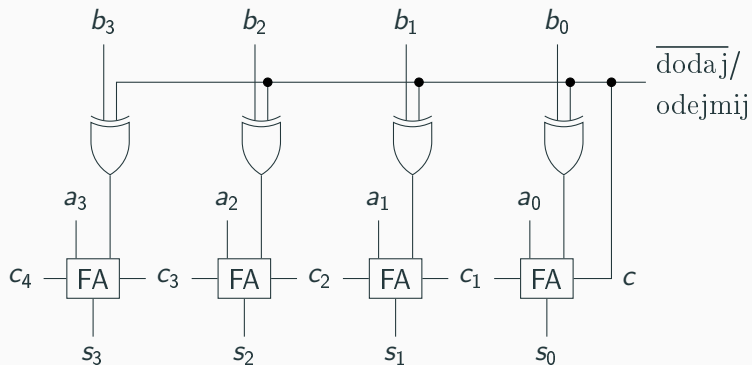
## Podsumowanie – kody liczb ze znakiem

bin	ZM	U1	U2
0000	0	0	0
⋮	⋮	⋮	⋮
0111	7	7	7
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

## Kody uzupełnień do dwóch – operacje modulo



## Układ dodajco-odejmujący



## Wykrywanie przepętnienia

$$6 - 2 = 4$$

1	1	1	0	
<hr/>				
	0	1	1	0
+	1	1	1	0
<hr/>				
1	0	1	0	0

## Wykrywanie przepętnienia

$$6 - 2 = 4$$

1	1	1	0	
<hr/>				
	0	1	1	0
+	1	1	1	0
<hr/>				
1	0	1	0	0

$$6 + 3 = 9 \approx -7$$

0	1	1	0	
<hr/>				
	0	1	1	0
+	0	0	1	1
<hr/>				
0	1	0	0	1

## Wykrywanie przepełnienia

$$6 - 2 = 4$$

1	1	1	0	
<hr/>				
	0	1	1	0
+	1	1	1	0
<hr/>				
1	0	1	0	0

$$6 + 3 = 9 \approx -7$$

0	1	1	0	
<hr/>				
	0	1	1	0
+	0	0	1	1
<hr/>				
0	1	0	0	1

$$-6 - 5 = -11 \approx 5$$

1	0	1	0	
<hr/>				
	1	0	1	0
+	1	0	1	1
<hr/>				
1	0	1	0	1

## Wykrywanie przepełnienia

$$6 - 2 = 4$$

$$\begin{array}{rcccc} 1 & 1 & 1 & 0 & \\ \hline & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 1 & 0 \\ \hline 1 & 0 & 1 & 0 & 0 \end{array}$$

$$6 + 3 = 9 \approx -7$$

$$\begin{array}{rcccc} 0 & 1 & 1 & 0 & \\ \hline & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 \end{array}$$

$$-6 - 5 = -11 \approx 5$$

$$\begin{array}{rcccc} 1 & 0 & 1 & 0 & \\ \hline & 1 & 0 & 1 & 0 \\ + & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 1 \end{array}$$

- Przepełnienie gdy dodajemy liczby z **tym samym znakiem**, a wynik ma **inny znak**

## Wykrywanie przepełnienia

$$6 - 2 = 4$$

	1	1	1	0
	0	1	1	0
+	1	1	1	0
	1	0	1	0

$$6 + 3 = 9 \approx -7$$

	0	1	1	0
	0	1	1	0
+	0	0	1	1
	0	1	0	0

$$-6 - 5 = -11 \approx 5$$

	1	0	1	0
	1	0	1	0
+	1	0	1	1
	1	0	1	0

- Przepełnienie gdy dodajemy liczby z **tym samym znakiem**, a wynik ma **inny znak**
- Można to rozpoznać przy użyciu bitów przeniesienia



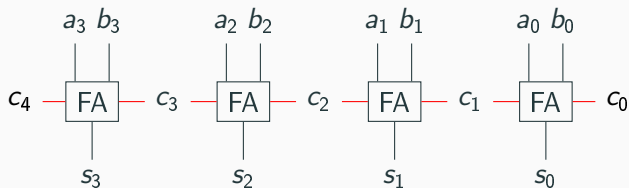
## Sumator szybki

---

# Ścieżka krytyczna

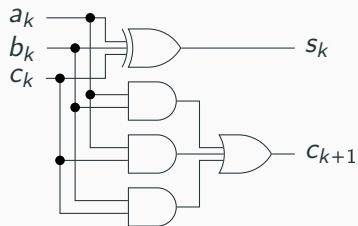
Ścieżka najdłuższego opóźnienia w układzie kombinacyjnym.

W sumatorze wielobitowym – ścieżka przeniesienia:



Długość ścieżki krytycznej –  $2n$  bramek

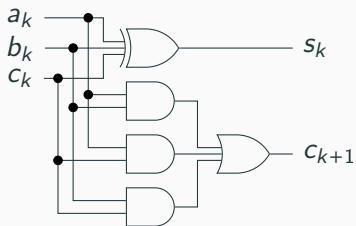
# Układ przewidywania przeniesienia



$$s_k = a_k \oplus b_k \oplus c_k$$

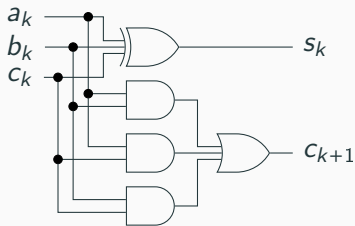
$$c_{k+1} = a_k b_k + a_k c_k + b_k c_k$$

# Układ przewidywania przeniesienia



$$\begin{aligned} s_k &= a_k \oplus b_k \oplus c_k \\ c_{k+1} &= a_k b_k + a_k c_k + b_k c_k \\ &= a_k b_k + (a_k + b_k) c_k \end{aligned}$$

# Układ przewidywania przeniesienia



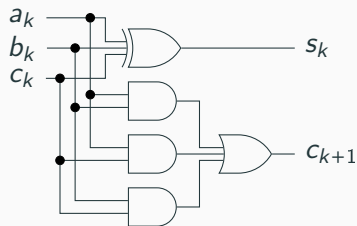
$$s_k = a_k \oplus b_k \oplus c_k$$

$$\begin{aligned} c_{k+1} &= a_k b_k + a_k c_k + b_k c_k \\ &= a_k b_k + (a_k + b_k) c_k \\ &= g_k + p_k c_k \end{aligned}$$

Bit *generowania*:  $g_k = a_k b_k$

Bit *propagacji*:  $p_k = a_k + b_k$

## Układ przewidywania przeniesienia



$$s_k = a_k \oplus b_k \oplus c_k$$

$$\begin{aligned} c_{k+1} &= a_k b_k + a_k c_k + b_k c_k \\ &= a_k b_k + (a_k + b_k) c_k \\ &= g_k + p_k c_k \end{aligned}$$

Bit *generowania*:  $g_k = a_k b_k$

Bit *propagacji*:  $p_k = a_k + b_k$

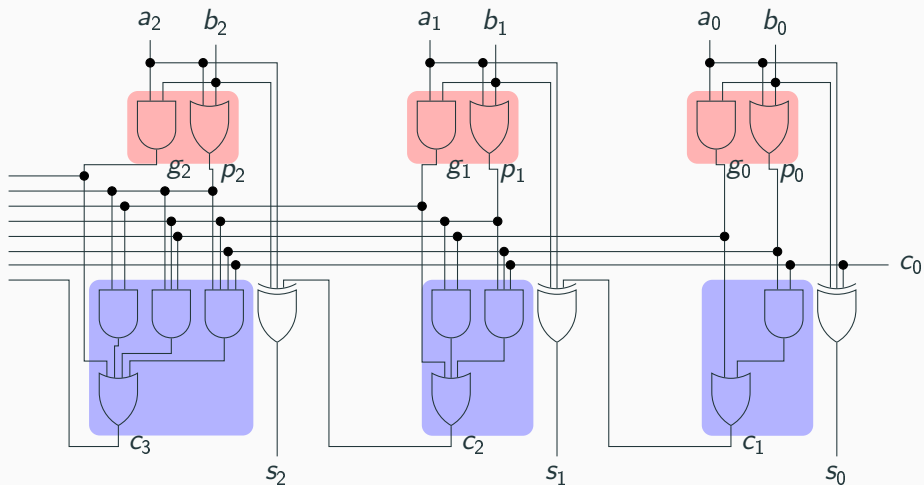
$$\begin{aligned} c_n &= g_{n-1} + p_{n-1} c_{n-1} \\ &= g_{n-1} + p_{n-1} (g_{n-2} + p_{n-2} c_{n-2}) \\ &= g_{n-1} + p_{n-1} g_{n-2} + p_{n-1} p_{n-2} c_{n-2} = \dots \\ &= g_{n-1} + p_{n-1} g_{n-2} + p_{n-1} p_{n-2} g_{n-3} + \dots + p_{n-1} p_{n-2} \dots p_0 c_0 \\ &= \sum_{i=0}^{n-1} g_i \prod_{j=i+1}^{n-1} p_j + c_0 \prod_{j=0}^{n-1} p_j = \sum_{i=-1}^{n-1} g_i \prod_{j=i+1}^{n-1} p_j \end{aligned}$$

## Układ przewidywania przeniesienia – charakterystyka

$$c_n = \sum_{i=-1}^{n-1} g_i \prod_{j=i+1}^{n-1} p_j$$

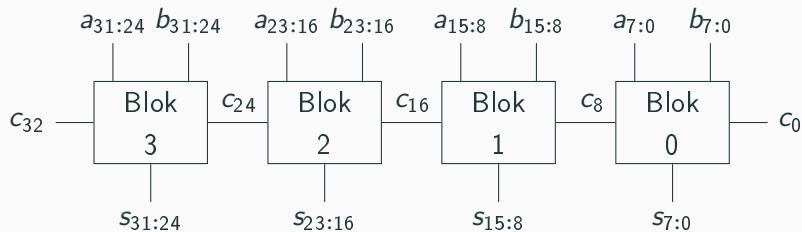
- $n + 1$  bramek, ścieżka krytyczna – 2 bramki, max  $n + 1$ -wejściowe
- W praktyce ograniczenie *fan-in*, dla bramek 2-wejściowych:  $O(n^2)$  bramek, ścieżka krytyczna –  $O(\log n)$  bramek
- Dla wysokich  $n$  skomplikowane, nieplanarne połączenia

## Sumator z przewidywaniem przeniesienia





## Łączenie sumatorów równoległych



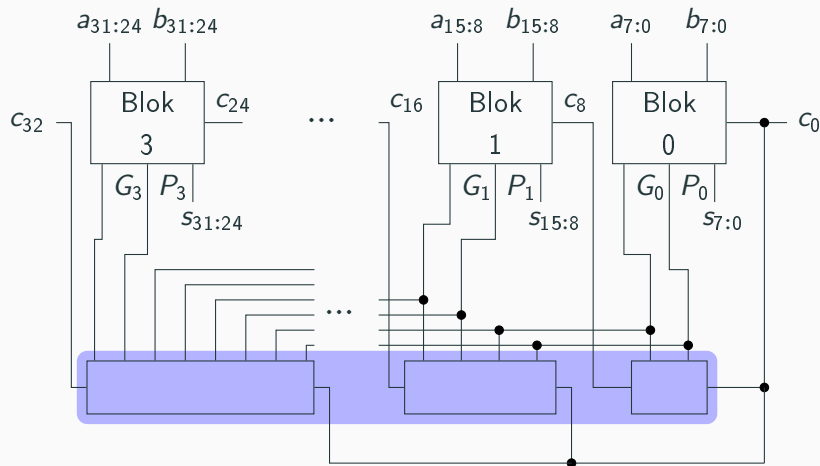
Zamiast łączyć szeregowo – wprowadzić drugi poziom przewidywania przeniesienia:

$$c_n = \sum_{i=0}^{n-1} g_i \prod_{j=i+1}^{n-1} p_j + c_0 \prod_{j=0}^{n-1} p_j = G_0 + P_0 c_0$$

Dla bloków  $n$ -bitowych:

$$\begin{aligned} G_k &= \sum_{i=kn}^{(k+1)n-1} g_i \prod_{j=i+1}^{(k+1)n-1} p_j \\ P_k &= \prod_{j=kn}^{(k+1)n-1} p_j \\ c_{kn} &= \sum_{i=0}^{k-1} G_i \prod_{j=i+1}^{k-1} P_j + c_0 \prod_{j=0}^{k-1} P_j \end{aligned}$$

# Sumator hierarchiczny



## Układy mnożące

---

# Przesunięcie bitowe

- Przesunięcie *logiczne*: wypełnia luki zerami

$$11001 \gg 2 =$$

$$11001 \ll 2 =$$

- Przesunięcie *arytmetyczne*: wypełnia lukę MSB bitem znaku

$$11001 \ggg 2 =$$

$$11001 \lll 2 =$$

- Przesunięcie *cykliczne* (obrót): wypełnia lukę uciętymi bitami

$$11001 \text{ ROR } 2 =$$

$$11001 \text{ ROL } 2 =$$

# Przesunięcie bitowe

- Przesunięcie *logiczne*: wypełnia luki zerami

$$11001 \gg 2 = 00110$$

$$11001 \ll 2 = 00100$$

- Przesunięcie *arytmetyczne*: wypełnia lukę MSB bitem znaku

$$11001 \ggg 2 =$$

$$11001 \lll 2 =$$

- Przesunięcie *cykliczne* (obrót): wypełnia lukę uciętymi bitami

$$11001 \text{ ROR } 2 =$$

$$11001 \text{ ROL } 2 =$$

# Przesunięcie bitowe

- Przesunięcie *logiczne*: wypełnia luki zerami

$$11001 \gg 2 = 00110$$

$$11001 \ll 2 = 00100$$

- Przesunięcie *arytmetyczne*: wypełnia lukę MSB bitem znaku

$$11001 \ggg 2 = 11110$$

$$11001 \lll 2 = 00100$$

- Przesunięcie *cykliczne* (obrót): wypełnia lukę uciętymi bitami

$$11001 \text{ ROR } 2 =$$

$$11001 \text{ ROL } 2 =$$

# Przesunięcie bitowe

- Przesunięcie *logiczne*: wypełnia luki zerami

$$11001 \gg 2 = 00110$$

$$11001 \ll 2 = 00100$$

- Przesunięcie *arytmetyczne*: wypełnia lukę MSB bitem znaku

$$11001 \ggg 2 = 11110$$

$$11001 \lll 2 = 00100$$

- Przesunięcie *cykliczne* (obrót): wypełnia lukę uciętymi bitami

$$11001 \text{ ROR } 2 = 01110$$

$$11001 \text{ ROL } 2 = 00111$$



## Binarne mnożenie pisemne

Przykład:  $14 \times 5 = 70$

$$\begin{array}{r} \phantom{\times} 1110 \\ \times \phantom{1110} 101 \\ \hline \end{array}$$

## Binarne mnożenie pisemne

Przykład:  $14 \times 5 = 70$

$$\begin{array}{r} \phantom{+} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ + \phantom{1} \phantom{1} \phantom{1} \phantom{1} \phantom{0} \\ \hline \end{array}$$

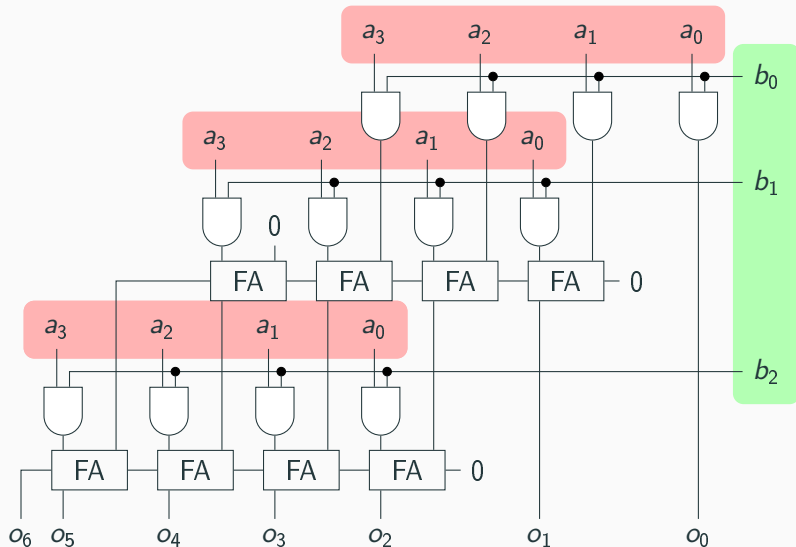
Przykład:  $14 \times 5 = 70$

[illegible]

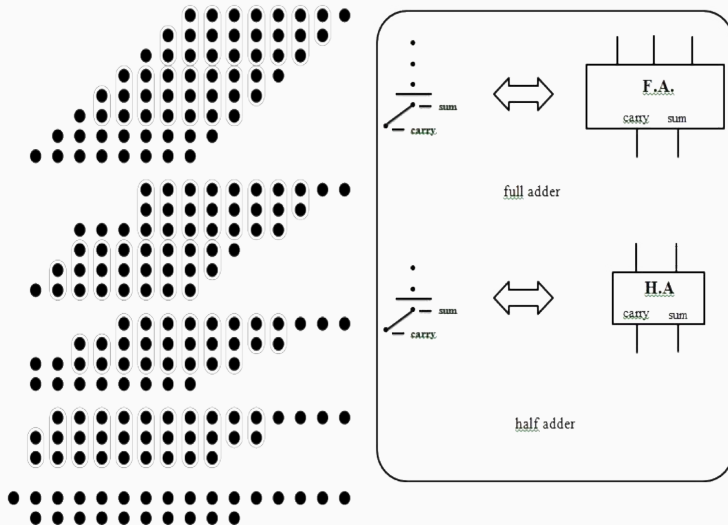
# Binarne mnożenie pisemne

$$\begin{array}{r}
 \begin{array}{cccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & & & \\
 & & & & b_2 & b_1 & b_0 \\
 \hline
 & & & a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
 & & a_3 b_1 & a_2 b_1 & a_1 b_1 & a_0 b_1 & \\
 & + & a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 & \\
 \hline
 o_6 & o_5 & o_4 & o_3 & o_2 & o_1 & o_0
 \end{array}
 \end{array}$$

# Układ mnożenia pisemnego



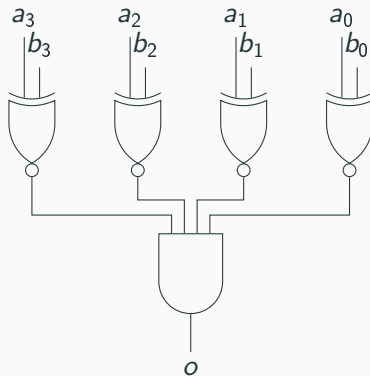
# Szybsze mnożenie – drzewo Wallace'a



## Układy porównujące

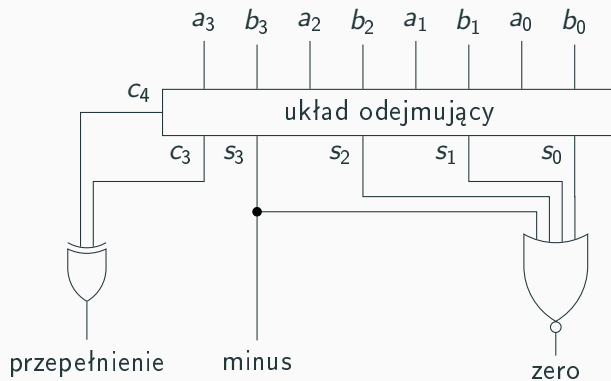
---

# Test równości





# Test nierówności



# Systemy liczbowe

---

# Liczby stałoprzecinkowe

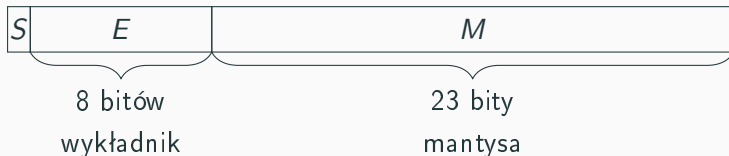
Dzielimy liczbę na części przed i po „kropce”:

$$B = b_{n-1}b_{n-2} \dots b_1b_0.b_{-1}b_{-2} \dots b_{-k}$$

- Wartość:  $\sum_{i=-k}^{n-1} b_i \times 2^i$
- Dodawanie i odejmowanie jak dla liczb całkowitych
- Mnożenie i dzielenie wymaga przesunięcia przecinka

# Liczby zmiennoprzecinkowe

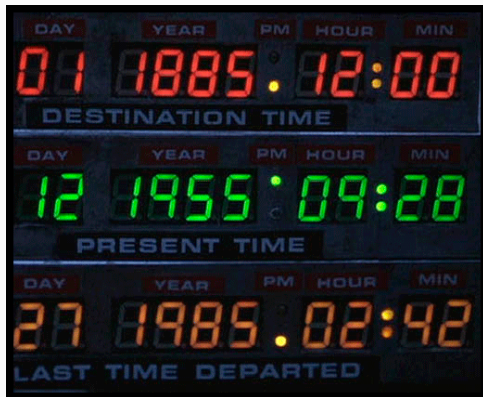
IEEE 754, pojedyncza precyzja (*float*):



- Wartość:  $(-1)^S \times 1.M \times 2^{E-127}$
- Jeśli  $E = 0$  i  $M = 0$ , 0 lub  $-0$
- Jeśli  $E = 255$  i  $M = 0$ ,  $\infty$  lub  $-\infty$
- Jeśli  $E = 255$  i  $M \neq 0$ , NaN (Not a Number)

# Liczby BCD – Binary Coded Decimal

- System dziesiętny kodowany binarnie
- Jedna cyfra dziesiętna reprezentowana przez 4 bity
- Wartości 10-15 nie kodują cyfr



cyfra	kod
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

# Arytmetyka w SystemVerilogu

---

# Operatory arytmetyczne

## Operatory binarne:

- Dodawanie: +
- Odejmowanie: -
- Mnożenie: \*
- Dzielenie: /
- Modulo: %
- Potęgowanie: \*\*

## Operatory unarne:

- Plus: +
- Negacja: -

- Wynik ma tyle bitów, co **większy** z parametrów
- Wartość  $x$  na dowolnym z bitów wejścia ustala **wszystkie** bity wyjścia na  $x$
- Operacje są domyślnie **bez znaku**
- Brak kontroli nad obwodem implementującym operację

- Reprezentacja: kod uzupełnień do dwóch
- W stałych: litera s, na przykład: 8'shfe, 4'sd6
- Przy deklaracji portu/drutu: słowo kluczowe **signed**
- Rzutowanie: funkcje \$signed i \$unsigned
- Aby operacja binarna była ze znakiem, **obydwa parametry** muszą być ze znakiem



```
module dzielenie(  
    output signed [7:0] o,  
    input signed [7:0] a,  
    input signed [7:0] b  
);  
    assign o = a / b;  
endmodule
```

- Wartości bez znaku są rozszerzane przez dopisanie zer (*zero extension*)
- Wartości ze znakiem są rozszerzane przez powtórzenie bitu znaku (*sign extension*)

```
module rozszerzanie(  
    output signed[15:0] o,  
    input signed[7:0] i  
);  
    assign o = i;  
endmodule
```

# Operatory porównania

- Test nierówności: <, <=, >, >=
- Test równości: ==, !=
- Mniejszy argument jest **rozszerzany** do rozmiaru większego
- Test nierówności uwzględnia znak, gdy **obydwa** argumenty są ze znakiem
- Wynik jest jednobitowy

```
module porownanie(  
    output signed o,  
    input signed [7:0] a, b  
);  
    assign o = a < b;  
endmodule
```

- Przesunięcia logiczne:  $\ll$ ,  $\gg$
- Przesunięcia arytmetyczne:  $\lll$ ,  $\ggg$
- $\lll$  jest równoważny  $\ll$
- $\ggg$  uzupełnia bitem znaku liczby ze znakiem, liczby bez znaku uzupełnia zerami

## SystemVerilog – różności

---

## Łączenie portów przez nazwy

```
module fulladder(  
    output o, co,  
    input a, b, c  
);  
    logic t, c1, c2;  
    halfadder ha1(.o(t), .c(c1), .a(a), .b(b));  
    halfadder ha2(.o(o), .c(c2), .a(t), .b(c));  
    assign co = c1 | c2;  
endmodule
```

SystemVerilog zezwala na `.o()` zamiast `.o(o)`, ale Yosys nie obsługuje tej składni ☹

## Funkcje – zwracanie wartości

```
module fulladder(  
    output o, co,  
    input a, b, c  
);  
    logic t, c1, c2;  
    function [1:0] ha(input a, b);  
        ha = ({a^b, a&b});  
    endfunction  
    assign {t, c1} = ha(a, b);  
    assign {o, c2} = ha(t, c);  
    assign co = c1 | c2;  
endmodule
```

## Funkcje – parametry wyjściowe

```
module fulladder(  
    output o, co,  
    input a, b, c  
);  
    logic t, c1, c2;  
    function ha(input a, b, output c);  
        ha = a^b;  
        c = a&b;  
    endfunction  
    assign t = ha(a, b, c1);  
    assign o = ha(t, c, c2);  
    assign co = c1 | c2;  
endmodule
```