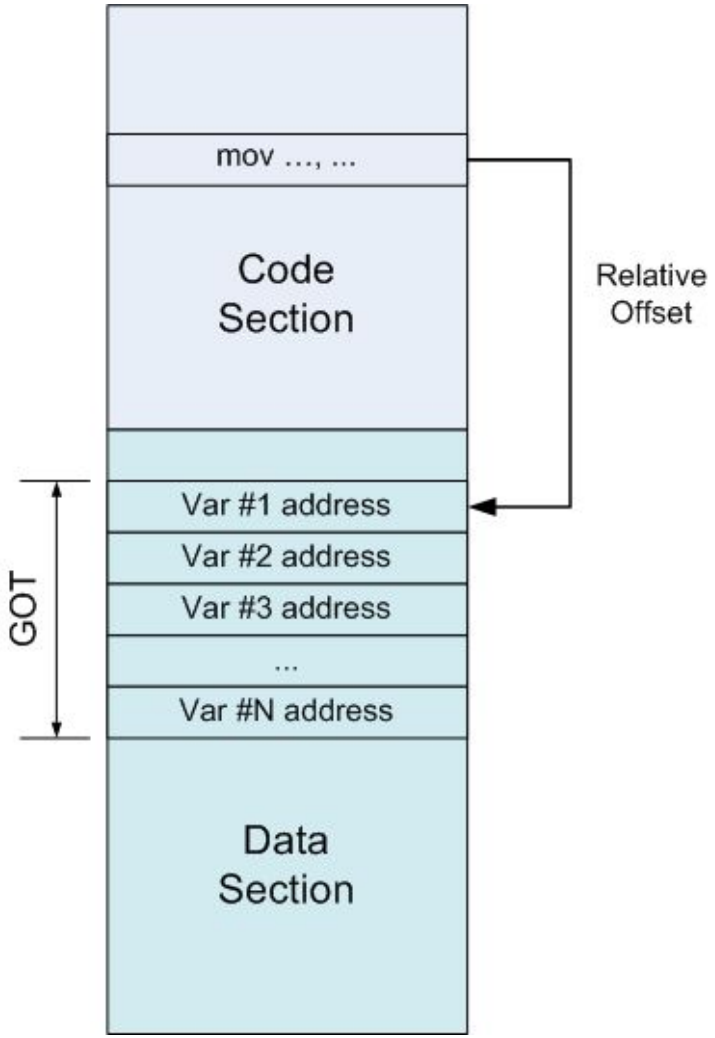
Algorytmy arytmetyki komputerowej

Dodawanie i mnożenie liczb całkowitoliczbowych

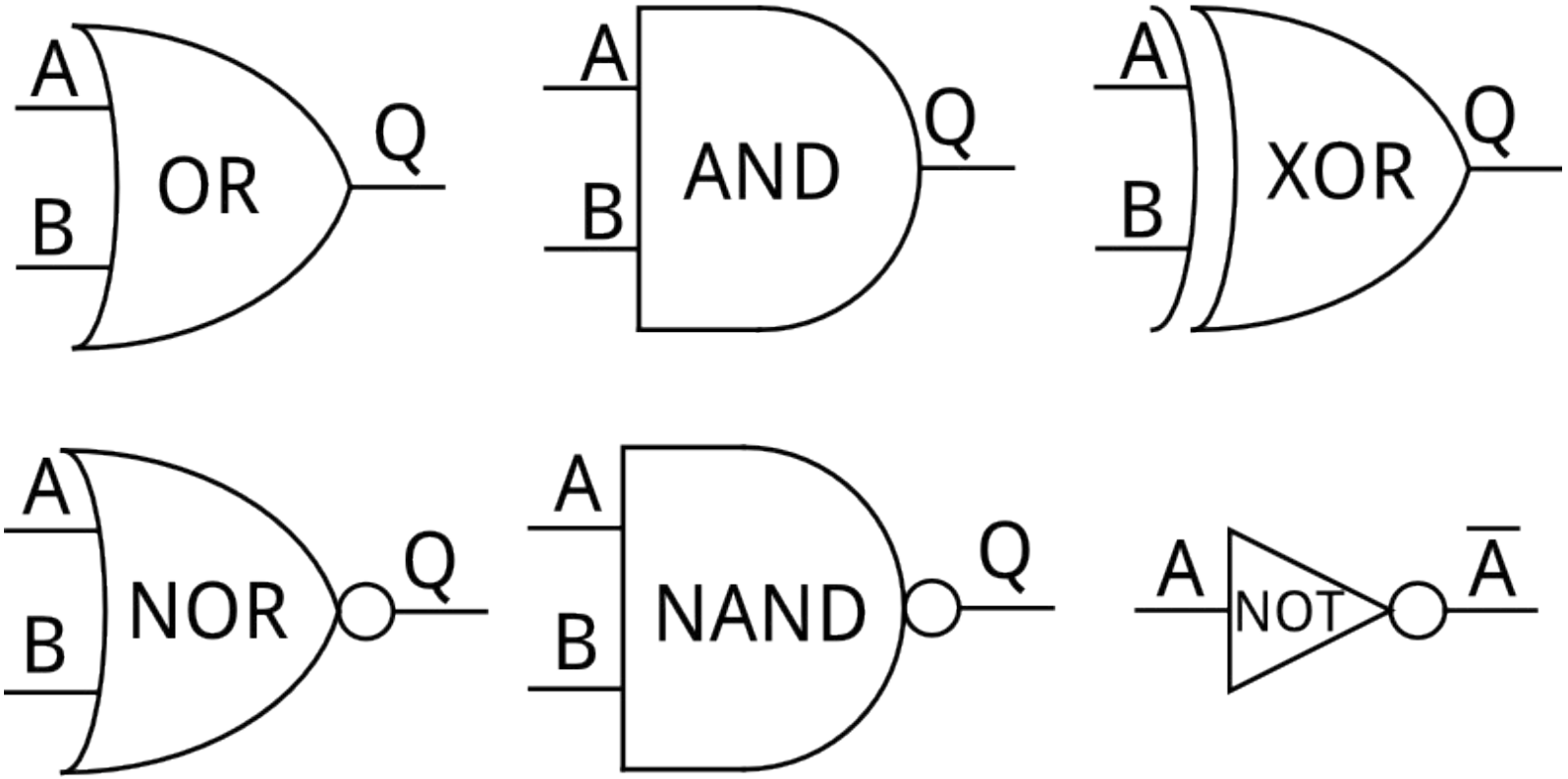
Julian Pszczołowski

Wykorzystanie

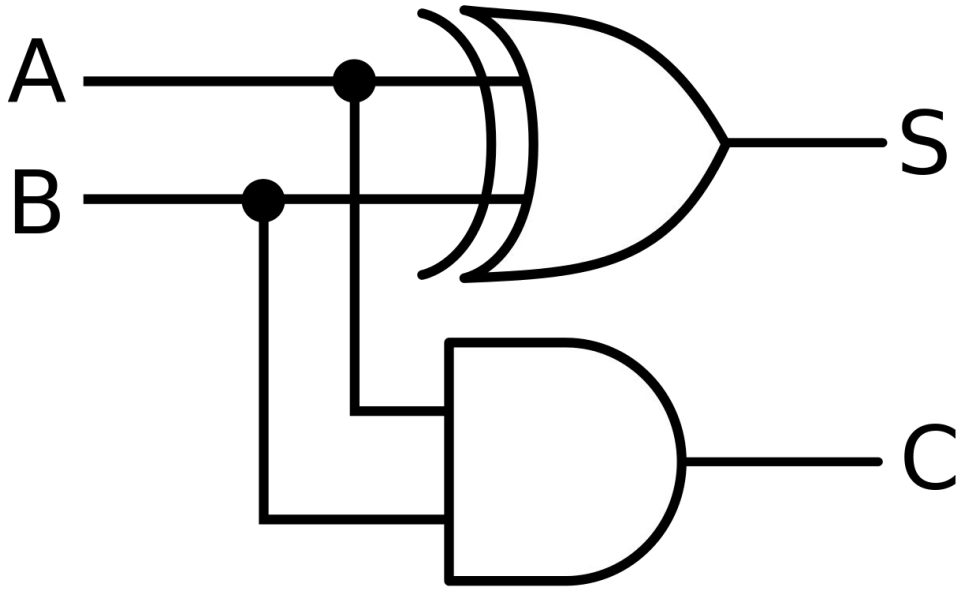
- odejmowanie
- jawnie wywołane operacje +, *
- niejawnie wywołane operacje +, *,
  np. podczas obliczania adresu na stosie, adresu komórki tablicy, czy adresu zmiennej w PIC
- arytmetyka zmiennoprzecinkowa
- algorytmy dzielenia



Przypomnienie symboli bramek logicznych



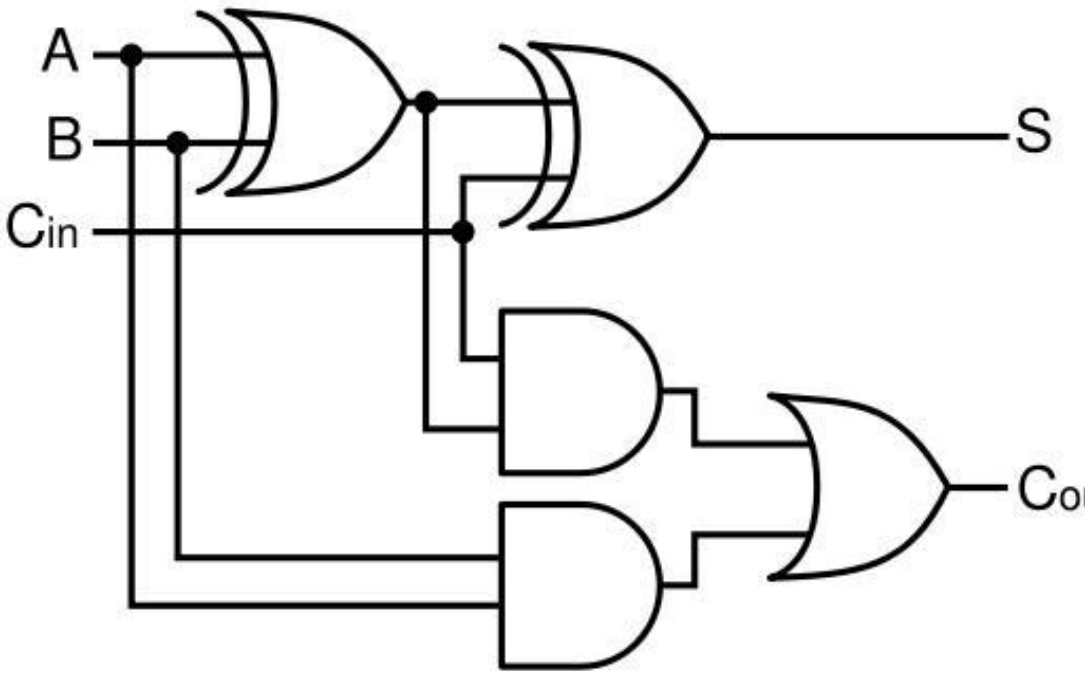
Half adder



$$S = A \oplus B$$

$$C = AB$$

Full adder



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

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

Ripple-carry adder

- dane jest 2n wejść (bitów): $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n$
- układ składa się z n full adderów
- do każdego full addera dane wejściowe dostępne są od razu, ale trzeba poczekać aż przeniesienie (ang. carry) się rozpropaguje (ang. ripple)
- na początku układ widzi wszystkie przeniesienia równe 0
- wynik zmienia się w czasie, aż po chwili osiągnie ostateczną wartość
- przeniesienie wchodzące do pierwszego full addera stale równe 0

Ripple-carry adder

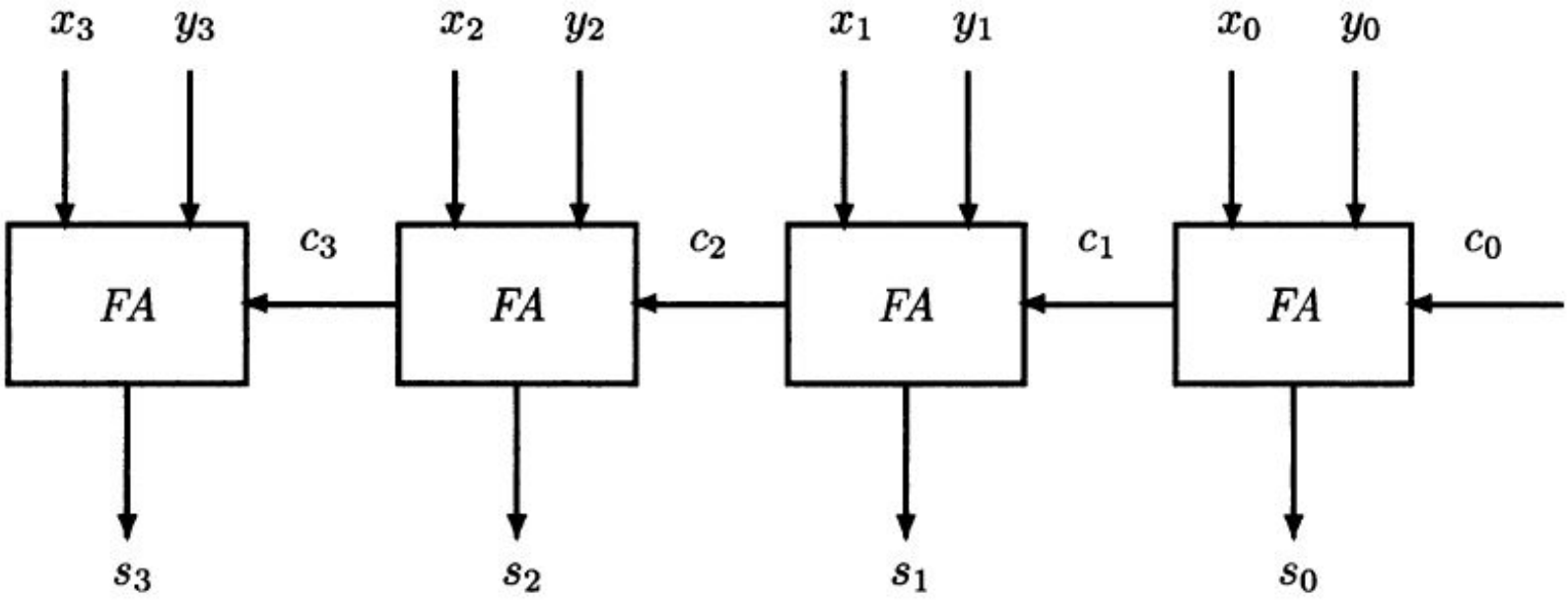


FIGURE 5.1 A 4-bit ripple-carry adder.

Ripple-carry adder

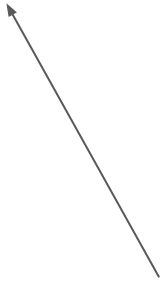
- złożoność czasowa liniowa
- możemy wykrywać, czy przeniesienie skończyło się propagować, np. w obliczaniu 00111+11000, ale nie chcemy (lub nie potrzebujemy) mieć układu, który działa zmienną ilość czasu

Odejmowanie

$$A - B = A + \bar{B} + 1$$

Odejmowanie

$$A - B = A + \bar{B} + 1$$



Odejmowanie

- ustawiamy przeniesienie wchodzące do pierwszego full addera na 1

$$A - B = A + \bar{B} + 1$$

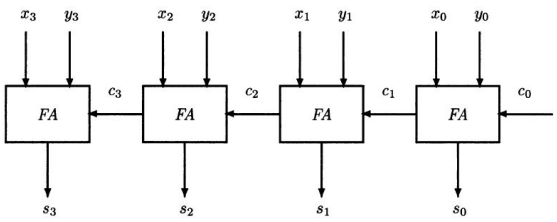


FIGURE 5.1 A 4-bit ripple-carry adder.

Carry-look-ahead adder, podejście 1

- pomysł: i-te przeniesienie jest funkcją danych wejściowych, więc można je wygenerować od razu, bez czekania na propagację
- zauważmy, że:
  - stan (1, 1) zawsze generuje przeniesienie
  - stan (1, 0) i (0, 1) generuje przeniesienie, jeśli przyszło przeniesienie
  - stan (0, 0) nigdy nie generuje przeniesienia
- niech $G_i = x_i y_i, P_i = x_i + y_i$
  oznaczają kolejno generowanie i propagowanie i-tego przeniesienia
- wartość kolejnego przeniesienia można zapisać wzorem $c_{i+1} = G_i + c_i P_i$

Carry-look-ahead adder, podejście 1

- możemy wygenerować wszystkie przeniesienia równolegle, według rozwiniętego wzoru

$$\mathrm{c}_{i+1} = G_i + G_{i-1}P_i + c_{i-1}P_{i-1}P_i =$$
$$= \mathrm{G}_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + c_{i-2}P_{i-2}P_{i-1}P_i = \cdots$$
$$= \mathrm{G}_i + G_{i-1}P_i + G_{i-2}P_{i-1}P_i + \cdots + c_0P_0P_1\cdots P_i$$

- oznaczmy opóźnienie pojedynczej bramki jako Δ_G
- obliczenie wszystkich P_i, G_i zajmuje Δ_G
- obliczenie wszystkich c_i zajmuje $2\Delta_G$
- obliczenie wszystkich s_i zajmuje $2\Delta_G$

Carry-look-ahead adder, podejście 1

- złożoność czasowa stała
- wymaga ogromnej liczby bramek logicznych
- wymaga, by bramki logiczne miały n wejść

Carry-look-ahead adder, podejście 1

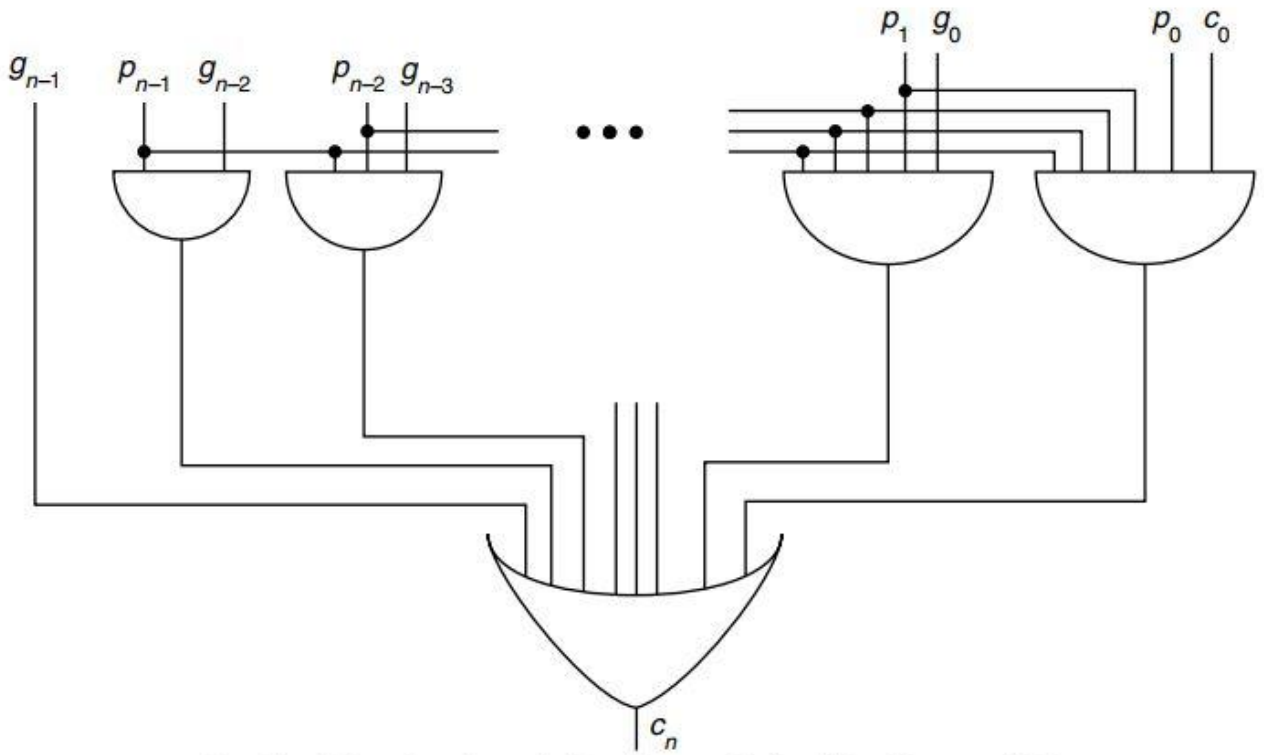


Figure J.14 Pure carry-lookahead circuit for computing the carry-out c_n of an *n*-bit adder.

Carry-look-ahead adder, podejście 2

- podzielmy wejście na bloki po k bitów
- każdy z bloków implementuje carry-look-ahead adder w czasie stałym
- przeniesienie propaguje się do kolejnych bloków
- złożoność mniej więcej k razy mniejsza niż złożoność ripple-carry adder
- zwykle k jest równe 4
  - długości słów maszynowych dzielą się przez 4
  - względy technologiczne (liczba wejść do bramek logicznych, ...)

Carry-look-ahead adder, podejście 3

- grupy po k (równym 4) bitów
- dodatkowa warstwa look-ahead na grupach, poza pierwotną warstwą na bitach
- niech G^*, P^* oznaczają kolejno przeniesienie generowane przez grupę i przeniesienie propagowane przez grupę

Carry-look-ahead adder, podejście 3

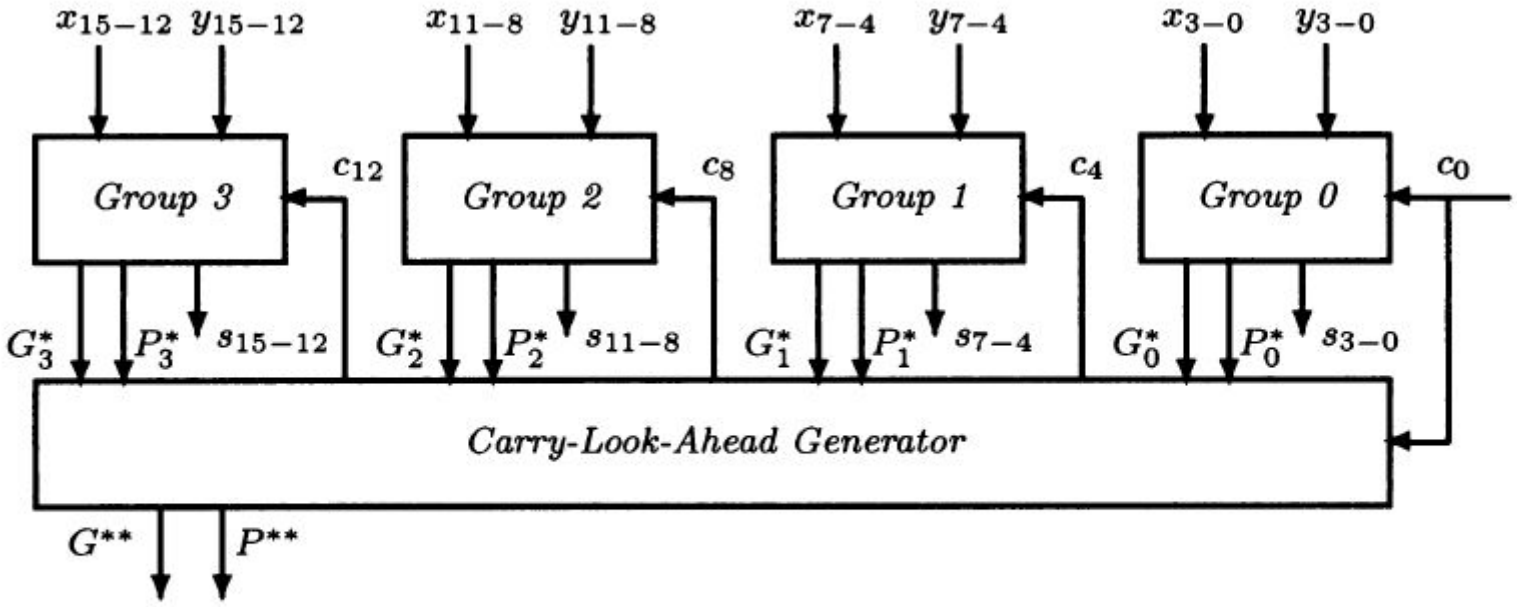


FIGURE 5.2 A 16-bit two-level carry-look-ahead adder. (The notation x_{3-0} means x_3, x_2, x_1, x_0.)

Carry-look-ahead adder, podejście 3

$$\mathrm{G}^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

$$\mathrm{P}^* = P_0P_1P_2P_3$$

$$\mathrm{c}_4 = G_0^* + c_0P_0^*$$

$$\mathrm{c}_8 = G_1^* + G_0^*P_1^* + c_0P_0^*P_1^*$$

$$\mathrm{c}_{12} = G_2^* + G_1^*P_2^* + G_0^*P_1^*P_2^* + c_0P_0^*P_1^*P_2^*$$

Carry-look-ahead adder, podejście 3

- fazy:
  - obliczanie przeniesień generowanych i propagowanych przez bity
  - obliczanie przeniesień generowanych i propagowanych przez grupy
  - obliczanie przeniesień wchodzących do grup
  - równoległe obliczanie wyniku przez wszystkie grupy
- złożoność całego układu to $9\Delta_G$
- zauważmy, że z układu wychodzą G^{**}, P^{**}
  tzn. przeniesienia generowane i propagowane przez sekcję
- z czterech takich układów (i dodatkowego carry-look-ahead generator) można stworzyć trzywarstwowy sumator 64-bitowy
- złożoność proporcjonalna do $log_4 n$

Conditional sum adder

- dzielimy bity na grupy
- dla każdej grupy generujemy równolegle dwa wyniki, w zależności od przychodzącego bitu przeniesienia
- gdy przychodzący bit przeniesienia będzie znany, wybieramy właściwy wynik

Conditional sum adder

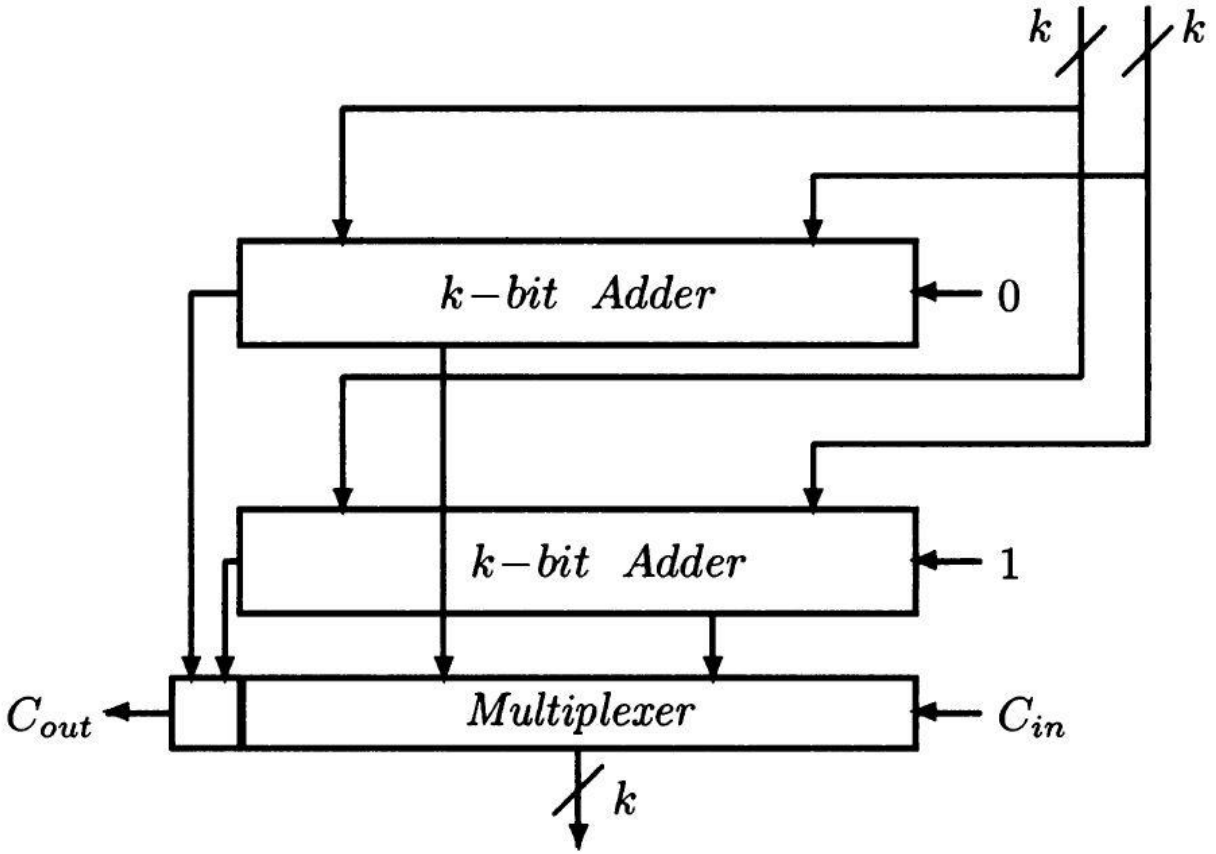


FIGURE 5.3 Selecting the correct set of sum bits and carry-out.

Conditional sum adder

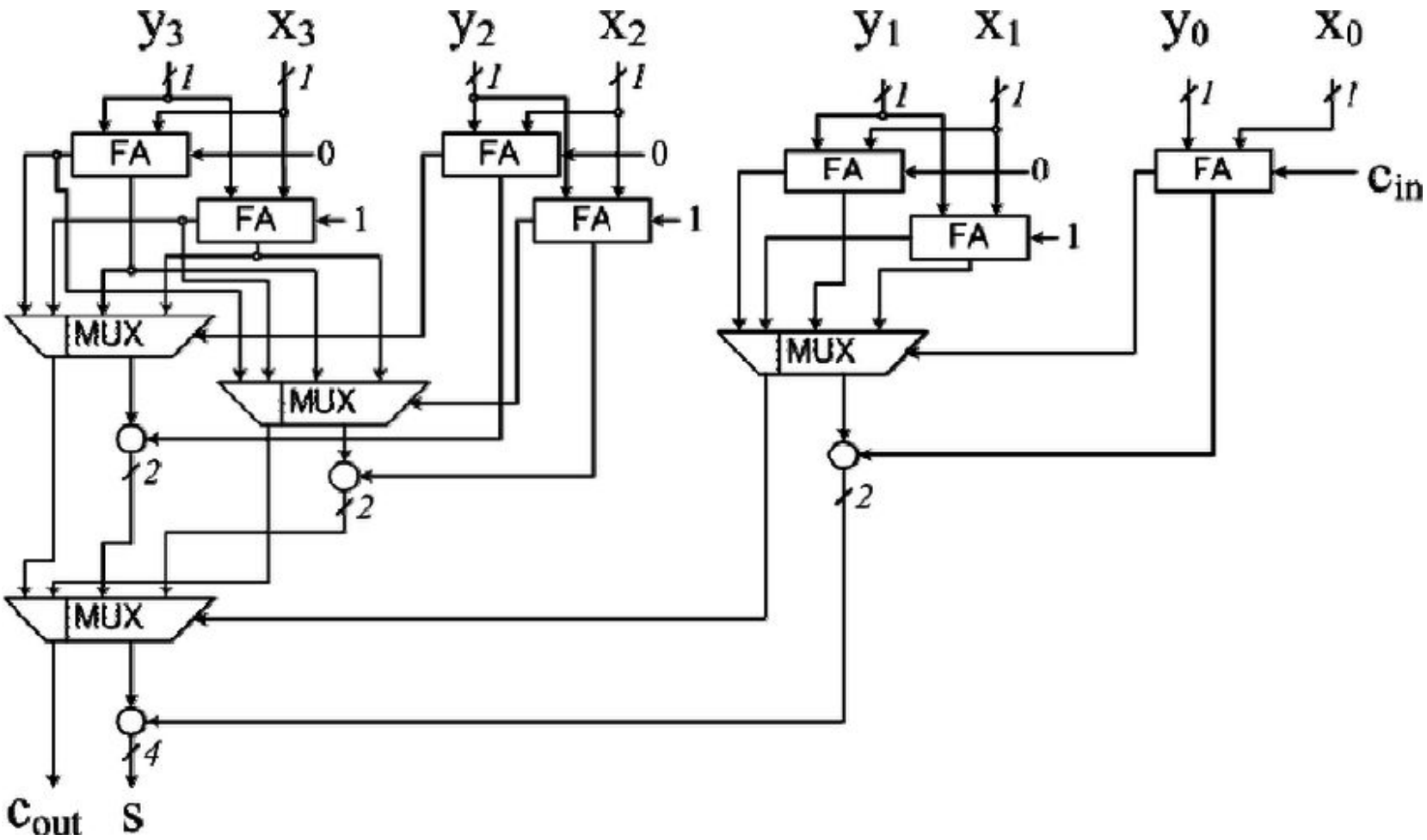
- podzielmy n bitów na dwie grupy po n/2 bitów
- w każdej z tych grup zastosujmy rekurencyjnie ten sam pomysł (podzielimy je na dwie grupy o rozmiarach n/4)
- ...
- profit
- złożoność czasowa logarytmiczna

Conditional sum adder

	i	7	6	5	4	3	2	1	0
	x_i	1	0	1	1	0	1	1	0
	y_i	0	0	1	0	1	1	0	1
Step 1	s_i^0	1	0	0	1	1	0	1	1
	c_{i+1}^0	0	0	1	0	0	1	0	0
	s_i^1	0	1	1	0	0	1	0	
	c_{i+1}^1	1	0	1	1	1	1	1	
Step 2	s_i^0	1	0	0	1	0	0	1	1
	c_{i+1}^0	0		1		1		0	
	s_i^1	1	1	1	0	0	1		
	c_{i+1}^1	0		1		1			
Step 3	s_i^0	1	1	0	1	0	0	1	1
	c_{i+1}^0	0				1			
	s_i^1	1	1	1	0				
	c_{i+1}^1	0							
Result		1	1	1	0	0	0	1	1

FIGURE 5.4 Conditional sum addition of two 8-bit numbers.

Conditional sum adder



Carry-look-ahead vs conditional sum adder

- podobna złożoność obliczeniowa (logarytmiczna)
- CLAA bardziej popularny niż CSA
  - budowa modularna
  - mniejszy fan-out elementów układu

W poszukiwaniu optymalnego sumatora

- czy można dodawać jeszcze szybciej?
- alternatywne systemy liczbowe
  - residue number system (brak przeniesień)
  - signed-digit number system (jedna cyfra wyniku zależy tylko od czterech cyfr z wejścia)
  - konwersje pomiędzy systemami zbyt kosztowne
- czy istnieje dolne ograniczenie na czas działania sumatora?
- wiele różnych czynników
  - technologia wykonania sumatora
  - łatwość projektowania
  - fizyczny rozmiar układu
  - koszt wykonania układu
  - prędkość obliczania sumy

Dolne ograniczenie na czas działania

- niektóre cyfry wyniku zależą od wszystkich cyfr wejścia
- zakładamy idealny model
  - jeden typ bramki (f, r), gdzie r (radix) to podstawa systemu liczbowego, a f to fan-in
  - bramka w jednym cyklu potrafi obliczyć dowolną funkcję f-arną
  - nie bierzemy pod uwagę fan-out
- można pokazać, że $T_{add} \geq \lceil log_f 2n \rceil$

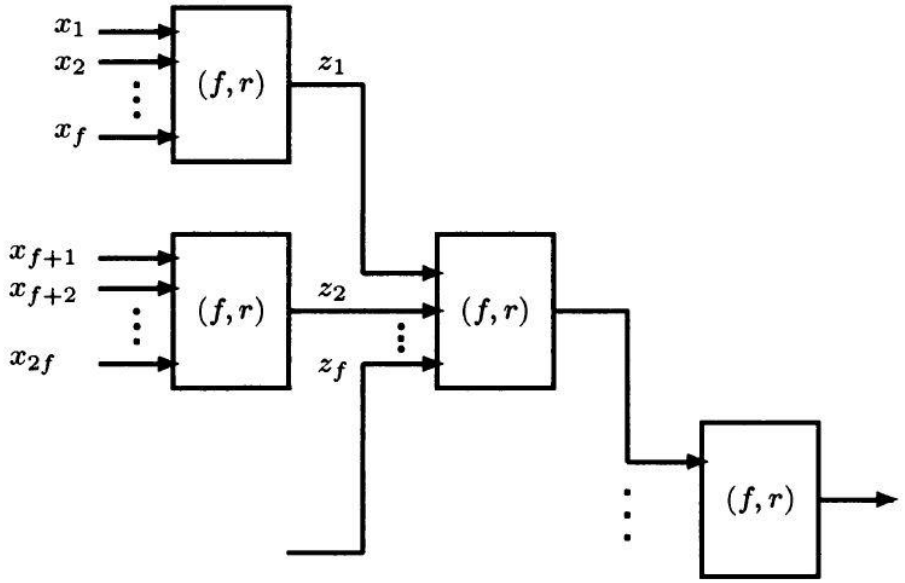


FIGURE 5.5 A partial diagram of a circuit implemented with (*f*, *r*) gates.

Generowanie i propagowanie c.d.

- przeniesienie propaguje się przez grupę, jeżeli po dojściu do najmniej znaczącej pozycji jest w stanie przejść i wyjść za najbardziej znaczącą pozycję
- przeniesienie jest generowane przez grupę, jeżeli jest generowane na jednej z pozycji grupy, a następnie propagowane za najbardziej znaczący bit

$$P_{i:j} = \begin{cases} P_i & \text{gdy } i = j \\ P_i P_{i-1:j} & \text{gdy } i > j \end{cases} \qquad G_{i:j} = \begin{cases} G_i & \text{gdy } i = j \\ G_i + P_i G_{i-1:j} & \text{gdy } i > j \end{cases}$$

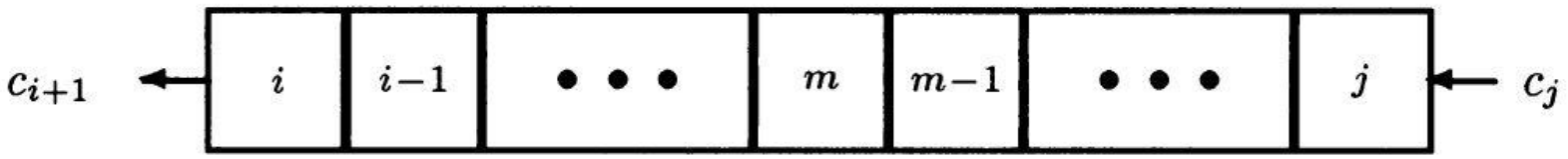


FIGURE 5.7 A group consisting of i - j + 1 bit positions ($i \geq j$).

Fundamental carry operator

- zdefiniujmy łączny operator $(P, G) \circ (\tilde{P}, \tilde{G}) = (P\tilde{P}, G + P\tilde{G})$
- wtedy $(P_{i:j}, G_{i:j}) = (P_{i:m}, G_{i:m}) \circ (P_{m-1:j}, G_{m-1:j}), \text{dla } i \geq m \geq j + 1$
- oraz $(P, G) \circ (P, G) = (P, G)$
- stąd $(P_{i:j}, G_{i:j}) = (P_{i:m}, G_{i:m}) \circ (P_{v:j}, G_{v:j}), \text{dla } v \geq m - 1$
- zatem grupy mogą się nachodzić!
- trochę jak programowanie dynamiczne

Fundamental carry operator

- jak dla grupy obliczyć $c_{i+1}, c_i, \cdots, c_{j+1}; s_i, s_{i-1}, \cdots, s_j$
- wykorzystujemy przychodzące przeniesienie c_j
- $\mathrm{c}_m = G_{m-1:j} + P_{m-1:j} c_j = (P_{m-1:j}, G_{m-1:j}) \circ (1, c_j)$

  $\mathrm{s}_m = c_m \oplus x_m \oplus y_m$

Fundamental carry operator

- 5-bitowy ripple-carry adder

$$(P_4, G_4) \circ \{(P_3, G_3) \circ ((P_2, G_2) \circ [(P_1, G_1) \circ \{(P_0, G_0) \circ (1, c_0)\}])\}$$

- 16-bitowy carry-look-ahead adder z czterema grupami wielkości 4, z ripple-carry pomiędzy nimi

$$(P_{15:12}, G_{15:12}) \circ \{(P_{11:8}, G_{11:8}) \circ [(P_{7:4}, G_{7:4}) \circ \{(P_{3:0}, G_{3:0}) \circ (1, c_0)\}]\}$$

Fundamental carry operator

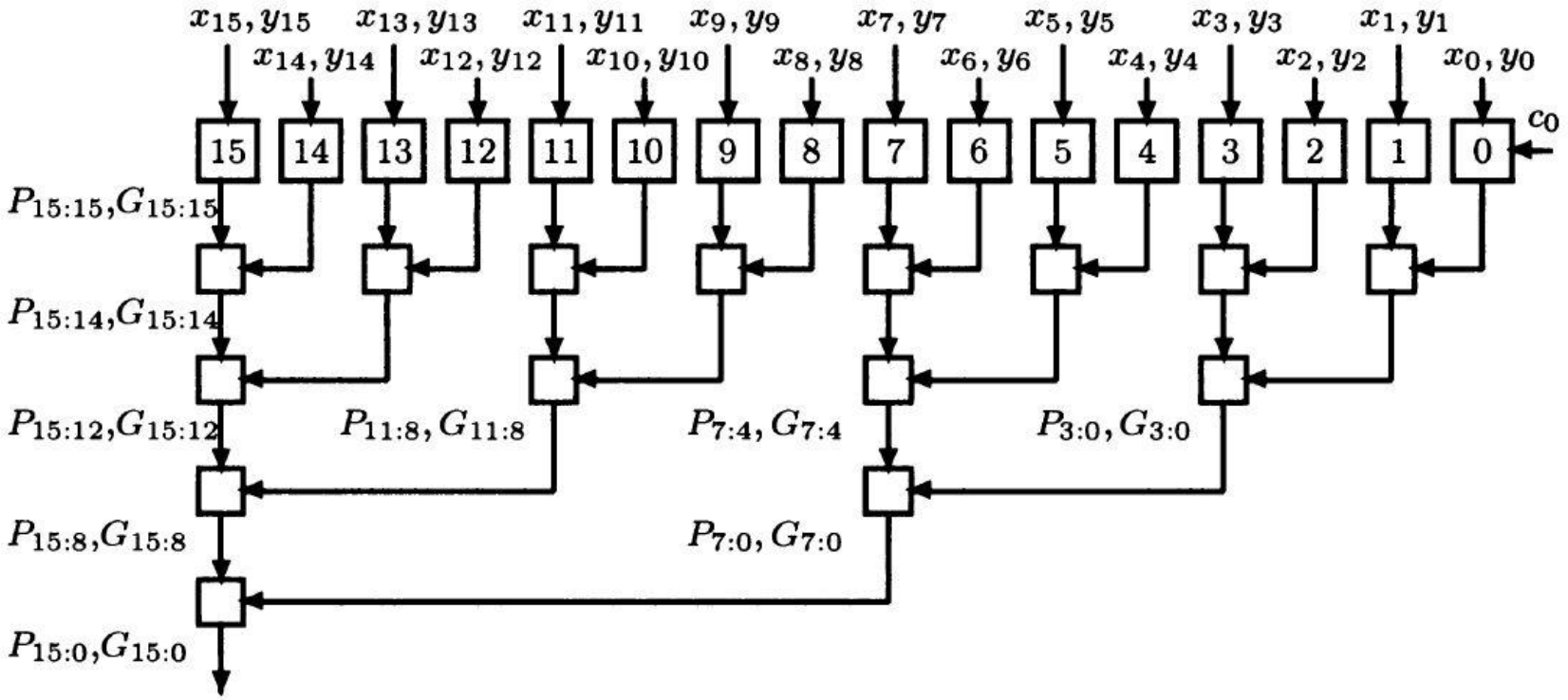


FIGURE 5.8 A tree structure for calculating c_{16} (each line, except c_0, represents two signals that are either x_m and y_m or $P_{v:m}$ and $G_{v:m}$).

Brent-Kung parallel prefix adder

- parallel prefix circuit to układ kombinacyjny, który dla wejścia $x_1, x_2, \cdots, x_n$ produkuje $x_1, x_2 \circ x_1, \cdots, x_n \circ x_{n-1} \circ \cdots \circ x_1$

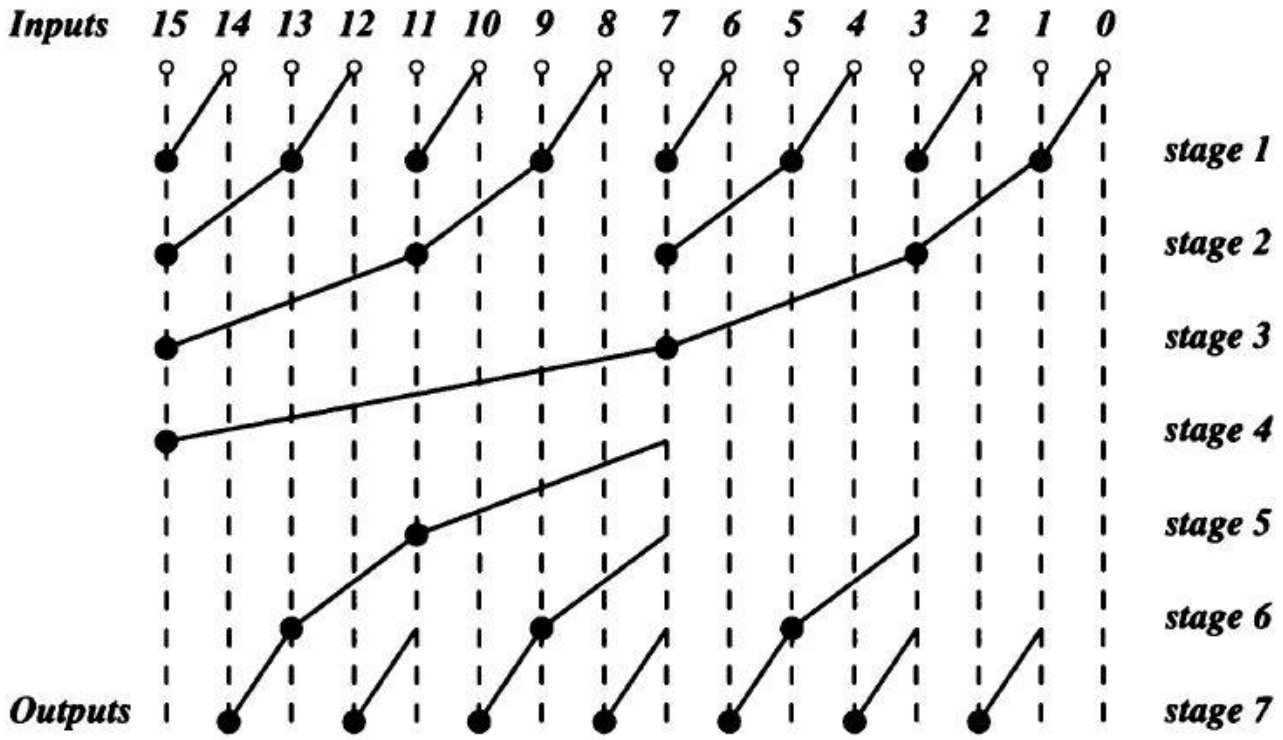


FIGURE 5.9 The Brent-Kung (2) parallel prefix graph.

Ladner-Fischer parallel prefix adder

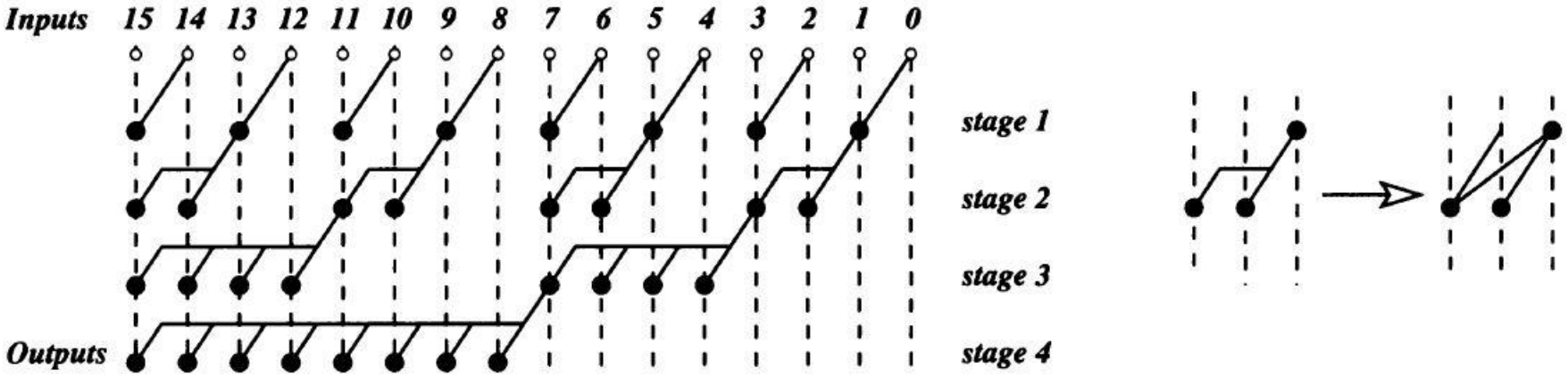


FIGURE 5.10 The Ladner-Fischer (17) parallel prefix graph.

Kogge-Stone parallel prefix adder

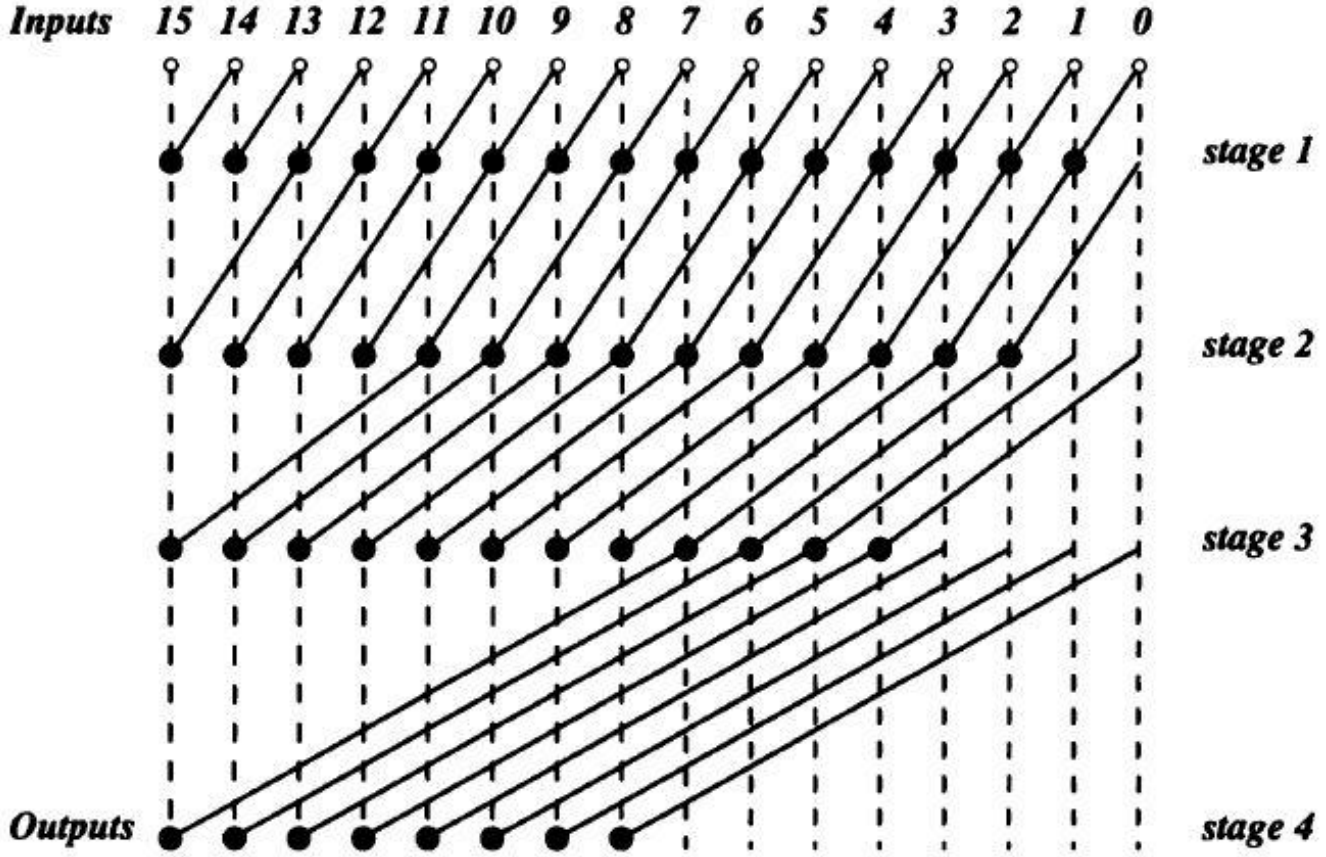
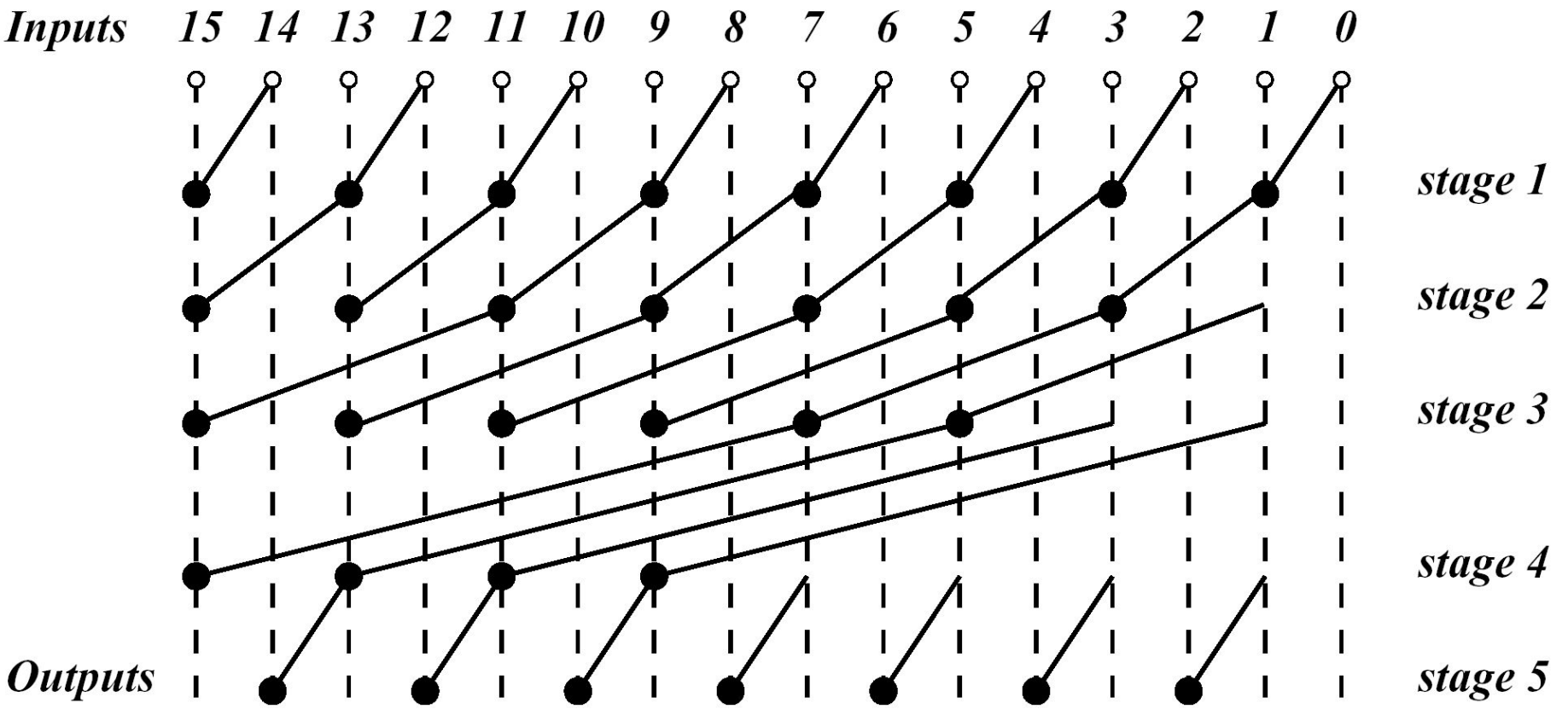


FIGURE 5.11 The Kogge-Stone (16) parallel prefix graph.

Han-Carlson parallel prefix adder



Ling adder

- wariacja carry-look-ahead adder
- zauważmy, że $c_4 = G_{3:0} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$ ma powtarzający się czynnik P_3
- zdefiniujmy $H_{3:0} = G_3 + G_2 + P_2G_1 + P_2P_1G_0$ którego możemy używać zamiast $G_{3:0}$
- trochę bardziej skomplikowane obliczanie wyniku
- mniejszy fan-in elementów skutkuje szybszym układem

Carry-select adder

- podobny pomysł do conditional sum adder, ale bez podejścia rekurencyjnego
- dzielimy bity na grupy (być może o różnych długościach)

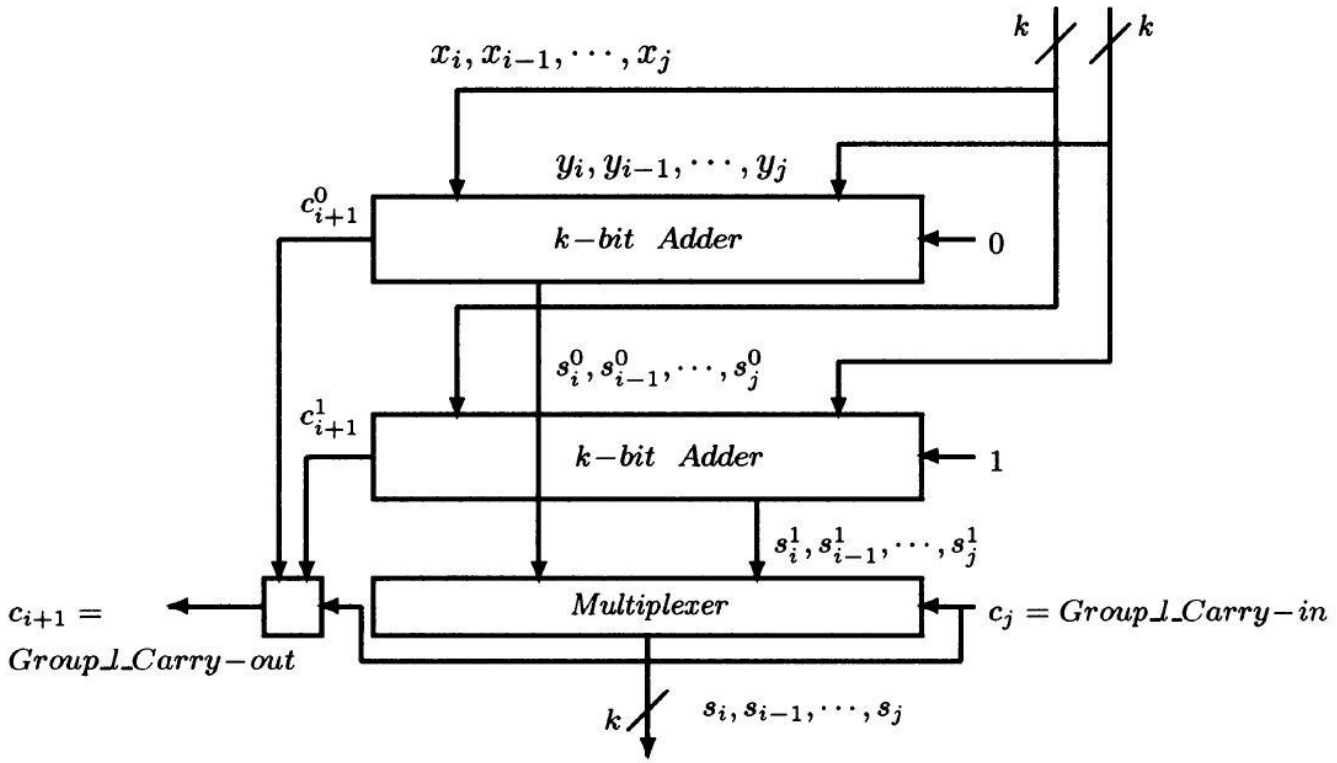


FIGURE 5.13 The *I* th group, consisting of the *k* bit positions *j*, *j* + 1, ···, *I*, in a carry-select adder.

Carry-select adder

- niech grupy mają kolejno długość 1, 2, 3, 4, …
- złożoność proporcjonalna do $\sqrt{n}$

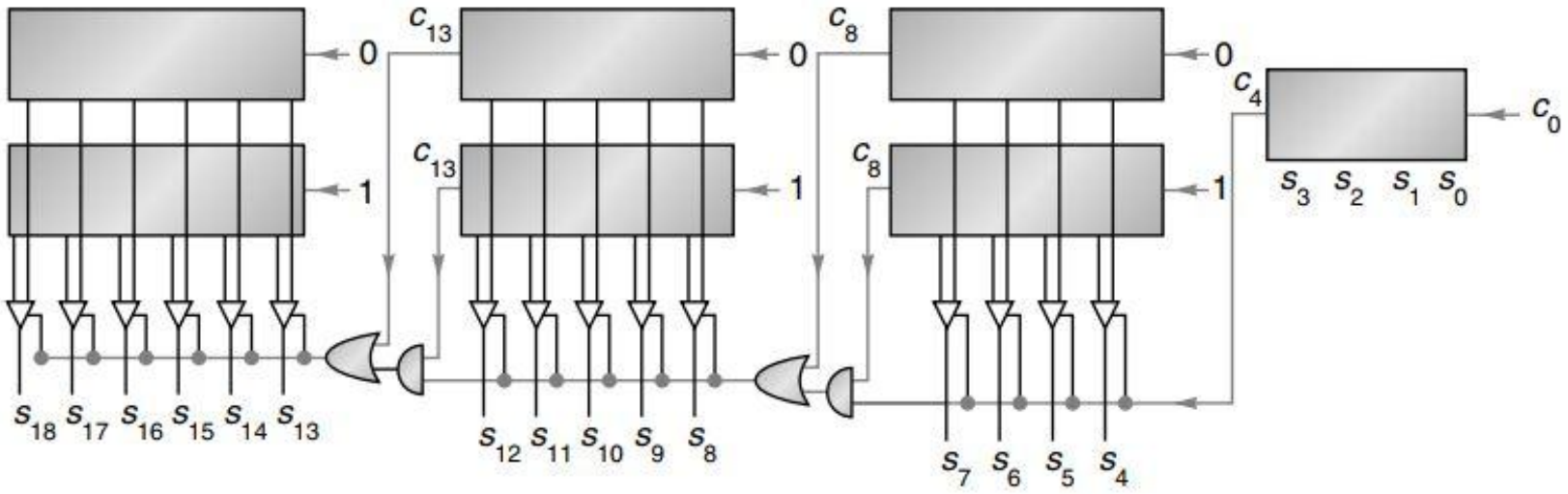


Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Carry-skip adder

- dzielimy bity na k (równe $\sqrt{n}$) grup
- początkowo dla każdej grupy zakładamy, że dostanie przeniesienie równe 0
- dla każdej grupy obliczamy P (propagację grupy)
- jeżeli grupa propaguje, i dostanie przeniesienie równe 1, od razu ustawiamy przeniesienie wychodzące na 1 (zatem kolejna grupa może równolegle zacząć się poprawnie wykonywać)
- jeżeli nie ustawiliśmy przeniesienia wychodzącego na 1, to na pewno będzie ono równe 0, czyli kolejna grupa i tak już równolegle liczy swój wynik
- złożoność proporcjonalna do $\sqrt{n}$

Carry-skip adder

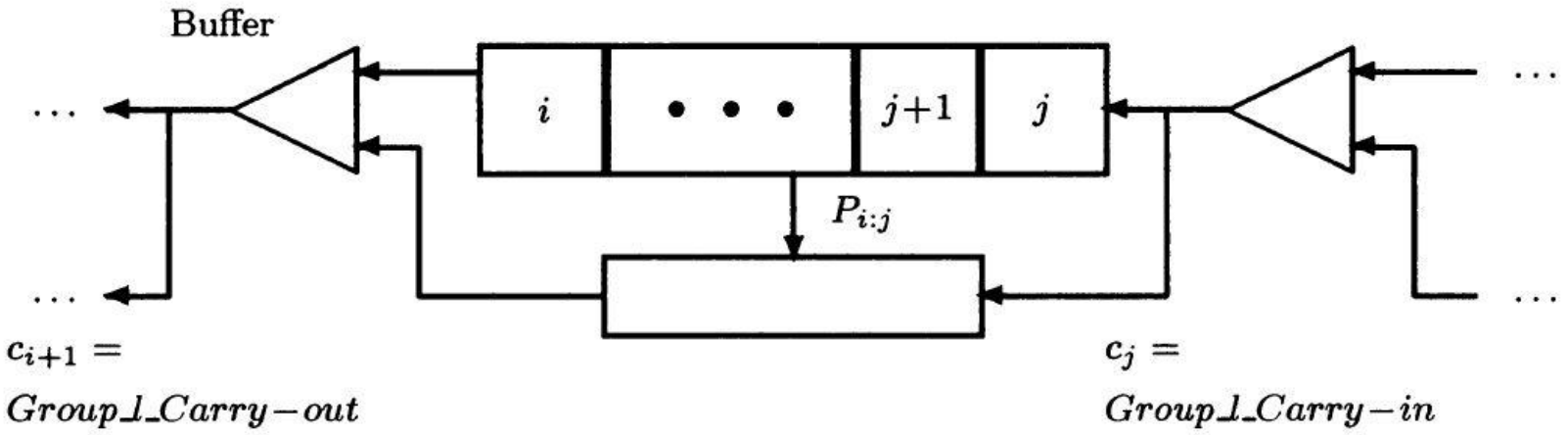


FIGURE 5.14 The *l* th group consisting of bit positions $j, j+1, \cdots, l$ in a carry-skip adder.

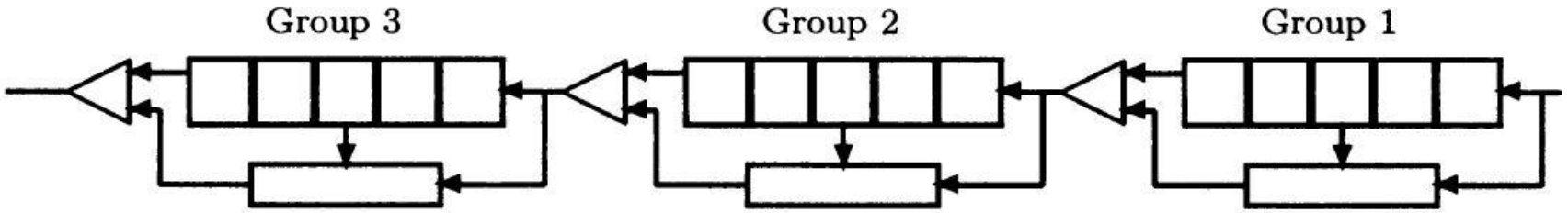


FIGURE 5.15 A 15-bit carry-skip adder.

Porównanie kilku algorytmów

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

- oczywiście istnieją rozwiązania hybrydowe!

Carry-save adder

- co jeżeli chcemy dodać więcej niż dwa operandy?
- niech propagowanie przeniesień odbywa się raz, w ostatniej fazie algorytmu
- zmieniamy myślenie, teraz pojedynczy układ przyjmuje trzy 1-bitowe liczby i zamiast sumy i przeniesienia generuje 2-bitową sumę

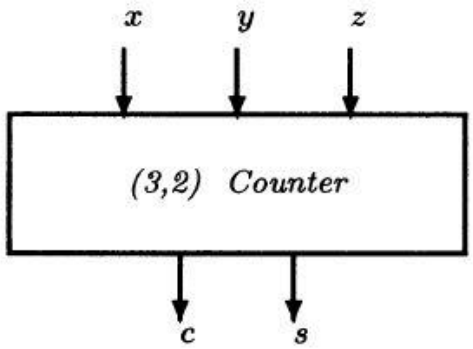


FIGURE 5.22 A (3,2) counter.

$$s = (x + y + z) \bmod 2 \qquad \text{and} \qquad c = \frac{(x + y + z) - s}{2}$$

Carry-save adder

- aby dodać k liczb, potrzebujemy (k-2) jednostek CSA oraz jedną jednostkę CPA (carry-propagating adder), która rozpropaguje przeniesienia

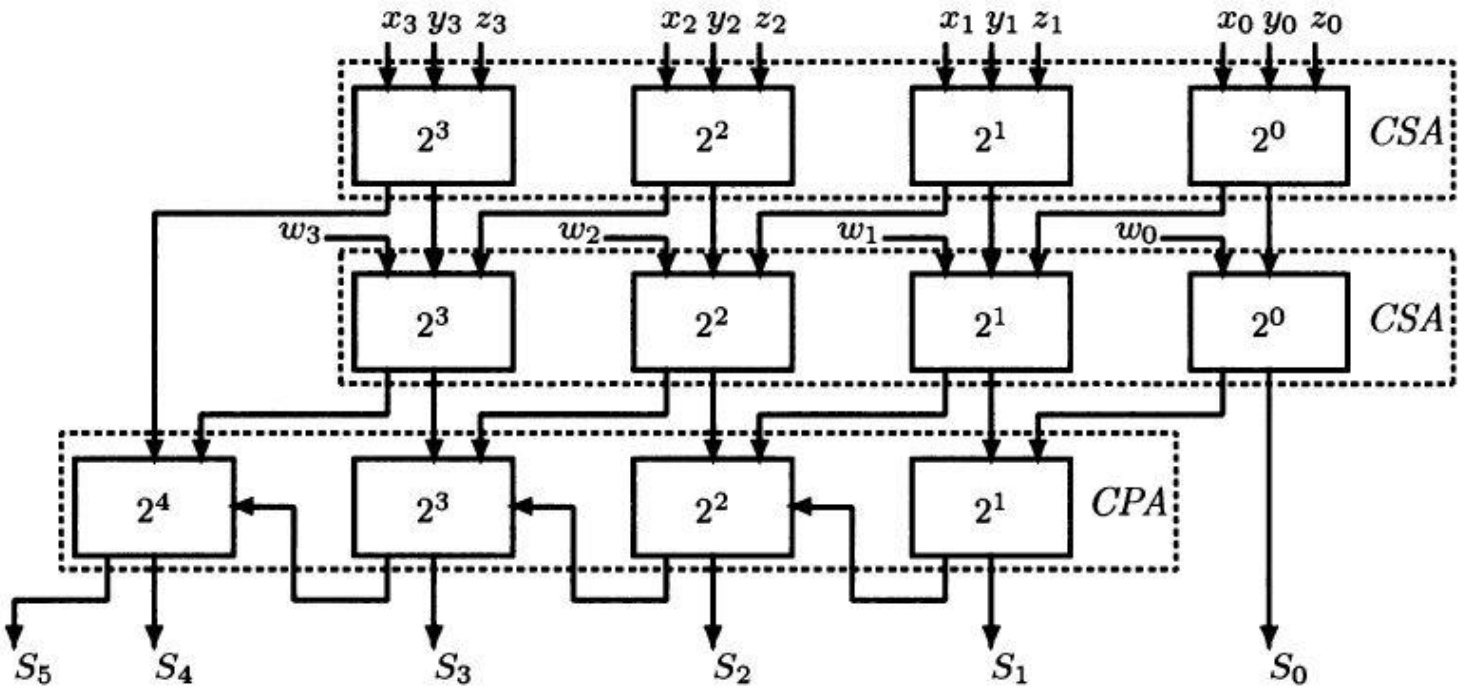


FIGURE 5.23 A carry-save adder for four operands.

Carry-save adder w mnożeniu

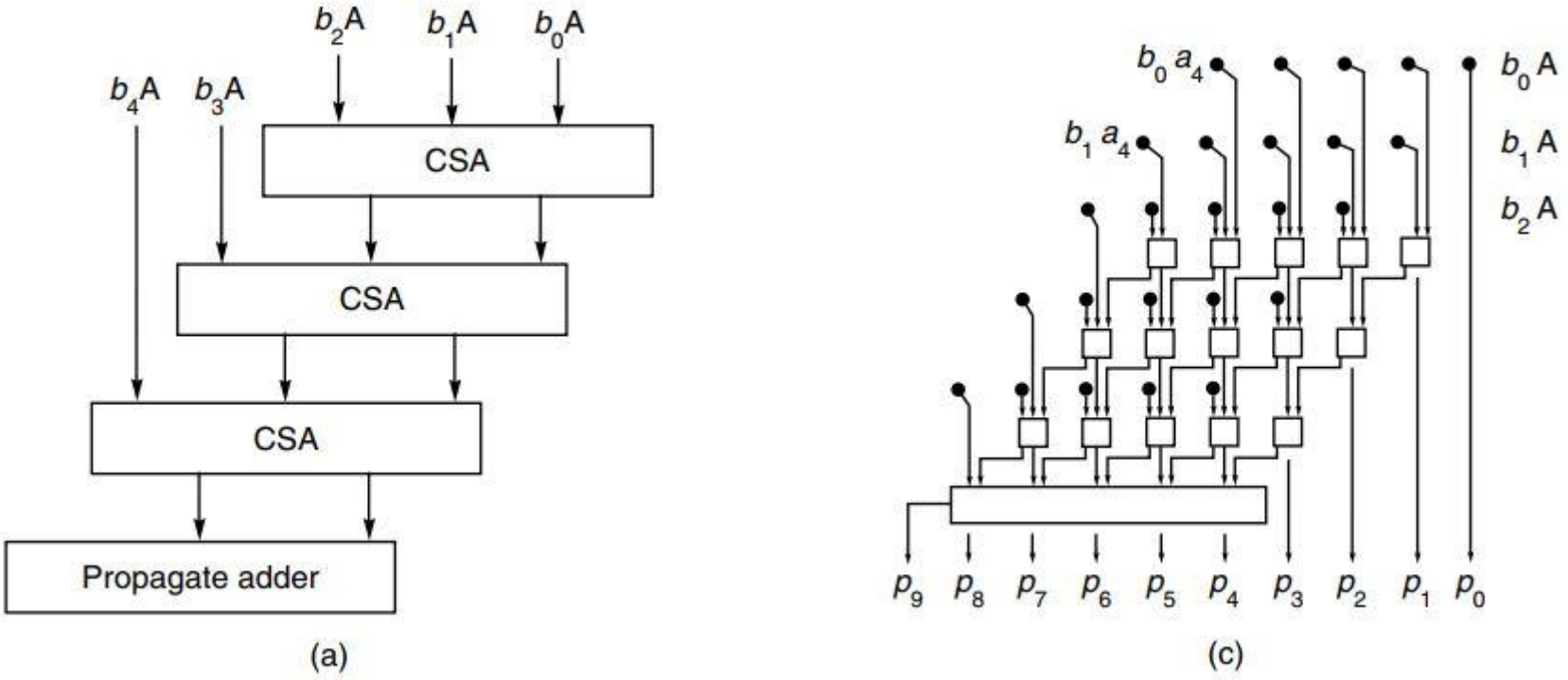


Figure J.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

Wallace tree

- bardziej efektywny sposób układania CSA
- dla k operandów potrzeba około $\frac{\log(k/2)}{\log(3/2)}$ poziomów drzewa
- nie możemy stworzyć normalnego drzewa binarnego, bo nie mamy liczników (2, 1)

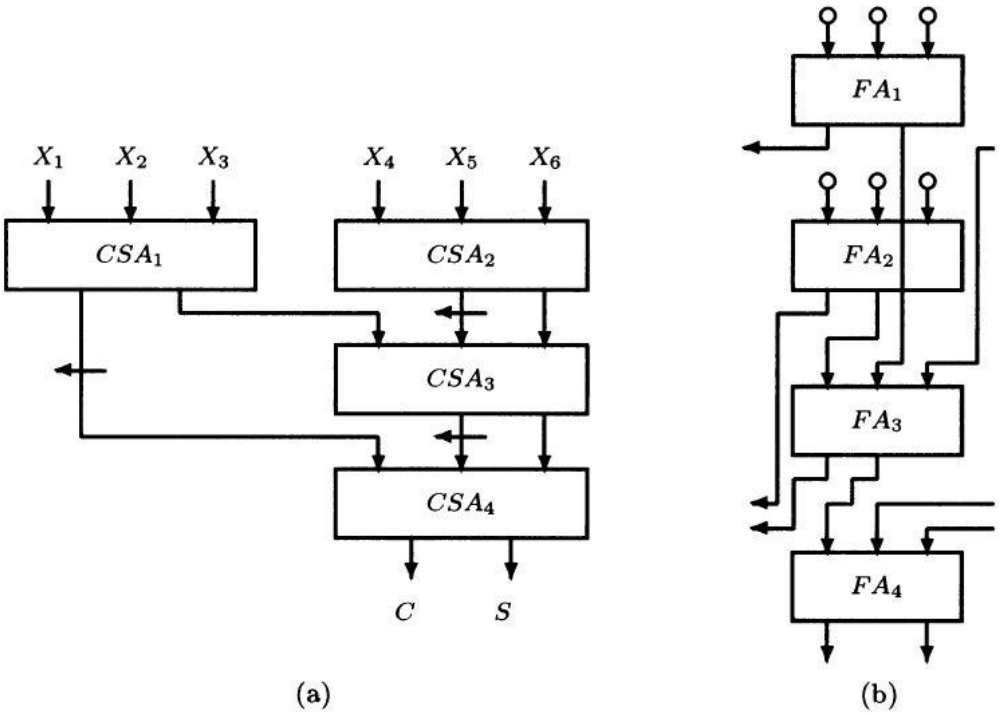


FIGURE 5.24 (a) A CSA tree for six operands. (b) An implementation of a 6-input bit-slice of the tree in (a).

Wallace tree

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

TABLE 5.1 The number of levels in a CSA tree for k operands.

Wallace tree w mnożeniu

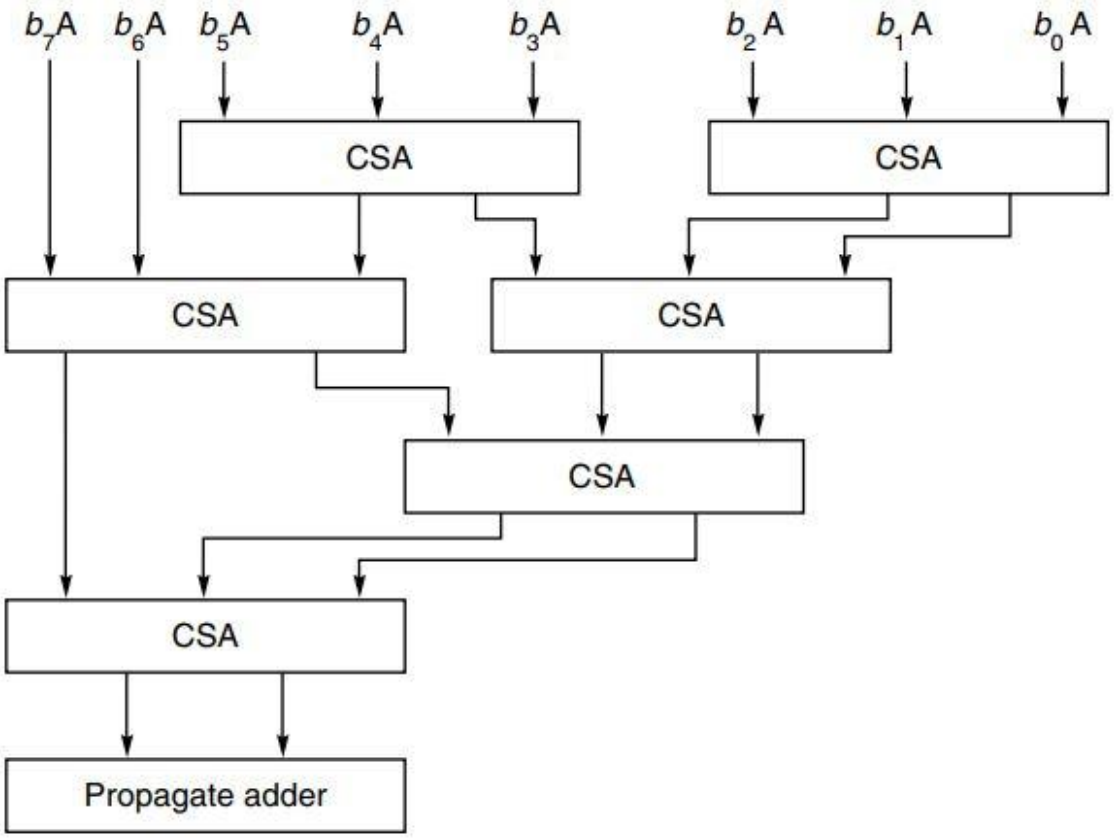
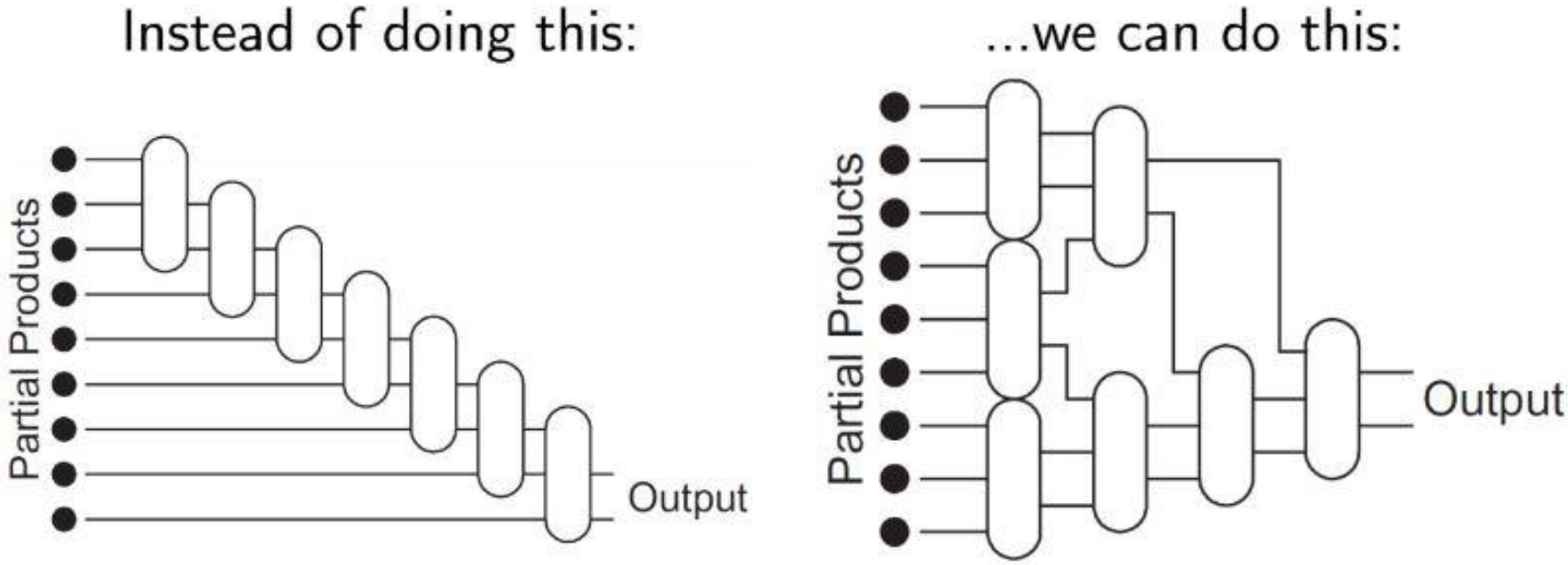


Figure J.30 Wallace tree multiplier. An example of a multiply tree that computes a product in 0(log *n*) steps.

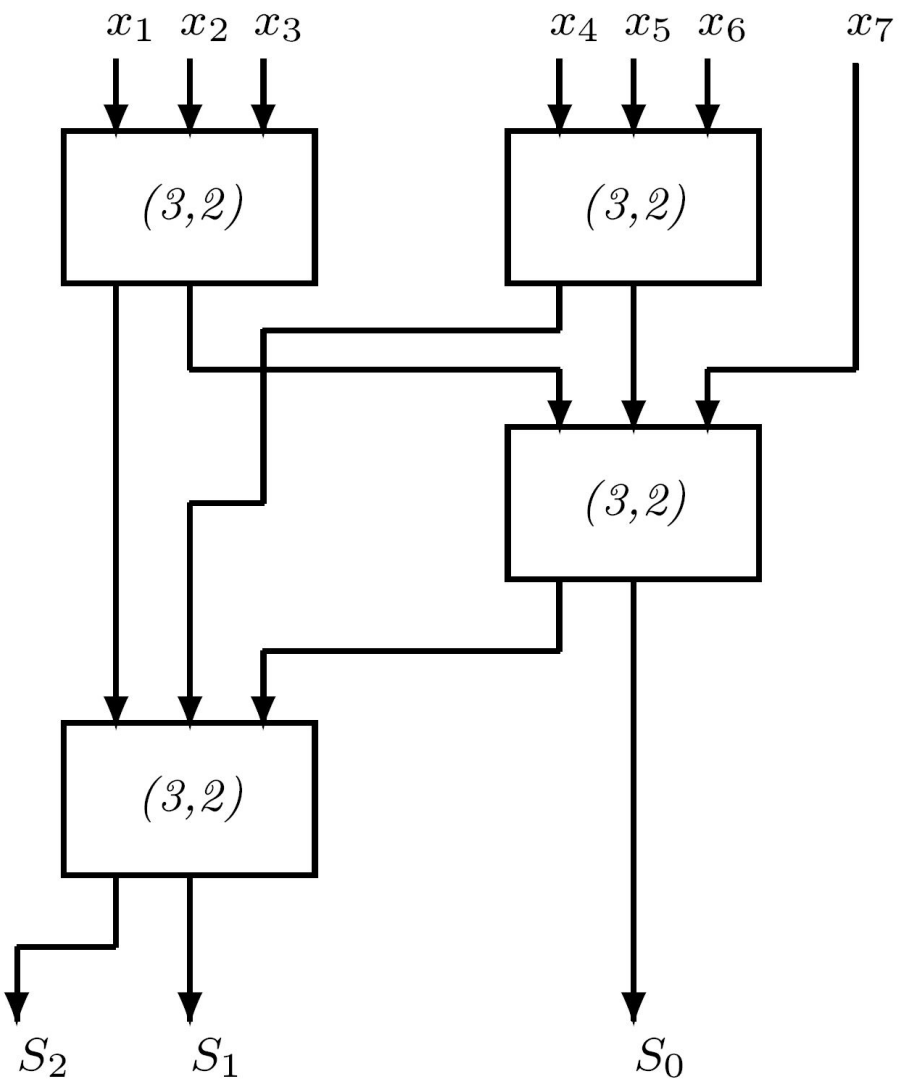
CSA array vs Wallace tree (mnożenie)



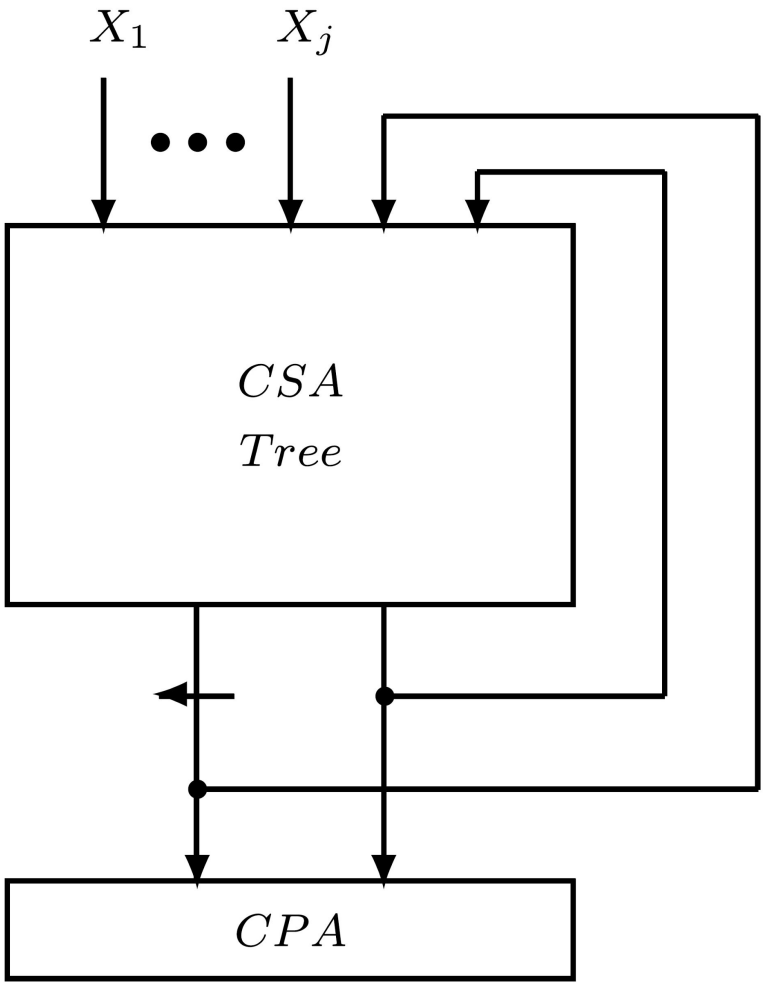
Kolejne usprawnienia w CSA

- użycie licznika (7, 3) zamiast (3, 2)
  - usuwa 4 bity, a nie zajmuje aż tak bardzo więcej miejsca
- można użyć pamięci ROM jako implementację liczników (m, n) dla większych m i n
- czasem używa się różnych liczników w różnych warstwach, np. (7, 3) w pierwszej, a (3, 2) w drugiej

Kolejne usprawnienia w CSA



Kolejne usprawnienia w CSA



Kolejne usprawnienia w CSA

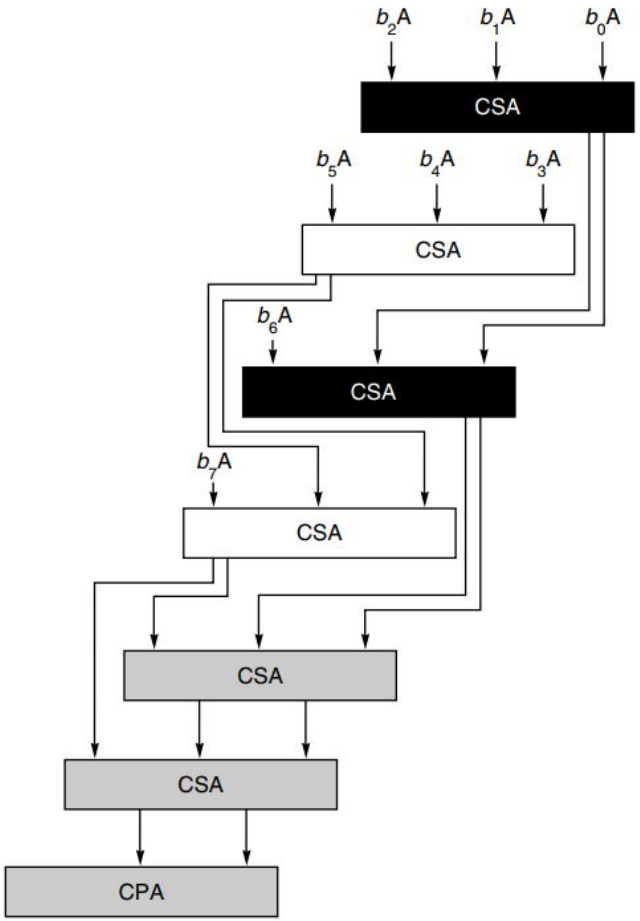


Figure J.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

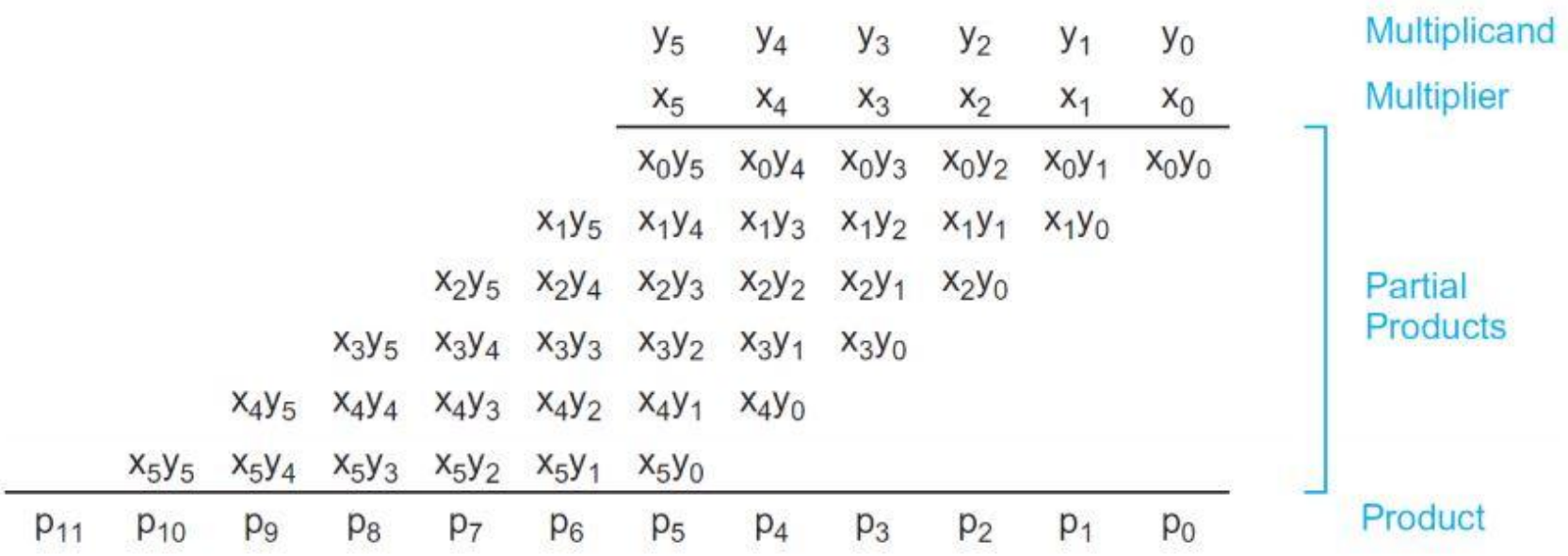
Potokowanie operacji dodawania

- przydatne gdy wykonujemy wiele dodawań z rzędu
- możliwe przy niektórych algorytmach, np. conditional sum adder
- niemożliwe (bez modyfikacji pomysłu/implementacji) przy innych, np. carry-look-ahead adder
- zalecane przy dodawaniu wielu operandów, np. carry-save adder (każdy poziom drzewa osobną fazą potoku)

Mnożenie

- trzy rodzaje układów mnożących
  - równoległy - równolegle generuje iloczyny cząstkowe i dodaje je używając sumatora przyjmującego wiele operandów
  - sekwencyjny - generuje iloczyny cząstkowe sekwencyjnie i dodaje je do dotychczasowej sumy
  - tablicowy - zawiera tablicę komórek, która równolegle generuje i dodaje nowe iloczyny cząstkowe (najszybsze, ale najbardziej skomplikowane w implementacji)
- sposoby na przyspieszenie
  - zmniejszenie liczby iloczynów cząstkowych
  - przyspieszenie fazy dodawania

Mnożenie


y_5 y_4 y_3 y_2 y_1 y_0
x_5 x_4 x_3 x_2 x_1 x_0
x_0y_5 x_0y_4 x_0y_3 x_0y_2 x_0y_1 x_0y_0
x_1y_5 x_1y_4 x_1y_3 x_1y_2 x_1y_1 x_1y_0
x_2y_5 x_2y_4 x_2y_3 x_2y_2 x_2y_1 x_2y_0
x_3y_5 x_3y_4 x_3y_3 x_3y_2 x_3y_1 x_3y_0
x_4y_5 x_4y_4 x_4y_3 x_4y_2 x_4y_1 x_4y_0
x_5y_5 x_5y_4 x_5y_3 x_5y_2 x_5y_1 x_5y_0
p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0
Multiplicand
Multiplier
Partial Products
Product

Zmniejszenie liczby iloczynów cząstkowych

- licząc AB, moglibyśmy na początku wygenerować A, 2A, 3A, a następnie analizować po 2 bity B
- zmniejsza liczbę iloczynów cząstkowych o połowę
- zwiększa stopień skomplikowania fazy dodawania
- …
- brak profitu

Booth’s algorithm

- przy mnożeniu A*B:
  - jeżeli B ma k zer z rzędu, wystarczy tylko k razy przesunąć bitowo akumulator w prawo
  - jeżeli B ma k jedynek z rzędu, możemy zamiast k dodawań zrobić jedno dodawanie i jedno odejmowanie

$$\cdots 0\{11\cdots 11\}0\cdots = \cdots 1\{00\cdots 00\}0\cdots - \cdots 0\{00\cdots 01\}0\cdots$$

Using *SD* (signed-digit) notation, discussed in Chapter 2, the above can be written as

$$\cdots 1\{00\cdots 0\bar{1}\}0\cdots$$

For example, $\cdots 0\{1111\}0\cdots = \cdots 1\{0000\}0\cdots - \cdots 0\{0001\}0\cdots$

Booth's algorithm

- tak naprawdę wzór na nowy mnożnik to $y_i = x_{i-1} - x_i$, gdzie $x_{-1} = 0$

x_i	x_{i-1}	Operation	Comments	y_i
0	0	shift only	string of zeros	0
1	1	shift only	string of ones	0
1	0	subtract and shift	beginning of a string of ones	$\bar{1}$
0	1	add and shift	end of a string of ones	1

TABLE 6.1 Booth's algorithm.

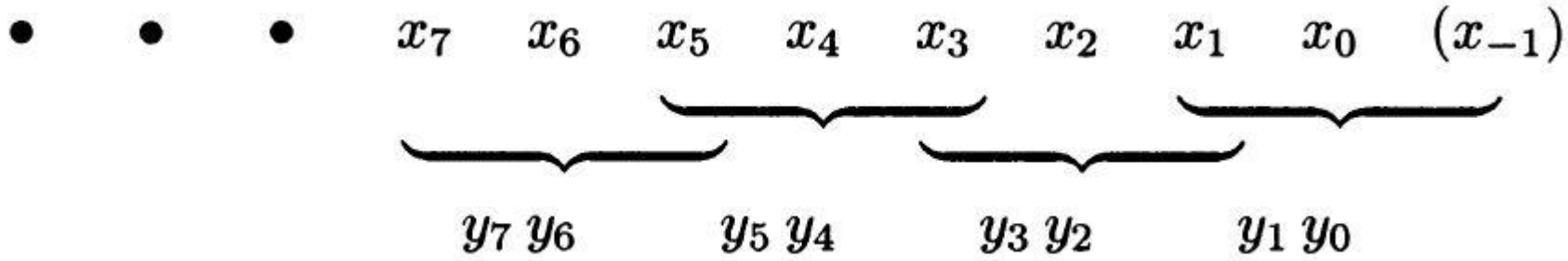
- wady:
  - liczba dodawań, odejmowań, oraz przesunięć bitowych pomiędzy dodawaniami lub odejmowaniami jest zmienna (czynnik niepożądany przy projektowaniu układów)
  - $1010101 = 1\bar{1}1\bar{1}1\bar{1}1\bar{1}$

Booth's algorithm

A		1	0	1	1			-5
X	$\times$	1	1	0	1			-3
Y		0	$\bar{1}$	1	$\bar{1}$			recoded multiplier
Add $-A$		0	1	0	1			
Shift		0	0	1	0	1		
Add A	$+$	1	0	1	1			
		1	1	0	1	1		
Shift		1	1	1	0	1	1	
Add $-A$	$+$	0	1	0	1			
		0	0	1	1	1	1	
Shift		0	0	0	1	1	1	1

Radix-4 modified Booth’s algorithm

- zamieniamy bity grupami po 3 (w tym 2 ulegają zmianie, a 1 to bit odniesienia)
- bardziej efektywne obsłużenie odizolowanych 1 i 0



Radix-4 modified Booth's algorithm

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

TABLE 6.3 A radix-4 modified Booth's algorithm.

Radix-4 modified Booth's algorithm

A			01	00	01			17
X	$\times$		11	01	11			-9
Y			$0\bar{1}$	10	$0\bar{1}$			recoded multiplier
			$-A$	$+2A$	$-A$			operation
Add $-A$	$+$		10	11	11			
2-bit Shift		1	11	10	11	11		
Add $2A$	$+$	0	10	00	10			
			01	11	01	11		
2-bit Shift			00	01	11	01	11	
Add $-A$	$+$		10	11	11			
			11	01	10	01	11	-153

Radix-8 modified Booth's algorithm & canonical recoding

- można patrzeć na grupy 4 bitów (3 zmiany + referencja), ale to wymagałoby obliczenia 3A
- można wygenerować kanoniczną formę B, która ma jak najmniej bitów, ale ta operacja wymaga przejrzenia B od lewej do prawej, więc tracimy równoległość produkowania iloczynów cząstkowych

Użycie mniejszych układów mnożących

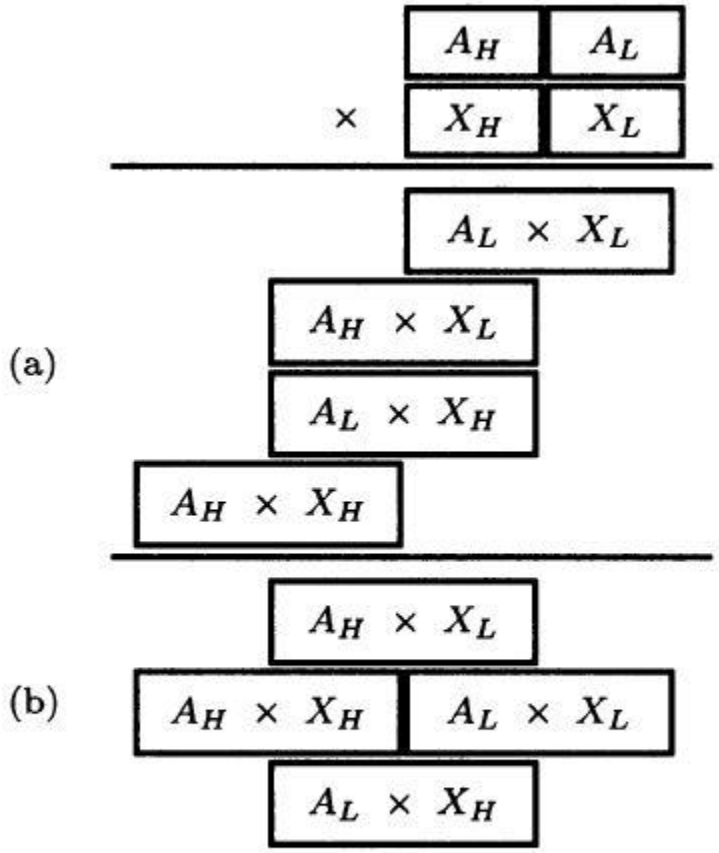


FIGURE 6.1 Aligning the four partial products in Equation (6.2).

Przykład z życia

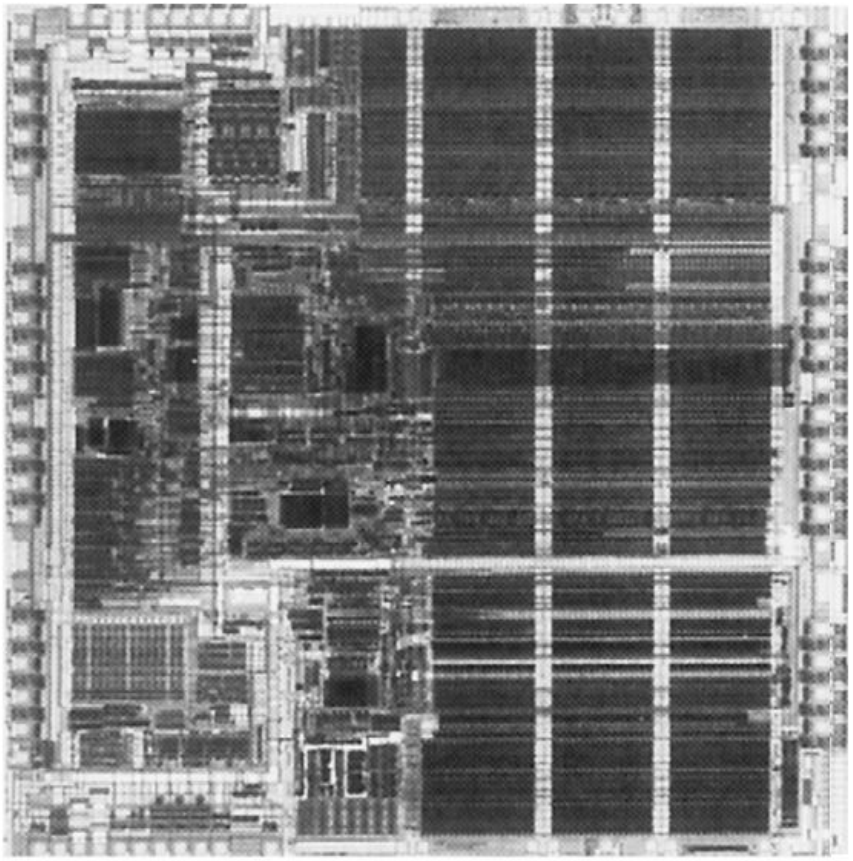
- Weitek 3364
- MIPS R3010
- Texas Instruments 8847
- wprowadzone w 1988, cykl około 40 ns (25 MHz)
- implementują dodawanie, odejmowanie, mnożenie, dzielenie według standardu IEEE (zatem również dodawanie i mnożenie liczb całkowitoliczbowych)

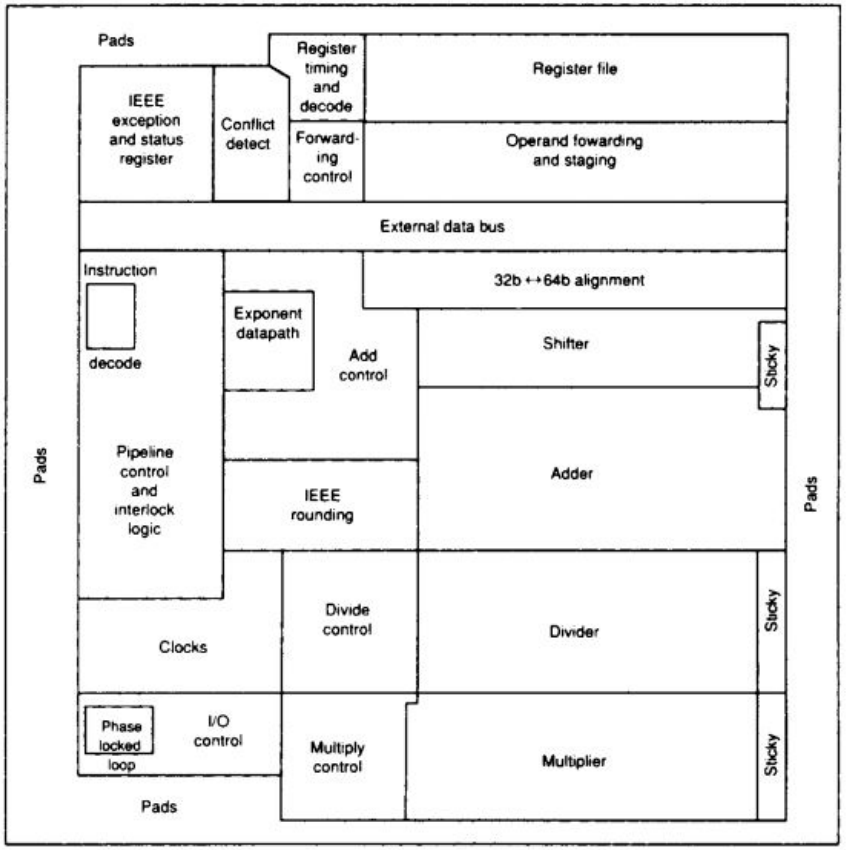
Przykład z życia

Features	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/square root	–	30	14

Figure J.36 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.

Przykład z życia





MIPS R3010

Przykład z życia

- MIPS
  - dodawanie: dzielimy bity na dwie połówki używając carry-select, wewnątrz bloków używamy hybrydy CLA i ripple-carry o różnych rozmiarach podbloków
  - mnożenie: radix-4 Booth recoding + technika parzyste/nieparzyste (CSLA)
- Weitek
  - dodawanie: carry-skip
  - mnożenie: radix-8 Booth recoding
- TI
  - dodawanie: carry-select
  - mnożenie: 3-etapowy potok na drzewie binarnym CSA (da się takie stworzyć używając systemu liczbowego o ujemnych cyfrach): obliczenie połowy bitów, obliczenie drugiej połowy bitów, zamiana wyniku na kod uzupełnień do dwóch
- trzy firmy miały podobne możliwości i ograniczenia, a każda wybrała inny algorytm dodawania i mnożenia

Koniec. Pytania?

Źródła

- Israel Koren, Computer Arithmetic Algorithms, 2nd Edition
- Hennessy, Patterson, Computer Architecture: A Quantitative Approach, 5th Edition, Appendix J