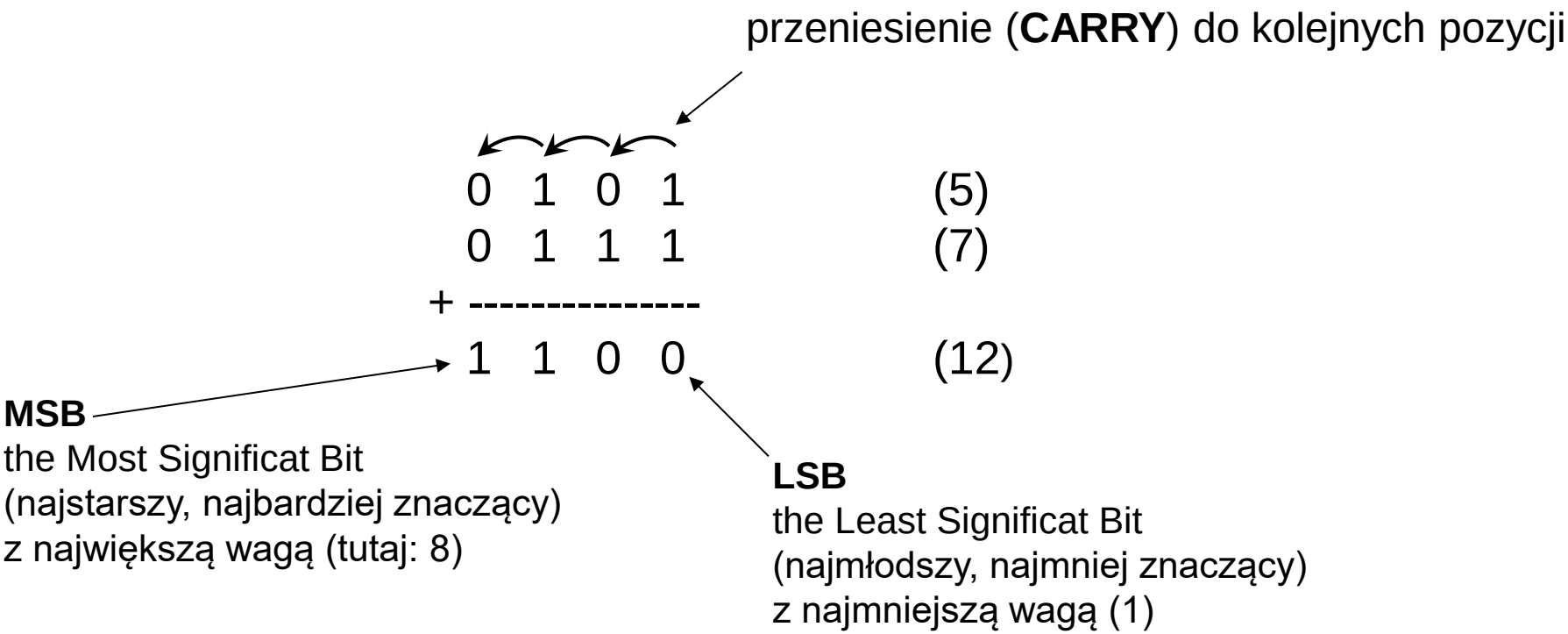
Arytmetyka komputerowa

- dodawanie
- odejmowanie / kod U2 (uzupełnień do dwóch)
- przesunięcia bitowe i arytmetyczne
- rozszerzenie długości słowa
- operacje na bitach testowanie/ustawianie/kasowanie/zmiana
- mnożenie
- dzielenie

Dodawanie

w systemie binarnym jest trywialne…



- przykład – półbajty (nibbles, 4 bity, zakres bez znaku od 0 do 15),
  niemniej zasady przeprowadzania operacji są takie same dla dłuższych słów.

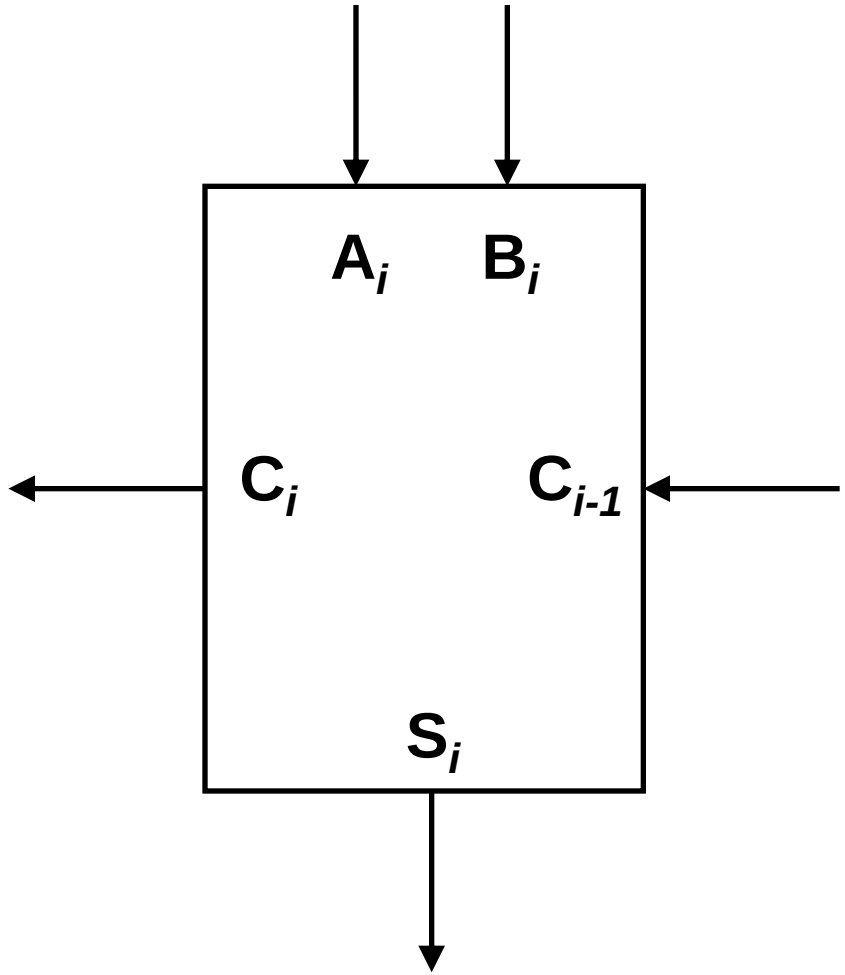
Sumator pełny (Full Adder - FA)
(jednobitowy)

A_i, B_i – wejścia *i*-tch bitów dodawanych argumentów

C_{i-1} – wej. przeniesienia z poprzedniej pozycji

C_i - wyj. przeniesienia do następnej pozycji

S_i – *i*-ty bit wyniku (sumy)



• tzw. **półsumator** (Half Adder - HA) nie posiada wejścia c_{i-1}

Sumator pełny

tablica prawdy

wszystkie kombinacje zmiennych wejściowych i odpowiadające im stany wyjść:

C_{i-1}	B_i	A_i	C_i	S_i
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

- stan każdego wyjścia określa funkcja logiczna* :

$$C_i = A_i B_i + C_{i-1} B_i + C_{i-1} A_i$$

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

- funkcje te można zapisać na bezpośrednio na podstawie tablicy prawdy, a następnie minimalizować na drodze przekształceń algebraicznych,

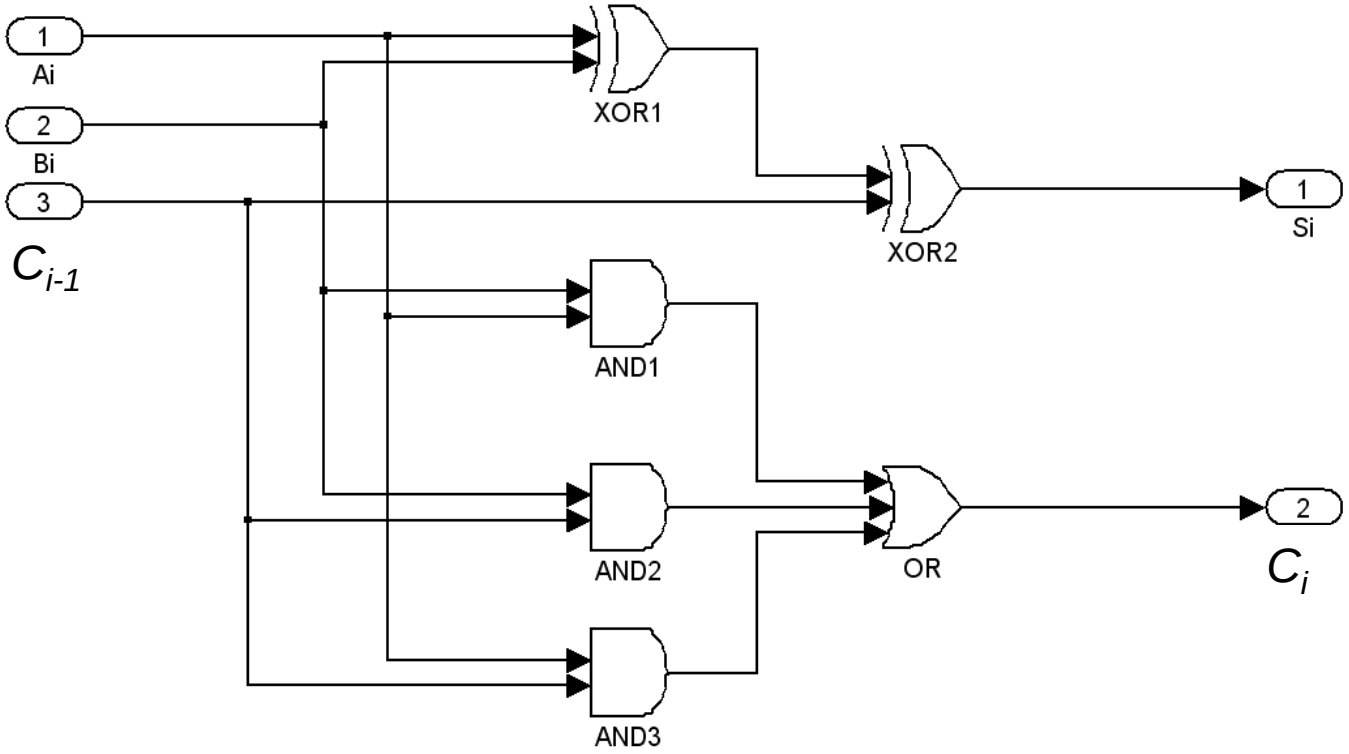
- minimalizację można przeprowadzić również używając metody graficznej (np. tablice Karnaugha).

Sumator pełny (1-bitowy)

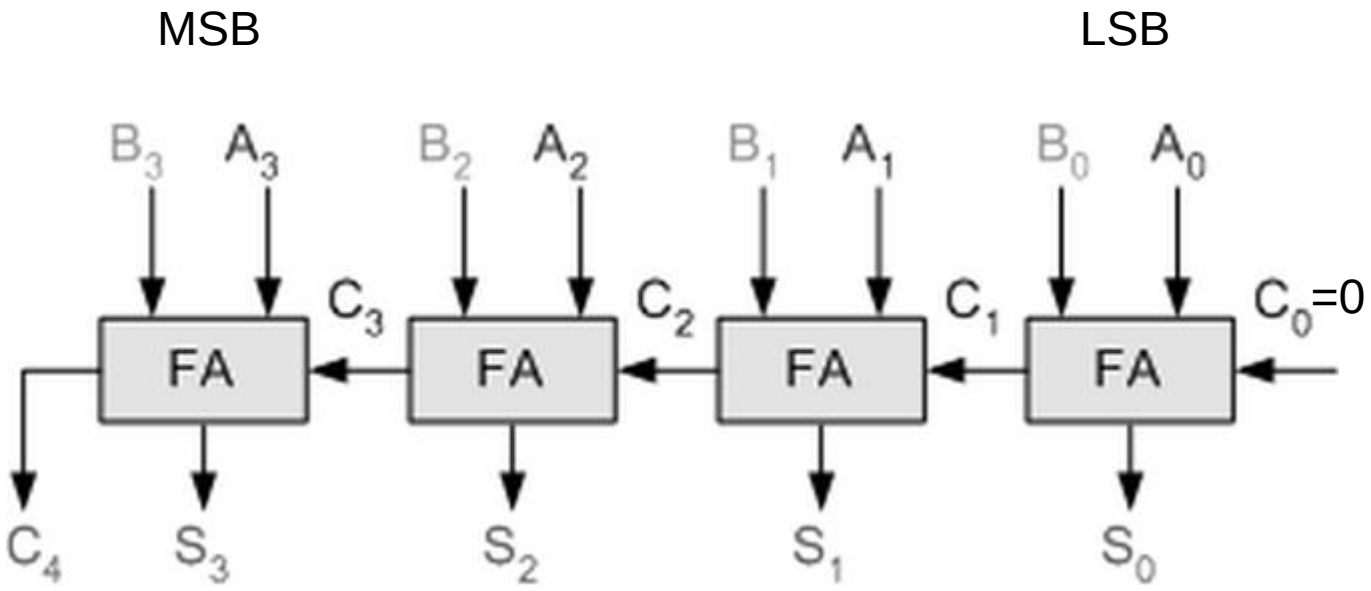
przykładowe (jedno z możliwych…)
rozwiązanie układowe:

$$C_i = A_i B_i + C_{i-1} B_i + C_{i-1} A_i$$

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



Prosty sumator wielobitowy



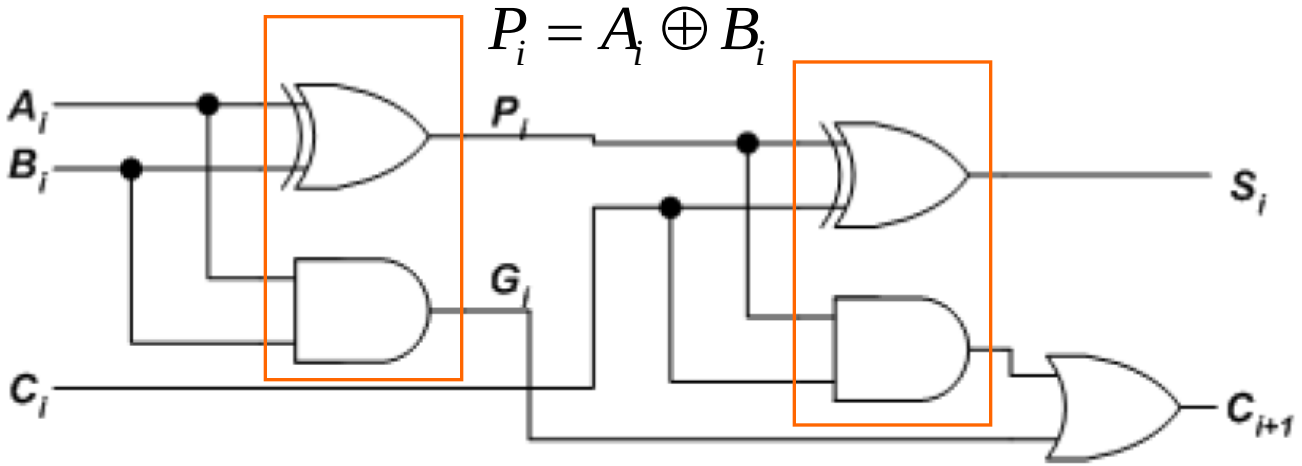
czas propagacji sygnału przeniesienia: *2nt*

(dla układu jednobitowego z poprzedniego schematu)

n – liczba bitów w słowie, *t* - czas propagacji jednej bramki logicznej

Sumator z przeniesieniami jednoczesnymi - Carry Look Ahead - CLA

(sumator pełny = dwa półsumatory)



G_i – generacja przeniesienia przez sumator i-tej pozycji (bez C_{i-1})
P_i – propagacja przeniesienia przez sumator i-tej pozycji

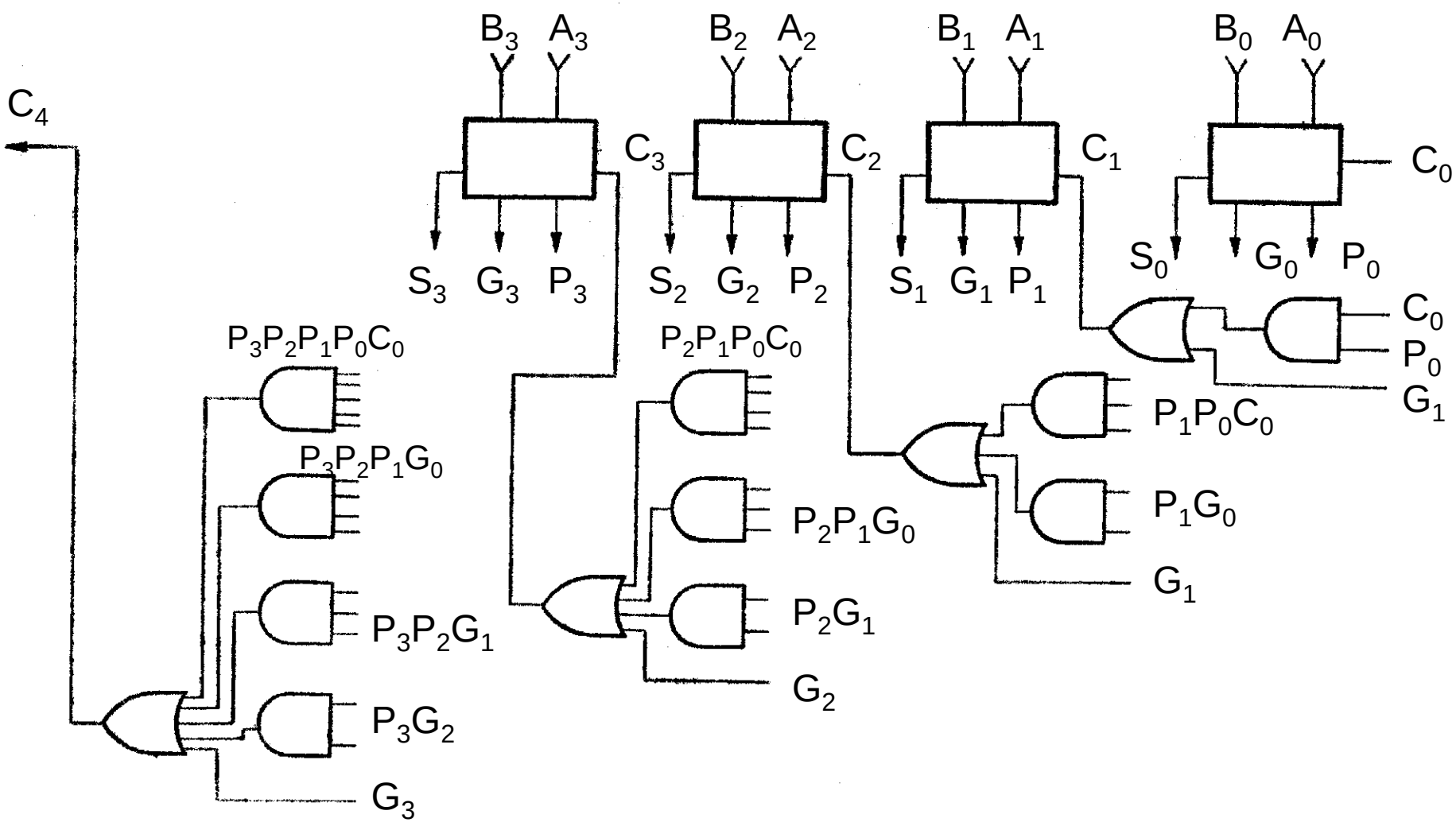
$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 \left(G_0 + P_0 C_0 \right) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

Sumator z przeniesieniami jednoczesnymi - Carry Look Ahead – CLA

4-bitowy



Czas propagacji: 4t (dla prostego sumatora kaskadowego 2*4t=8t).

Kod uzupełnień do 2 - U2 (two's complement)

kodowanie liczb ze znakiem

• różnica między naturalnym kodem binarnym (NB):

- waga najstarszej pozycji ze znakiem minus

$$N_{U2} = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad a_i \in \{0,1\}$$

np.
półbajt (4bit) wagi **8** 4 2 1 (NB) zakres: 0,…,15
-8 4 2 1 (U2) zakres: -8,…,7

bajt bez znaku: wagi **128**, 64,…, 1 zakres: 0 ,…, 255
bajt ze znakiem: **-128**, 64,…, 1 zakres: -128,…,127

• nie ma „strat" np. podwójnego zera (-0,+0), wykorzystane wszystkie kombinacje „0" i „1"

• MSB oprócz **wartości** określa również **znak** liczby

• ten sam ciąg „zer i jedynek" można interpretować jako różne liczby:
np. „1001" jako 9 (NB) lub -7 w U2…

Odejmowanie

$$A - B = A + (-B)$$

Jak obliczyć „-*B*” ?

$$B + (-B) = 0 \qquad \text{(x)}$$

Z własności algebry boolowskiej:

$$B + \overline{B} = 1$$

analogicznie, dla słów wielobitowych:

$$B + \overline{B} = 1 \ldots 1$$

+1 do obu stron:

$$B + \overline{B} + 1 = 0 \qquad \text{(y), przeniesienie z MSB zaniedbać…}$$

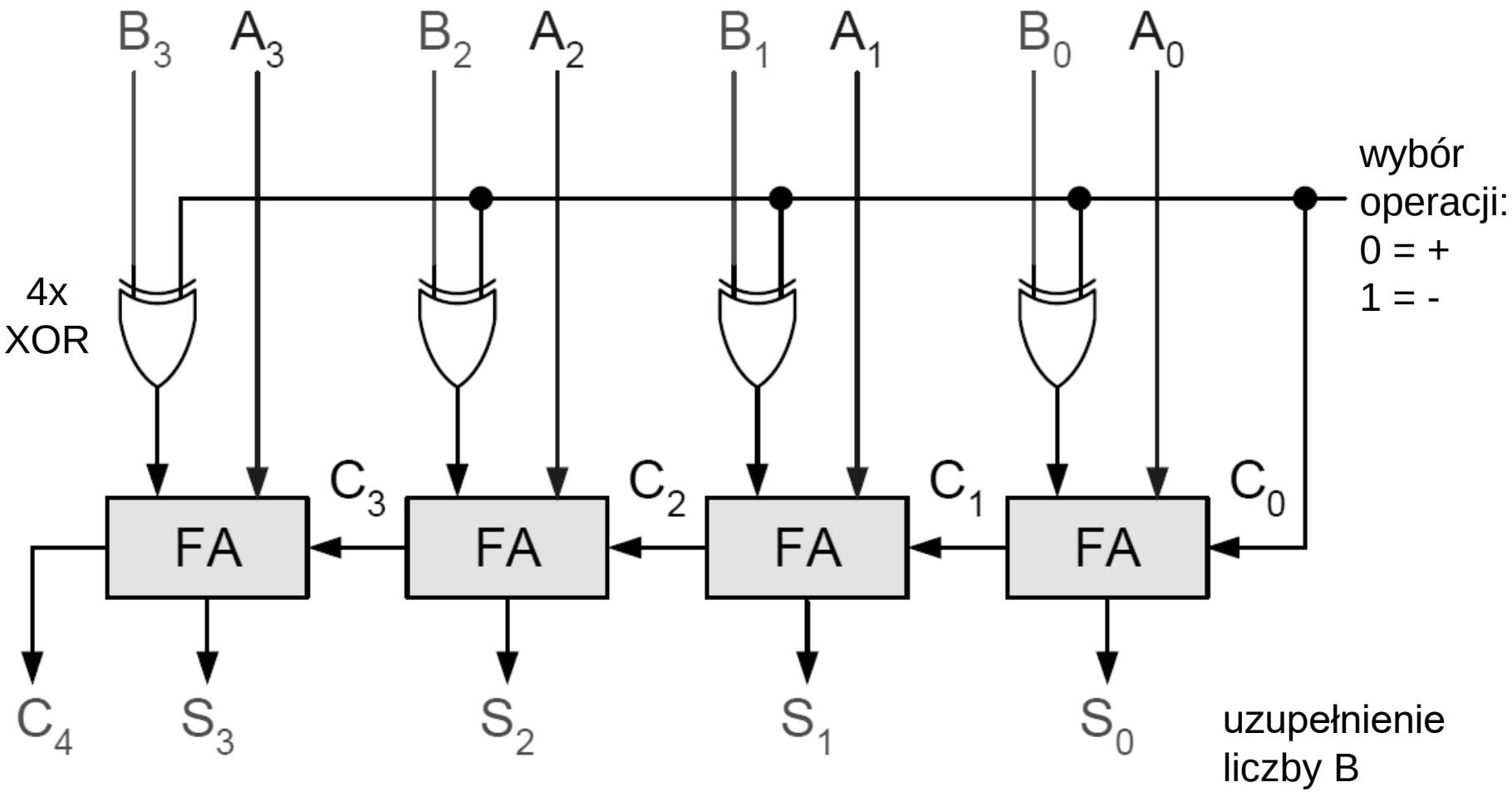
porównując (x) z (y) otrzymujemy **uzupełnienie do dwóch liczby *B*** (two’s complement):

$$-B = \overline{B} + 1$$

- **w arytmetyce komputerowej odejmowanie *A - B* przeprowadza się poprzez dodanie uzupełnienia (do dwóch) liczby *B:***

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

Sumator można bardzo łatwo rozbudować o możliwość wykonywania drugiej operacji - odejmowania…



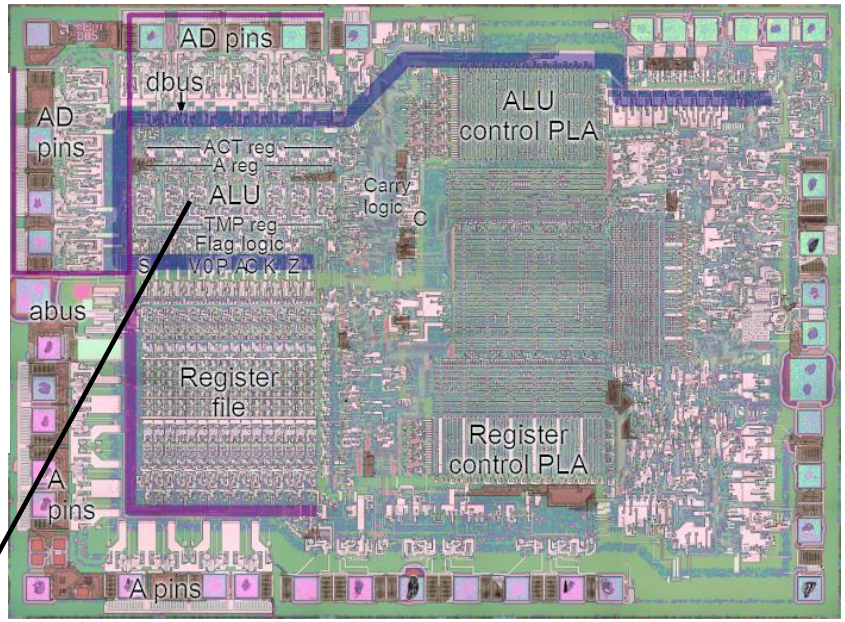
$$A - B = A + \overline{B} + 1$$

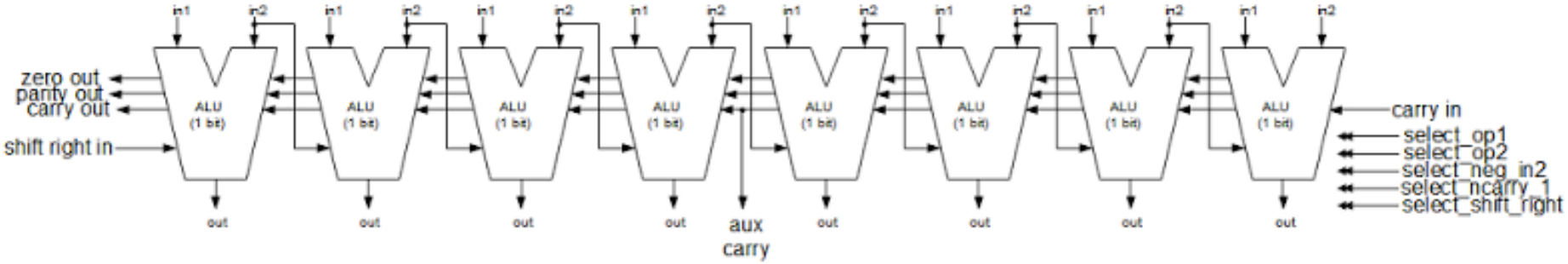
$$-B = \overline{B} + 1$$

Jednostka arytmetyczno logiczna (ALU) 8-bitowego procesora Intel 8085

www.righto.com/2013/07/reverse-engineering-8085s-alu-and-its.html

Operation	*select_neg_in2*	*select_op1*	*select_op2*	*select_shift_right*	*select_ncarry_1*	Carry in/out
or	0	0	0	0	1	1
add	0	1	0	0	0	/carry
xor	0	1	0	0	1	1
and	0	1	1	0	1	0
shift right	0	0	1	1	1	0
complement	1	0	0	0	1	1
subtract	1	1	0	0	0	borrow





Działania na liczbach bez znaku

- **poprawność obliczeń**
- **przekroczenie zakresu**

przykład: dodawanie półbajtów (zakres: 0…15)

C_{MSB}=0

```
  0110 (6)
  0111 (7)
+ -------
  1101 (13) OK
```

C_{MSB}=1

```
  0111 (7)
  1101 (13)
+ -------
  0100 (4) ?? (4bit)
```

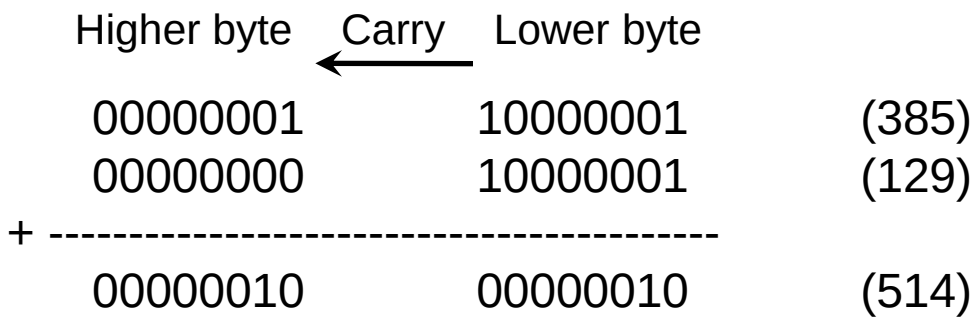
10100 (16+4) OK (potrzeba 5 bitów)

interpretacja przeniesienia (CARRY) z najstarszej pozycji (MSB) podczas dodawania liczb bez znaku jest prosta i jednoznaczna:

- wystąpienie przeniesienia oznacza przekroczenie zakresu,

- sam wynik jest błędny, ale sygnał przeniesienia można wykorzystać do obliczeń na dłuższych typach danych.

Dodawanie słów dłuższych niż szerokość magistrali/rejestrów w ALU:



np. procesor 32-bitowy (80386 – Pentium 4):

1. liczba 64-bitowa w parze %ebx (H) %eax (L)
2. liczba 64-bitowa w parze %edx (H) %ecx (L)

```
add     %ecx,%eax
adc     %edx,%ebx        #adc = add with carry
```

wynik zapisany zostaje w parze rejestrów %ebx : %eax (H : L).

- operacja musi być przeprowadzona etapami
  (na porcjach danych odpowiadających szerokości szyny ALU i rejestrów),

- wymaga zatem użycia większej liczby instrukcji i zajmuje odpowiednio więcej miejsca w pamięci oraz czasu (cykli zegarowych).

Działania na liczbach bez znaku

- **poprawność obliczeń**
- **przekroczenie zakresu**

przykład: **odejmowanie (dodawanie uzupełnienia) półbajtów** (zakres: 0…15)

$C_{MSB}=1$

```
  0101 (5)
  1111 (-1)
+ -------
  0100 (4)
```

$C_{MSB}=0$

```
  0101 (5)
  1001 (-7)
+ -------
  1110 (?? 14)
```

„Surowe” (wychodzące bezpośrednio z ALU) przeniesienie z najstarszej pozycji (MSB) podczas odejmowania zachowuje się **odwrotnie** niż podczas dodawania:

- wystąpienie tego przeniesienia oznacza prawidłowy wynik,
- jego brak – przekroczenie zakresu.

- W powyższy sposób ustawiana jest flaga CARRY w np. procesorach ARM,

- w innych rozwiązaniach (**x86**, AVR) **przeniesienie po odejmowaniu jest negowane**, (więc flaga CARRY zachowuje się tak, jak w przypadku dodawania).

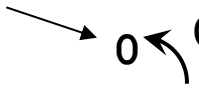
Subtract with CARRY (ARM)

Przykład (4-bitowy):

35 – 17

```
          H       L
A       0010    0011    (35)
B       0001    0001    (17)
    - --------------
        ????    ????
```

/C=0 – zanegowane C w rej. flag:



C=1 - flaga C bezpośrednio z ALU

```
          0
        0010  0011
/B+1    1111  1111
 + --------------
        0001  0010
```

dodawanie uzupełnień B

pierwsza część (najmłodsza)

$$A_L - B_L = A_L + /B_L + 1$$

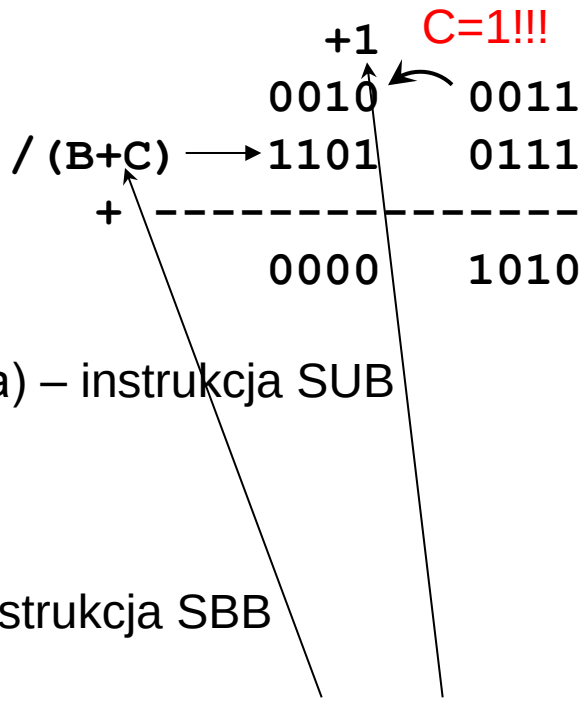
druga (i kolejne części):

$$A_H - B_H + /C = A_H + /B_H + 1 + /C$$

Subtract with BORROW (x86, AVR)

przykład:
35 - 25

```
          H       L
A       0010    0011    (35)
B       0001    1001    (25)
    - ------------
        ????    ????
```



pierwsza część (najmłodsza) – instrukcja SUB

$A_L - B_L = A_L + /B_L + 1$

druga (i kolejne części): - instrukcja SBB

$A_H - B_H - C = A_H - (B_H + C) = A_H + /(B_H + C) + 1$

Działania na liczbach ze znakiem (U2)

- **poprawność obliczeń**
- **przekroczenie zakresu**

Przykłady obliczeń na **półbajtach** (zakres: -8…+7):

$C_{MSB-1}=0$, $C_{MSB}=0$

```
  0 0 1 0  (2)
  1 1 0 1 (-3)
+ ----------
  1 1 1 1 (-1) OK
```

$C_{MSB-1}=1$, $C_{MSB}=1$

```
  1 1 1 0 (-2)
  1 1 0 1 (-3)
+ ----------
  1 0 1 1 (-5) OK
```

$C_{MSB-1}=1$, $C_{MSB}=0$

```
  0 0 1 1 (3)
  0 1 1 0 (6)
+ ----------
  1 0 0 1 (-7) ?? 9 ?? - nadmiar!
```

$C_{MSB-1}=0$, $C_{MSB}=1$

```
  1 1 0 1 (-3)
  1 0 1 0 (-6)
+ ----------
  0 1 1 1 (7) ?? - nadmiar!
```

W przypadku dodawania (odejmowania) liczb ze znakiem (U2) **przekroczenie zakresu sygnalizowane jest ustawieniem flagi OVERFLOW** obliczanej jako:
OVERFLOW = C_{MSB} XOR C_{MSB-1}.

Flagi (arytmetyczne) procesora

- bity w dedykowanym rejestrze procesora (rejestrze flag, statusu, stanu itp.),
- **ogólnie: ustawiane na podstawie wyniku każdej operacji arytm./ logicznej,**
- **w praktyce: – sprawdzić w dokumentacji procesora!**

  np. **instrukcje inc i dec, służące głównie do modyfikacji licznika iteracji, nie ustawiają flagi CARRY**
  (ponieważ przeniesienia mogą się przenosić podczas np. obliczeń w pętli).

- **Instrukcje porównująco-testujące (np. cmp i test w x86) ustawiają tylko flagi, nie zapisują nigdzie wyniku!**
- **Na podstawie stanu flag są wykonywane (lub ignorowane) skoki warunkowe.**
- W niektórych architekturach (ARM) instrukcje arytmetyczne domyślnie nie ustawiają flag. Ustawieniem flag steruje dodatkowe pole bitowe w kodzie rozkazu.

Flagi (arytmetyczne) procesora

- procesor nie posiada dedykowanych instrukcji do dodawania i odejmowania liczb ze znakiem i bez znaku (w przeciwieństwie np. do instr. mnożenia i dzielenia),

- pobiera jedynie „ciągi zer i jedynek” (o danej długości) i przetwarza je w ściśle określony (pokazany na poprzednich slajdach) sposób,

- oraz **ustawia wszystkie* flagi arytmetyczne**:

CARRY – przeniesienie (liczby bez znaku),
OVERFLOW – nadmiar (ze znakiem),
SIGN = bit znaku = najstarszy bit (MSB),
ZERO (stan wysoki gdy wynik operacji wynosi zero),
AUX CARRY – przeniesienie pomocnicze (między młodszym a starszym półbajtem),
PARITY – wskazuje, czy liczba jedynek w najmłodszym bajcie jest parzysta.

- **Od programisty zależy interpretacja wyników i sens używania poszczególnych flag** (np. przy skokach warunkowych).

*to zawsze należy sprawdzić w dokumentacji danego procesora. Np. w przypadku instrukcji typu ADD / SUB / CMP ustawiane są wszystkie ww. flagi. Instrukcja INC nie modyfikuje CARRY, a po operacji XOR stan AUX CARRY jest nieokreślony…

Przesunięcia bitowe

logiczne - bitowe : liczby bez znaku

np. zmienna 8 bitowa: `1001 1110`
w lewo o 1 bit `Carry < 1 < 0011 1100`
w prawo o 1 bit `0100 1111 > 0 > Carry`

• **na „zwolnione” miejsce wpisywane jest zawsze zero!**

w x86 instrukcje: **`SHL i SHR $liczba_bitów , %rejestr/pamięć`**

arytmetyczne : liczby ze znakiem

np. zmienna 8 bitowa: `1001 1110`
w lewo o 1 bit `Carry < 1 < 0011 1100` zmiana znaku = nadmiar (jak „+” U2)
w prawo o 1 bit `1100 1111 > 0 > Carry`

• w lewo – tak jak przesunięcie logiczne,
• **w prawo: powielona poprzednia wartość najstarszego bitu (zachowuje znak).**

w x86 instrukcje: **`SAL i SAR $liczba_bitów , %rejestr/pamięć`**

Przesunięcia służą również za szybkie mnożenie i dzielenie przez stałą - potęgę 2.

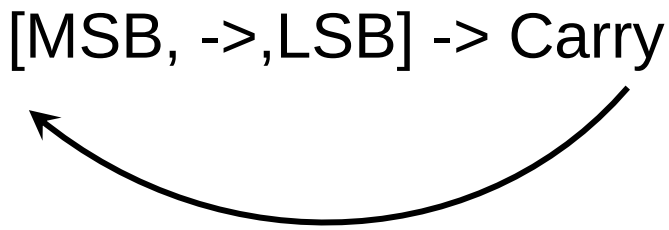
Obroty bitowe

„zwykłe” w lewo i w prawo (ROtate Left/Right)

`ROR i ROL $liczba_bitów , %rejestr/pamięć`

poprzez flagę przeniesienia – Rotate through Carry Left/Right

`RCR i RCL $liczba_bitów , %rejestr/pamięć`



Rozszerzenie długości słowa 1/3

np. z 4 na 8 bitów:

bez znaku - ZERO EXTEND

0101 -> 0000 0101 – uzupełnienie bardziej znaczącej części zerami,

ze znakiem – SIGN EXTENT

0101 -> **0000** 0101 (5)
1101 -> **1111** 1101 (-3)

uzupełnienie bardziej znaczącej części zgodnie ze znakiem liczby!

w x86 instrukcje typu **`MOVZX, MOVSX`**…

Rozszerzenie długości słowa 2/3

Architektura x86–64, rejestry 64-bitowe

• operacje 8- i 16-bitowe np. mov nie modyfikują starszych, niewykorzystanych części rejestrów:

```
movabs $0,%rax      %rax = 0x0000000000000000 (64 zera)
not %rax            %rax = 0xFFFFFFFFFFFFFFFF (64 jedynki)
mov $0,%ah          %rax = 0xFFFFFFFFFFFF00FF
```

w wyniku starsze 56 bitów=1 i młodsze 8 wyzerowane

• operacje 32-bitowe zerują starszą część rejestru:

```
movabs $0,%rax      %rax =  0x0000000000000000 (64 zera)
not %rax            %rax =  0xFFFFFFFFFFFFFFFF (64 jedynki)
mov $0,%eax         %rax =  0x0000000000000000
```

Rozszerzenie długości słowa 3/3

**Architektura x86–64 i używanie stałych,
przekazywanych bezpośrednio w kodzie rozkazu (immediate):**

• stałe 64-bitowe można ładować do rejestru jedną z rodziny instrukcji mov:

```
mov $0x0000000000000000 , %rax    (mov $0, %eax)
mov $0x00000000FFFFFFFF , %rax    (movabs $stała64, %rax)
mov $0xFFFFFFFFFFFFFFFF , %rax    (movabs $stała64, %rax)
```

• inne operacje np. arytmetyczno-logiczne (or, and…) na rejestrach 64 bitowych umożliwiają użycie tylko stałej 32-bitowej rozszerzanej do 64 bitów z uwzględnieniem jej znaku:

```
mov   $0,%rax                    %rax = 0x0000000000000000
or    $0x80000000,%rax           %rax = 0xFFFFFFFF80000000 (błąd w gas!)
or    $0xFFFFFFFF80000000,%rax   (syntax AT&T, zgodne z gas, wynik jak wyżej)

mov   $0,%rax                  %rax = 0x0000000000000000
or    $0x70000000,%rax         %rax = 0x0000000070000000
mov   $0,%rax                  %rax = 0x0000000000000000
or    $0x80000000,%eax         %rax = 0x0000000080000000
```

Operacje na bitach

testowanie (sprawdzanie) bitu znajdującego się na danej pozycji:

```
      0011 0110
      0001 0000 – maska z zer i jedynki na testowanej pozycji
AND ----------
      0001 0000 – wynik = waga testowanej pozycji testowany dany bit = 1,
                  wynik = 0 gdy testowany bit = 0.
```

W x86 oprócz instrukcji **and** jest również rozkaz:

test argument1 , argument2

wykonuje iloczyn logiczny (AND) bez zapisywania wyniku,
tylko ustawia flagi procesora, czyli nie niszczy (nie nadpisuje) drugiego argumentu.

Operacje na bitach

zerowanie bitu na danej pozycji:

```
      0011 0110
      1110 1111 – maska z jedynek i zera na pozycji kasowanej
AND ----------
      0010 0110
```

ustawianie bitu na danej pozycji:

```
      0011 0110
      0000 1000 – maska z zer i jedynki na ustawianej pozycji
 OR ----------
      0011 1110
```

oczywiście ww. operacje można stosować dla grup kilku bitów…

Operacje na bitach

zmiana wartości bitu na danej pozycji **na wartość przeciwną**:

0011 0110
1111 0000 – jedynki w masce – zmiana wartości na przeciwną
XOR ---------- zera – pozostawienie bez zmian.
1100 0110

Mnożenie liczb całkowitych (bez znaku)

```
          1011 (11)
          0101 (5)
        * -------
          1011
         0000
        1011
       0000
    + -----------
      00110111 (55)
```

MSB Carry = 0 →

dane: *n*-bitowe argumenty *a* - mnożna, *b* - mnożnik, **iloczyn 2*n* bitów:** *p*=0, maska z przesuwaną „1” do testowania kolejnych bitów mnożnika.

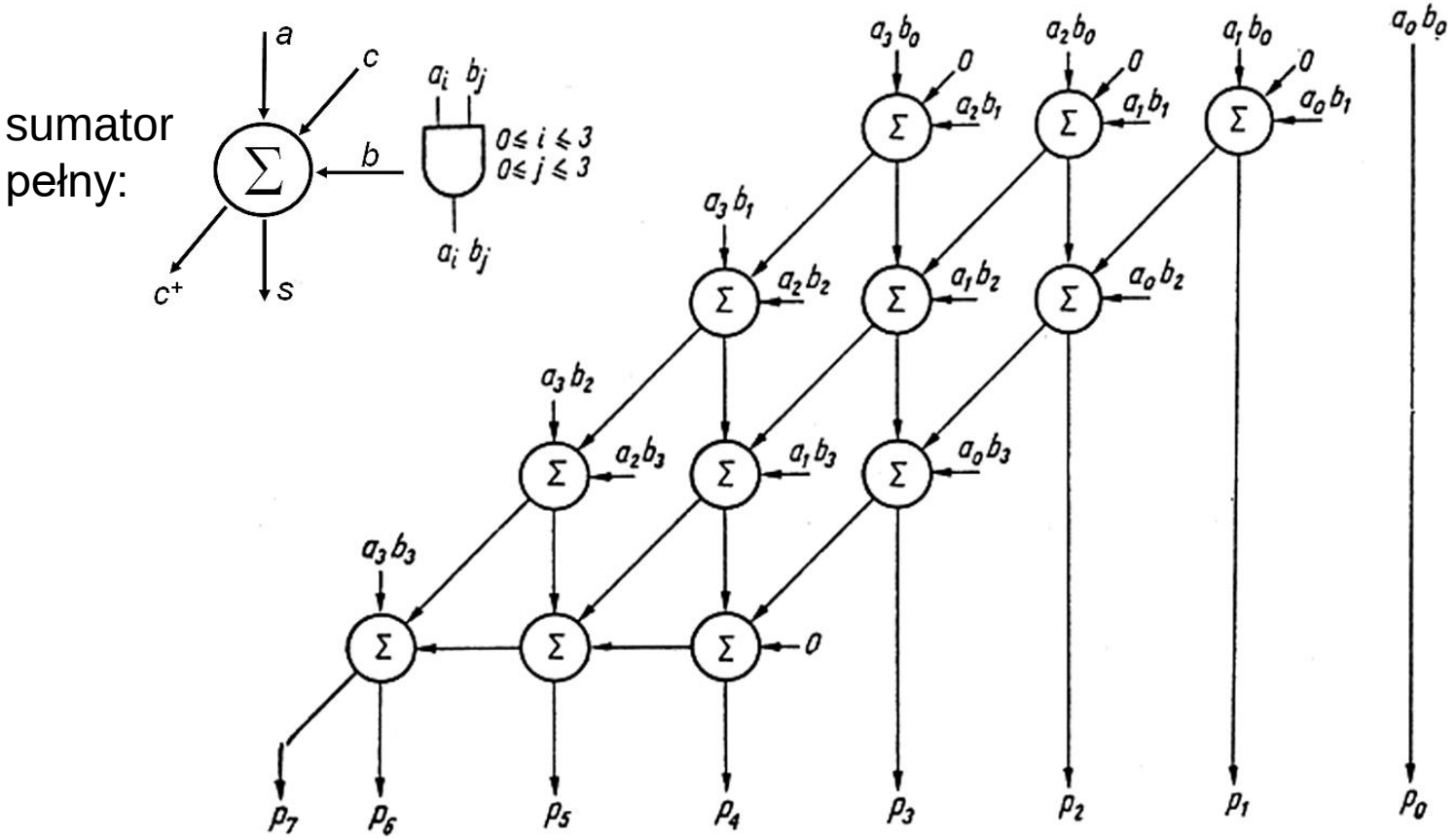
```
for (i=0; i<n; i++) {
      testuj i-ty bit w b: jeśli bit=1 to p:=p+a
      przesuń a o 1 bit w lewo
}
```

- proste, ale liczba iteracji (cykli zegarowych - tym samym czas całej operacji) zależy od długości mnożonych słów…

Mnożenie liczb całkowitych (bez znaku)

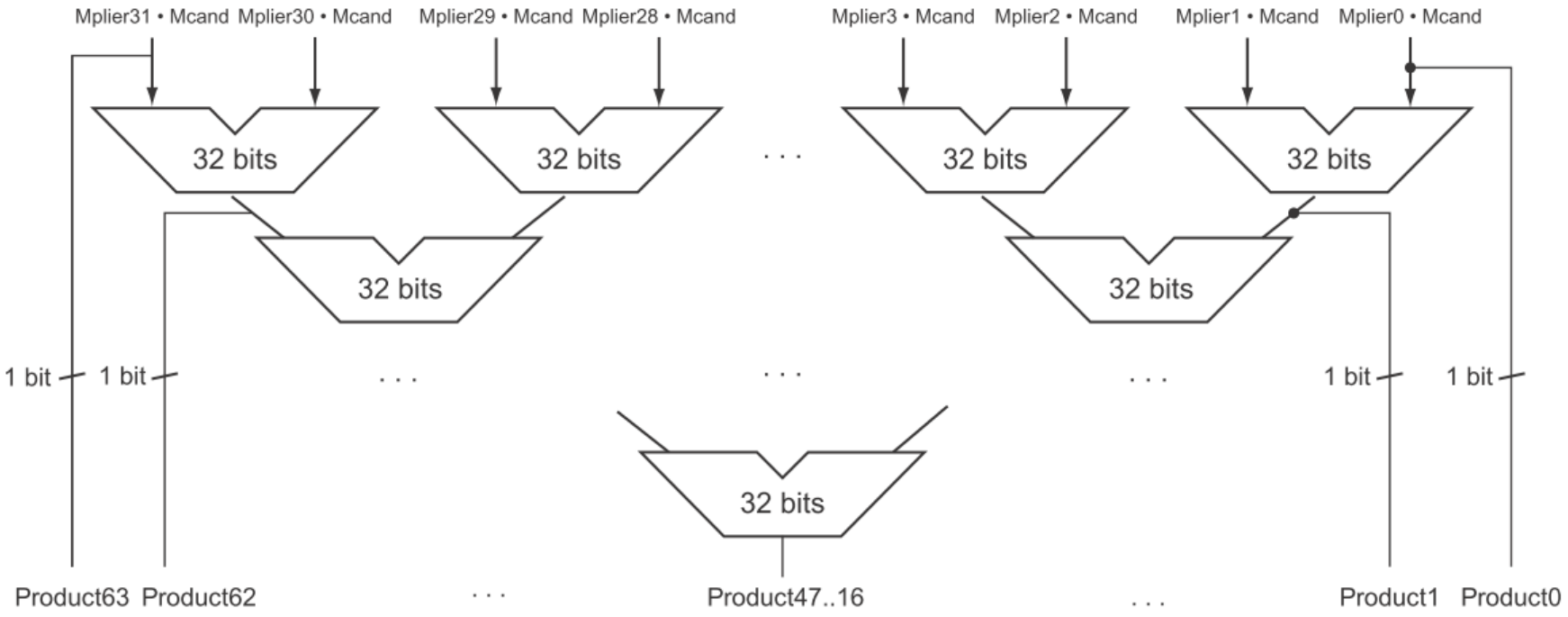
mnożący układ macierzowy

- zasada działania – mnożenie „w słupku”…
- zaleta: duża szybkość, (ograniczona czasem propagacji przez bramki logiczne w najdłuższej ścieżce sygnałowej).



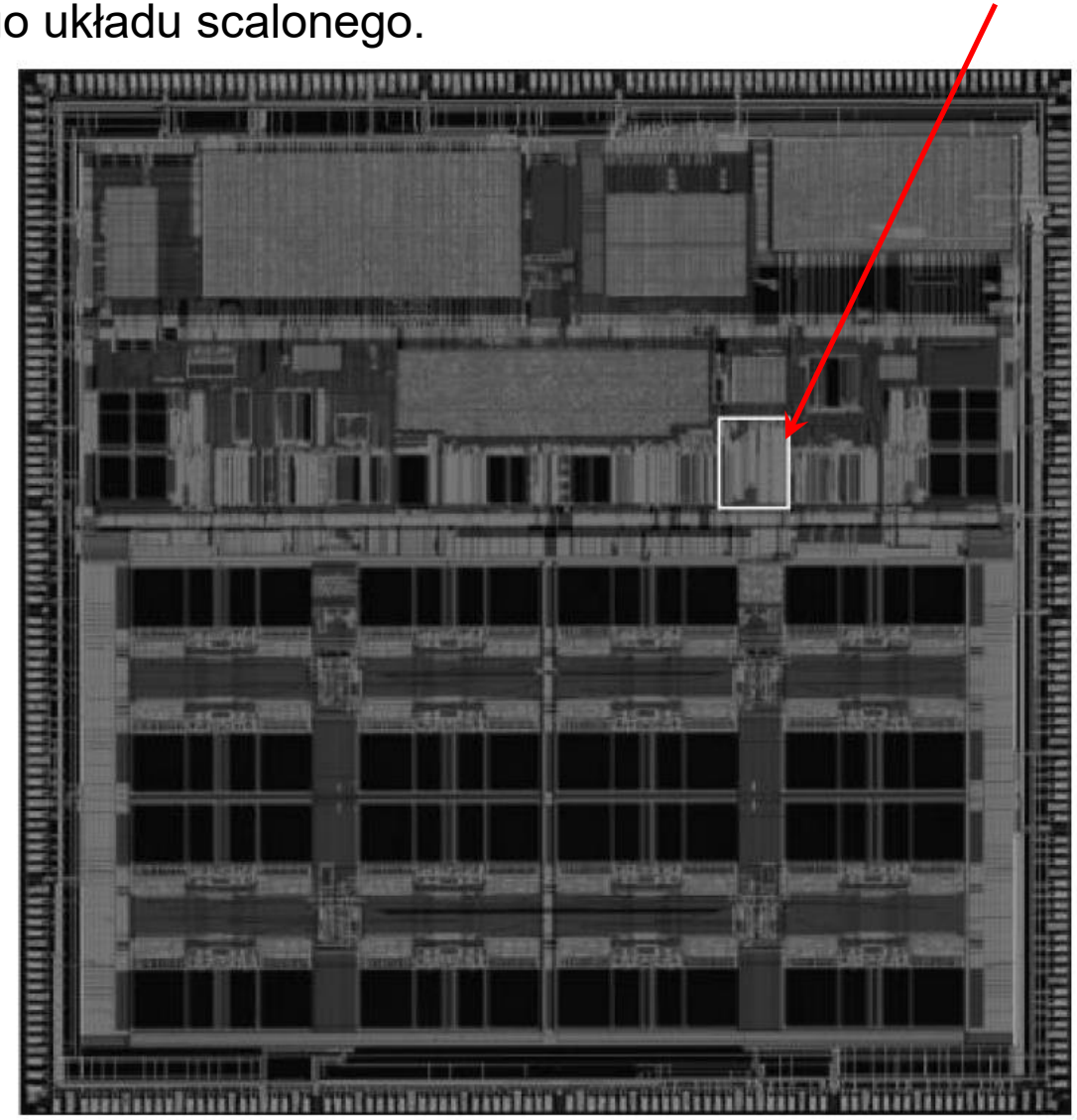
Mnożenie liczb całkowitych mnożący układ macierzowy – inna topologia.

Czas mnożenia dwóch słów 32-bitowych: 5 * czas dodawania dwóch liczb 32-bitowych.



Struktura procesora ARM10200

Szerokie (np. 32-bitowe) macierzowe układy mnożące są stosunkowo złożone (liczba tranzystorów/bramek) zajmują więc stosunkowo dużą powierzchnię w strukturze całego układu scalonego.

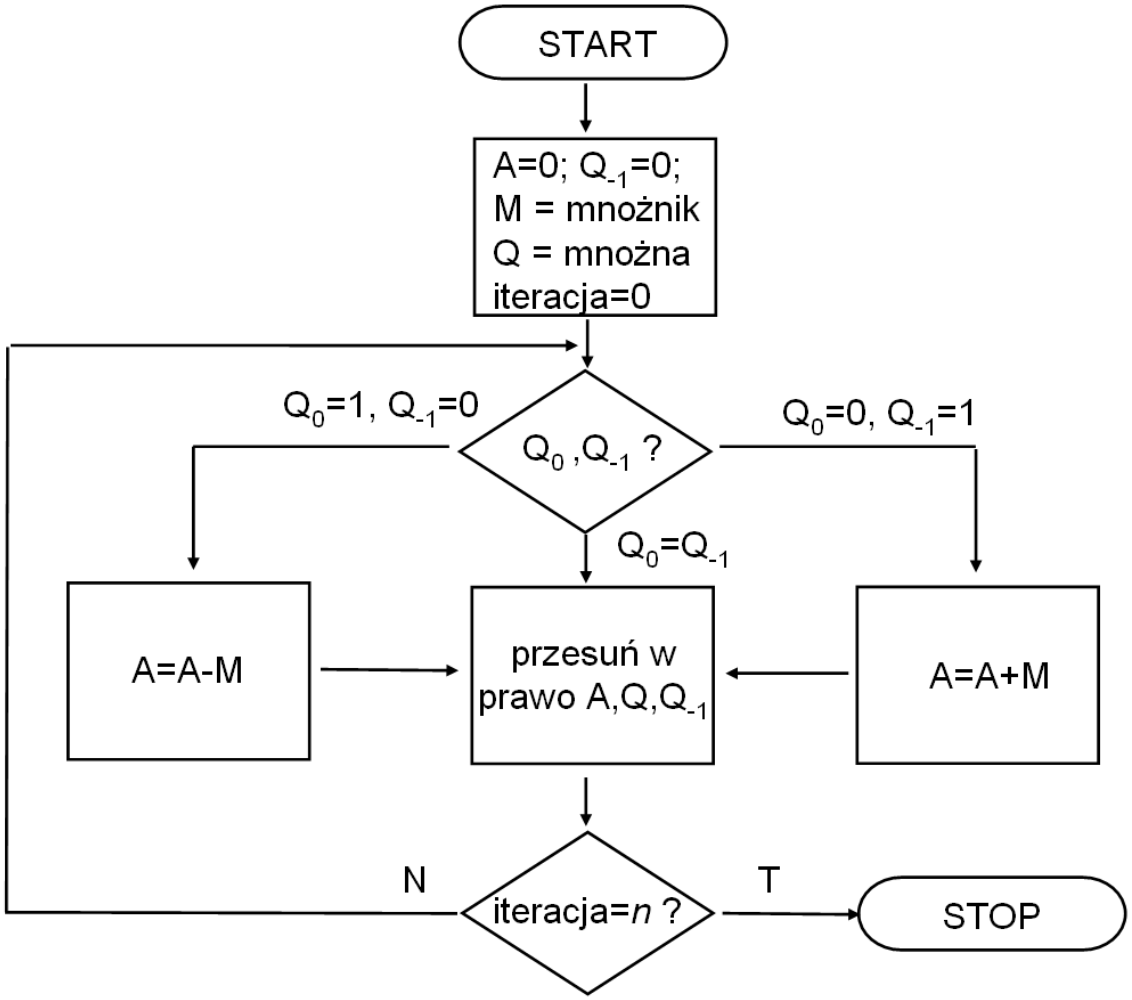


Algorytm Bootha - mnożenie liczb całkowitych ze znakiem

dane: mnożna *Q* i mnożnik *M* - każdy argument ma *n* bitów,

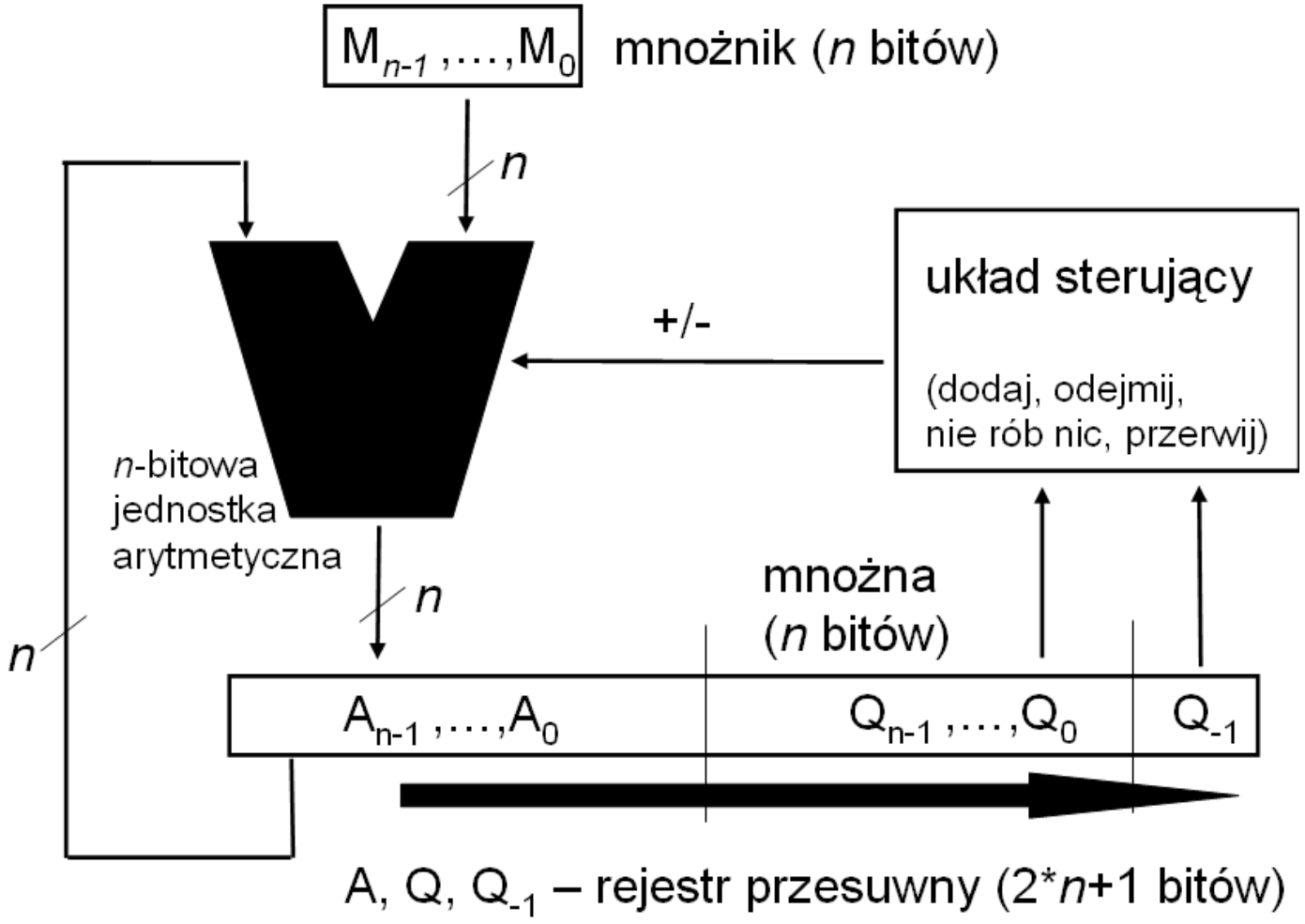
rejestr przesuwny - trzy części: *A* i *Q* po *n* bitów, Q_{n-1} – 1 bit (razem 2*n*+1 bitów),

wynik: po *n* iteracjach w części *A* i *Q* (2*n* bitów).



Algorytm Bootha

Mając już w procesorze ALU z możliwością „+” i „–”, stosunkowo prosto można dobudować układ mnożący ww. algorytmem…



Przykład -3 x 7 (argumenty 4-bitowe)

A Q Q_{-1}

0000 110**1 0** 1 i 0 – odejmij mnożnik

1001 -7

+-------

1001 1101 0

-> SAR – przesunięcie arytmetyczne w prawo!

1100 111**0 1** 0 i 1 – dodaj mnożnik

0111 +7

+-------

0011 1110 1

-> SAR

0001 111**1 0**

1001 -7

+-------

1010 1111 0

-> SAR

1101 011**1 1** ostatnie bity jednakowe

-> SAR

1110 1011 (-21)

Restoring division - dzielenie liczb całkowitych bez znaku

dane: *a* - dzielna, *b* – dzielnik,
rejestr przesuwny: dwie części *A* i *Q* po *n* bitów każda
wynik: reszta *r* z dzielenia (modulo) w *A*, iloraz *q* w *Q*

wykonywane działanie:

$$\frac{a}{b} = q + \frac{r}{b}$$

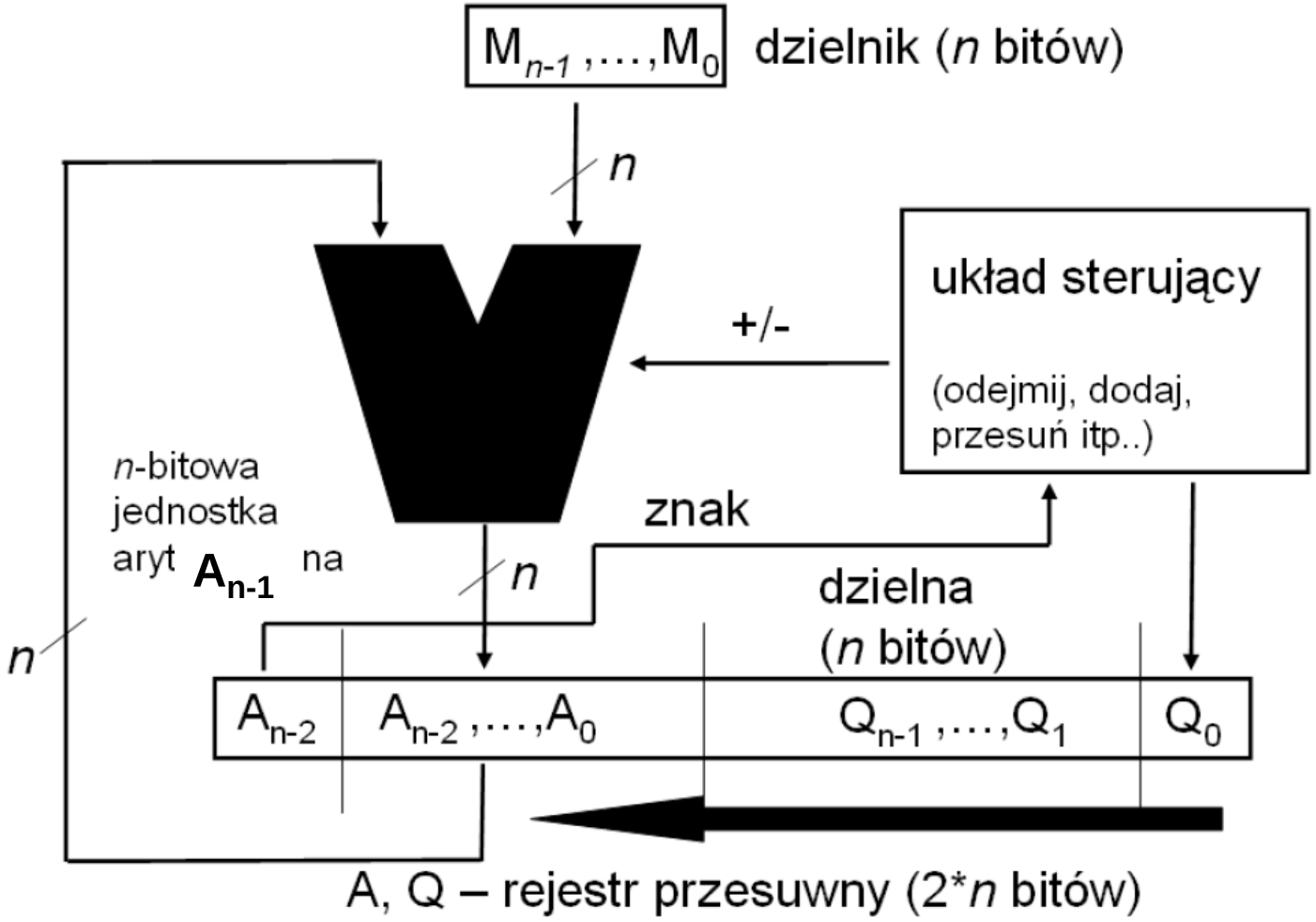
inicjowanie zmiennych:
M = *b* (dzielnik, *n* bitów)
A = 0 (akumulator, *n* bitów)
Q = *a* (dzielna, *n* bitów)*

```
for (i=0; i<n; i++){
    przesuń A i Q o 1 bit w lewo
    oblicz A:=A-M
    jeśli A<0 to Q0=0 i przywróć (restore)
    poprzednią wartość A (A:=A+M),
    w przeciwnym razie ustaw Q0=1.
}
```

*dzielna może mieć 2*n* bitów, zajmuje wtedy cały rejestr przesuwny (A i Q). Wynik (iloraz) takiej operacji musi jednak mieścić się w *n* bitach – w przeciwnym wypadku wynik będzie błędny.

Restoring division - dzielenie liczb całkowitych bez znaku

podobnie jak w przypadku a. Bootha, mając w procesorze zwykłe ALU, stosunkowo niedużym kosztem można dobudować prosty układ dzielący.



Przykład 11 / 3 (argumenty 4-bitowe)

A Q

0000 1011 A=0 Q=11

<- shl - 1. iteracja

0001 0110

1101 (-3)

+-------

1110 (<0 – przywróć poprzednią wartość, Q_0=0)

0001 011**0**

<- shl - 2. iteracja

0010 11**0**0

1101 (-3)

+-------

1111 (<0 – przywróć poprzednią wartość, Q_0=0)

0010 11**00**

<- shl - 3. iteracja

0101 1**00**0

1101 (-3)

+-------

0010 (>0 – pozostaw wynik obliczeń, Q_0=1)

0010 1**001**

<- shl - 4. iteracja,

0101 **001**0

1101 (-3)

+-------

0010 (>0 – pozostaw i Q_0=1)

0010 **0011**

A=2 reszta

Q=3 iloraz

Mnożenie i dzielenie liczb całkowitych we współczesnych procesorach

- W przeciwieństwie do **+** i **–** procesory posiadają dedykowane instrukcje (i algorytmy) do ***** i **/** liczb ze znakiem i bez znaku.

- Większość współcześnie projektowanych/produkowanych procesorów (również sygnałowych, graficznych i mikrokontrolerów) pozwala wykonać mnożenie w jednym (góra kilku) cyklu zegarowym.

- Dzielenie jest generalnie operacją złożoną – iteracyjną.
  W przedstawionym (prostym…) algorytmie kolejne etapy (*restore* i wyznaczenie bitów: ilorazu i reszty) zależą od wyniku (znaku) poprzedzającego je odejmowania. Nie można ich zatem (w prosty sposób)* wykonać równolegle (jak w mnożeniu)

- Liczba cykli (mikrooperacji) zależy nie tylko od implementowanego algorytmu, ale również od rozmiaru danych i konkretnych wartości dzielnej i dzielnika. Dodatkowo, liczba cykli * i / może się znacznie różnić między kolejnymi generacjami tej samej architektury (np. x86-64).

*w praktyce, w mikroprocesory często wykonują dzielenie w oparciu o algorytm SRT.

Mnożenie i dzielenie liczb całkowitych we współczesnych procesorach

Liczba cykli zegarowych, potrzebnych do wykonania * i /
w 32-bitowym mikrokontrolerze ARM-Cortex M4:

Multiply	Multiply	MUL Rd, Rn, Rm	1	
	Multiply accumulate	MLA Rd, Rn, Rm	1	
	Multiply subtract	MLS Rd, Rn, Rm	1	
	Long signed	SMULL RdLo, RdHi, Rn, Rm	1	
	Long unsigned	UMULL RdLo, RdHi, Rn, Rm	1	
	Long signed accumulate	SMLAL RdLo, RdHi, Rn, Rm	1	
	Long unsigned accumulate	UMLAL RdLo, RdHi, Rn, Rm	1	
Divide	Signed	SDIV Rd, Rn, Rm	2 to 12	Division operations terminate when the divide calculation completes, with the number of cycles required dependent on the values of the input operands. Division operations are interruptible, meaning that an operation can be abandoned when an interrupt occurs, with worst case latency of one cycle, and restarted when the interrupt completes.
	Unsigned	UDIV Rd, Rn, Rm	2 to 12	

Zestawienia liczby mikrooperacji, niezbędnych do wykonania rozkazów
w architekturach x86 różnych generacji są dostępne na stronie:

https://www.agner.org/optimize/instruction_tables.pdf