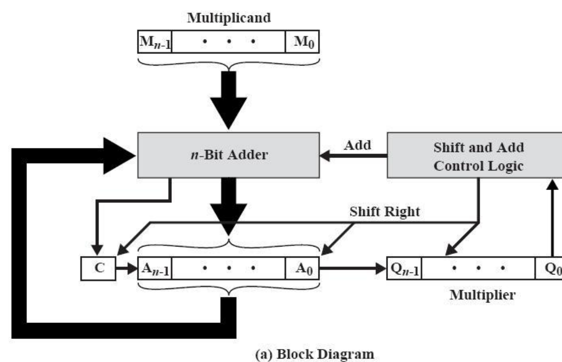
Arytmetyka całkowitoliczbowa

1 Mnożenie

1.1 Mnożenie maszynowe liczb w naturalnym kodzie binarnym

Najpierw przyjrzymy się mnożeniu liczb nieujemnych. Na rysunku 1 przedstawiamy układ symulujący mnożenie pisemne. Bierzemy pierwszą od prawej cyfrę mnożnika Q. Jeśli napotkamy jedynkę to dodajemy mnożną M do rejestru A i przesuwamy bity ciągu rejestrów CAQ o jeden w prawo. Końcowy wynik zawarty jest w rejestrach AQ. Oczywiście jeśli dane wejściowe są liczbami n-bitowymi to wynik daje się zapisać na $2n$-bitach.



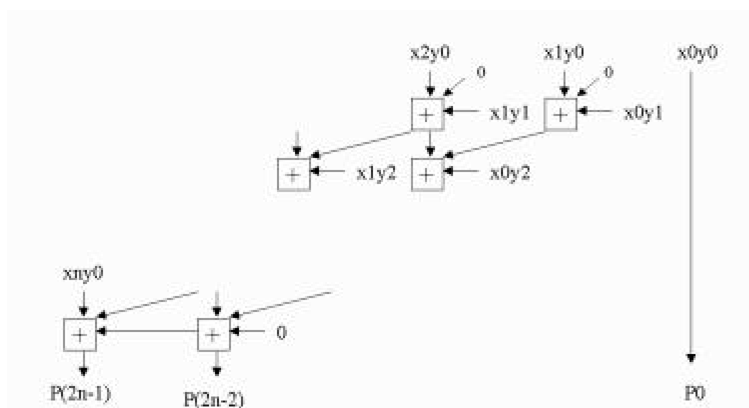
C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Rysunek 1: Schemat blokowy układu mnożącego oraz przykład obliczeń: $1011 \cdot 1101$

Oczywiście taki schemat mnożenia nie działa w przypadku reprezentacji uzupełnień do dwóch i liczb ujemnych. Najprostsze rozwiązanie tego problemu: przed mnożeniem przekształcić obydwie liczby na liczby dodatnie, przemnożyć i na koniec, jeśli wejściowe liczby miały różne znaki, zamienić wynik na ujemny. Nie jest to sposób zbyt elegancki. Możemy za to użyć **algorytmu Bootha**, o którym opowiemy za chwilę.

1.2 Mnożenie matrycowe

Wykonując pewne operacje równolegle możemy uzyskać całkowity czas mnożenia liniowy względem n (jeśli używamy zwykłego sumatora kaskadowego, to metoda z poprzedniego podrozdziału ma oczywiście czas proporcjonalny do kwadratu n). Pomysł jest prosty: używamy jednej bramki AND na pomnożenie każdego bitu mnożnika przez każdy bit mnożnej, a następnie sumujemy uzyskane iloczyny częściowe (używając sumatorów). Oczywiście zysk czasowy uzyskujemy kosztem złożoności układu: użyta liczb bramek jest proporcjonalna do kwadratu n.



Rysunek 2: Układ realizujący mnożenie matrycowe (*array multiplier*)

1.3 Algorytm Booth'a

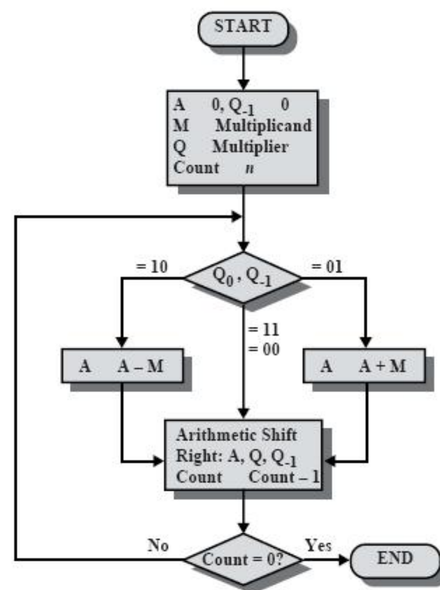
Algorytm Bootha, oprócz tego, że będzie działał poprawnie dla reprezentacji uzupełnień do dwóch (w jednakowy sposób dla liczb dodatnich jak i ujemnych), to w dodatku będzie potrafił „przeskakiwać" szybko zarówno nad ciągami zer jak i ciągami jedynek w mnożniku (przedstawiony poprzednio algorytm dla liczb dodatnich dobrze radzi sobie tylko z zerami – wtedy wy-

konuje jedynie operację przesunięcia, bez sumowania; oczywiście aby wykorzystać tą własność należałoby odpowiednio zaimplementować ten algorytm sprzętowo - w rozwiązaniu sugerowanym przeze mnie na wykładzie czas działanie całego układu jest niezależny od tego jakie liczby są na wejściu)). Prosta własność jaka jest tu wykorzystywana: ciąg jedynek na pozycjach o wagach 2^u do 2^v odpowiada wartości $2^{u+1} - 2^v$. Np. 001110 ma wartość $2^4 - 2^1 = 14$.

Załóżmy, że danymi wejściowymi są $M = b_{n-1}b_{n-2}\ldots b_1b_0$ oraz $Q = q_{n-1}q_{n-2}\ldots q_1q_0$. Przyjmijmy $q_{-1} = 0$ oraz $S = 0$. Przeglądamy kolejne bity q zaczynając od prawej strony (dla $i = 0, 1, \ldots n-1$). Jeśli $q_i = q_{i-1}$ nie robimy nic. Jeśli $q_i = 1$ oraz $q_{i-1} = 0$ (pierwsza jedynka) odejmujemy od S liczbę $M \cdot 2^i$. Jeśli $q_i = 0$ oraz $q_{i-1} = 1$ (poprzednia jedynka była ostatnia) dodajemy do S liczbę $M \cdot 2^i$.

Dlaczego algorytm działa, jeśli dane podamy w reprezentacji uzupełnień do 2? W rzeczywistości wyliczamy wartość nasępującego wyrażenia: $(q_{-1} - q_0) \cdot M \cdot 2^0 + (q_0 - q_1) \cdot M \cdot 2^1 \ldots (q_{n-2} - q_{n-1}) \cdot M \cdot 2^{n-1}$, które można zapisać jako $M \cdot (-q_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} q_i \cdot 2^i)$. A wyrażenie w nawiasie to dokładnie wartość liczby Q (przypomnij sobie jakie wagi mają bity w reprezentacji uzupełnień do 2).

Na rysunku 3 przedstawiamy blokowy zapis algorytmu Booth'a w wersji przeznaczonej do implementacji sprzętowej. Mnożnik i mnożna umieszczane są w rejestach Q i M. Dodatkowo używamy jednobitowego rejestru Q_{-1}. Logicznie rejestry ułożone są w ciąg AQQ_{-1}. Uwaga: przesunięcie jakie wykonujemy, gdy kolejne bity są jednakowe jest **przesunięciem arytmetycznym**: jeśli najbardziej znaczący bit kopiowanej liczby jest równy 1, to zwalniane przez niego pole uzupełniamy również jedynką, jeśli jest równy 0 - uzupełniamy zerem (sprawdziliśmy na wykładzie, że dopisanie kopii najbardziej znaczącego bitu z lewej strony liczby nie zmienia jej wartości w reprezentacji uzupełnień do dwóch).



Rysunek 3: Algorytm Booth'a

Na rysunkach 4 i 5 przedstawione są przykłady działania algorytmu Bootha.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Rysunek 4: Przykład zastosowania algorytmu Booth'a

```
    0111
   ×0011     (0)
11111001     1-0
0000000      1-1
000111       0-1
00010101     (21)
```

(a) (7) × (3) = (21)

```
    0111
   ×1101     (0)
11111001     1-0
0000111      0-1
111001       1-0
11101011     (-21)
```

(b) (7) × (-3) = (-21)

```
    1001
   ×0011     (0)
00000111     1-0
0000000      1-1
111001       0-1
11101011     (-21)
```

(c) (-7) × (3) = (-21)

```
    1001
   ×1101     (0)
00000111     1-0
1111001      0-1
000111       1-0
00010101     (21)
```

(d) (-7) × (-3) = (21)

Rysunek 5: Przykłady zastosowania algorytmu Booth'a

W pewnych sytuacjach algorytm Booth'a może być gorszy od prostego algorytmu przedstawionego na początku. Rozważmy ciąg bitów 01010101. W algorytmie Booth'a musimy wykonać aż osiem dodawań i odejmowań, podczas gdy algorytm prosty wymaga tylko czterech. Istnieją usprawnienia algorytmu Booth'a gwaratnujące, że dodawań i odejmnowań będzie najwyżej $n/2$.

2 Usprawnienie układów realizujących dodawanie i odejmowanie

Na jednym z poprzednich wykładów przedstawiliśmy układ sumatora kaskadowego. Wadą tego układu był długi czas obliczeń: sumator dodający i-te bity liczb wejściowych dawał poprawny wynik dopiero po tym jak otrzymał poprawne przeniesienie z sumatora $i-1$. Zakładając, że pojedynczy sumator podaje poprawne przeniesienie po czasie t, całe n-bitowe dodawanie trwa mniej więcej nt. Nieco dokładniej: załóżmy, że każdy poziom bramek logicznych powoduje opóźnienie o 1 jednostkę czasu (uproszczenie!). Wtedy każdy sumator produkuje wynik w czasie 2 (dwie bramki XOR), a przeniesienie w czasie 3 (XOR, AND, OR). Jeżeli przeniesienie zapiszemy jako $c_{i+1} = a_i b_i + a_i c_i + b_i c_i$ dostajemy przeniesienie w czasie 2. Użyjmy tej zmodyfikowanej wersji. Jaki jest całkowity czas obliczeń sumatora, np. 64-bitowego: $2 \cdot 64 = 128$.

Jednym z pomysłów na ulepszenie układu sumatora kaskadowego jest wyliczenie przeniesień dla kolejnych sumatorów szybciej, bez czekania na obliczenia poprzednich. Omówione poniżej rozwiązanie nazywa się metodą *podglądu przeniesienia* (*carry-lookahead*).

Oznaczmy $g_i := a_i b_i$ (*generowanie* przeniesienia na i-tym poziomie) oraz $p_i := a_i + b_i$ (*propagowanie* poprzedniego przeniesienia). Możemy zapisać: $c_{i+1} = g_i + p_i c_i$. I dalej $c_{i+i} = g_i + p_i(g_{i-1} + p_{i-1}c_{i-1})$. Aż do: $c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} p_{i-2} \cdot \ldots \cdot p_1 p_0 c_0$. Zatem możemy każde przeniesienie wyliczyć w czasie stałym (trzy bramki opóźnienia). Cały 64-bitowy sumator może działać wtedy w czasie 1+2+2=5 (wyliczenie p_i oraz g_i + wyliczenie c_i + działanie pojedynczego sumatora). Oczywisty problem: liczba wejść bramek AND i OR - potrzebujemy bramek o 64 wejściach. Niepraktyczne.

Pierwsze rozwiązanie: zbudujmy układ działający tak jak opisano wyżej dla 4-bitów i traktujmy go jako „blok". Przeniesienia między blokami podajemy w standardowy sposób. Zauważmy, że drugi blok dostaje przeniesienie po czasie 5, trzeci po czasie 9 (p_i i g_i liczy się wcześniej), itd. czwarty po 13, itd. Ostatni 16 ma przeniesienie wejściowe po czasie 61. Całkowity czas: 65. Wystarczą w tym przypadku bramki 4-wejściowe.

Kolejny pomysł: wprowadźmy funkcje propagowania P i G wyższego poziomu (jeden Blok wyższego poziomu będzie się składał z czterech bloków niższego): $P_0 = p_3 p_2 p_1 p_0$ - blok 0 propaguje przeniesienie, jeśli każdy jego bit propaguje. $G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$ - blok 0 generuje przeniesienie jeśli generuje je któryś z jego bitów i propagowane jest ono na wyjście bloku. Analogicznie: $P_1 = p_7 p_6 p_5 p_4$ oraz $G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$, i tak samo dla $i = 2, 3$.

Teraz: $c_4 = G_0 + P_0 c_0$, $c_8 = G_1 + P_1 c_4$, $c_{12} = G_2 + P_2 c_8$. Podobnie jak poprzednio rozwijamy te wzory podstawiając odpowiednie wyrażenia pod zmienne c_i pojawiające sie po lewych stronych równości.

Możemy wprowadzić jeszcze jeden poziom i wyliczać przeniesienia pomiędzy Blokami drugiego poziomu: c_{16}, c_{32}, c_{48}.

Stosując tą technikę dla sumatorów n-bitowych uzyskujemy w efekcie sumator o czasie działania rzędu $\log n$.