CURS 1 — 26.02.2019

Cap I: Aritmetica în sistemele de calcul (Computer Arithmetics)

1.1. Introducere

→ Algoritmul lui Robertson

• operanzi în C_2 și ne interesează să îl explicităm pe B: (biții lui B)

$A * B$

↑ multiplicand ↑ multiplier

$$B = (b_{n-1} b_{n-2} \dots b_i \dots b_1 b_0)_{C_2}$$

$$b_i \in \mathbb{B} = \{0, 1\} \text{ cu } i = \overline{0, n-1}$$

Formula lui Robertson: $$B_{C_2} = \left(-b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i\right)$$

$i = 0$: $b_0 \times A \times 2^0$

$+ \; b_1 \times A \times 2^1$

$+ \; \dots\dots$

$+ \; b_{n-2} \times A \times 2^{n-2}$

$- \; b_{n-1} \times A \times 2^{n-1}$

P

Lucru care se întâmplă în algoritmul lui Robertson

- Fiecare 1 are 2 clocki (shiftare + adunare), fiecare 0 are 1 clock => => greu pt 1111 1111, de ex. => => Booth
- Un pas în acest algoritm: $(b_{i-1} - b_i) * 2^i * A$
- Se consideră $b_{-1} = 0$
- $i = 0$: $(b_{-1} - b_0) \times 2^0 \times A$

  $i = 1$: $+ (b_0 - b_1) \times 2^1 \times A$

  $i = 2$: $+ (b_1 - b_2) \times 2^2 \times A$

  (...)

  $i = j-1$: $+ (b_{j-2} - b_{j-1}) \times 2^{j-1} \times A$

  $i = j$: $+ (b_{j-1} - b_j) \cdot 2^j \times A$

→ Algoritmul lui Booth

b_i	b_{i-1}	Op
0	0	0
0	1	+A
1	0	−A
1	1	0

Scanned by CamScanner

...

$i = n-2: \quad + (b_{n-3} - b_{n-2}) \times 2^{n-2} \times A$

$i = n-1: \quad + (b_{n-2} - b_{n-1}) \times 2^{n-1} \times A$

- Termenii se vor simplifica
- Pentru termenii $i = j$ și $i = j+1$:

$$-b_j \times 2^j + b_j \times 2^{j-1} = b_j \times 2^j (-1+2) = b_j \times 2^j$$

- Practic se ajunge la: $P = \left(\sum_{i=0}^{n-2} b_i \times 2^i - b_{n-1} \times 2^{n-1}\right) \times A$ adică tot formula lui Robertson. Diferența e în faptul că Booth e mai eficient (mai puțini clocki)

- Ex: $B = 0\,1\,0\,1\,0\,1\,0\,1$ => 4 operații Robertson, dar 8 operații pt Booth: $B' = 0\;1\;0\;1\;0\;1\;0\;1\;|\;0$ =>
  $1\;\bar{1}\;1\;\bar{1}\;1\;\bar{1}\;1\;\bar{1}$

  => Booth modificat

- Ex: $B = 1111\;0000\;|\;0$ – Robertson = 4
  $B' = 000\bar{1}\;0000$ – Booth = 1

→ Algoritmul lui Booth modificat

- combină Robertson și Booth, verifică cazurile de 1 sau 0 izolat

b_{i+1}	b_i	F	B''	F'	
0	0	0	0	0	
0	0	1	1	0	← cazul de 01 de la Booth (+)
0	1	0	1	0	← 1 izolat
0	1	1	0	1	← nu știm ce e după 0, dar acum încă suntem în șir de 1
1	0	0	0	0	
1	0	1	$\bar{1}$	1	
1	1	0	$\bar{1}$	1	
1	1	1	0	1	

F = statutul șirului din care vin

F' = statutul șirului modificat

B'' = referă la ce să facem

- Avem nevoie de un bit în plus la stânga și: $b_n = b_{n-1}$ (e egal cu c. m. semnificativ)

Scanned by CamScanner

- Tentru 11 : $-A \times 2^i + A \times 2^{i+1} = A \times 2^i(-1+2) = A \times 2^i \Rightarrow (+)$
  (10, 01)

- Tentru 11 : $+A \times 2^i - A \times 2^{i+1} = -A \times 2^i \Rightarrow (-)$
  (01, 10)

- Exemple:

> B = 0 0 1 0 1 0 1 0 1 0 — Robertson = 4
> B' = 1 1 1 1 1 1 1 1 — Booth = 8
> B'' = 0 1 0 1 0 1 0 1 — Booth mod. = 4
> F = 0 0 0 0 0 0 0 0 0

> B = 1 1 1 1 1 0 0 0 0 0 — Robertson = 4
> B' = 0 0 0 1 0 0 0 0 — Booth = 1
> B'' = 0 0 0 1 0 0 0 0 — Booth mod. = 1
> F = 1 1 1 1 0 0 0 0 0 0

- Exercitiu pt aplicarea lui Booth modificat:

> X = -105
> Y = -73
> => P = +7665 (trb. să dea)

overflow

COUNT	OVF	A	Q[8]	Q	F	M
000	0	0000 0000	(1)	1001 0111	0	1011 0111
	+	0100 1001			1	
	0	0100 1001				
		0010 0100	1	1100 1011		
001	0	0001 0010	0	1110 0101	1	
010	0	0000 1001	0	0111 0010	1	
011	+	0100 1001			1	
	0	0101 0010				
		0010 1001	0	0011 1001		
100	0	0001 0100	1	0001 1100	1	
101	+	1011 0111			0	
		1100 1011				
		1100 1011	1	1000 1110		
110	0	1111 0010	1	1100 0111	0	
111	+	0100 1001			1	
	0	0011 1011				
		0001 1101	1	1110 0011		

Scanned by CamScanner

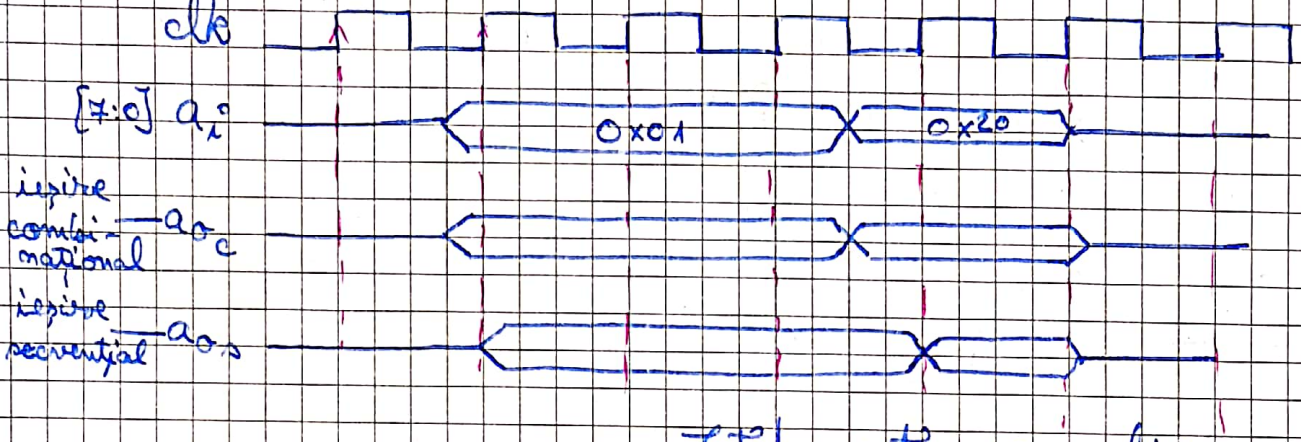
$X = 1\,1\,1\,0\,1\,0\,0\,1_{SM} = \boxed{1\,0\,0\,1\,0\,1\,1\,1}_{C_2}$

$Y = 1\,1\,0\,0\,1\,0\,0\,1_{SM} = \boxed{1\,0\,1\,1\,0\,1\,1\,1}_{C_2}$

$M = 1\,0\,1\,1\,0\,1\,1\,1 \Rightarrow -M = 0\,1\,0\,0\,1\,0\,0\,1$

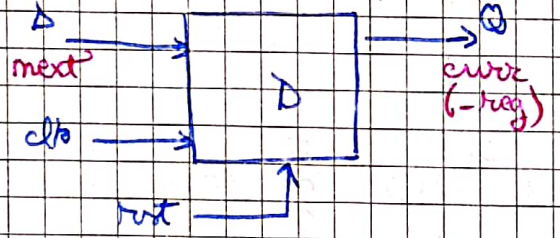
LAB 2

- Circuite combinaționale vs. secvențiale



- Circuit secvențial < Latch: activ pe palier
  Flip flop: activ pe front.

- Exemplu f.f: D ff



  - stare curentă: Q (cea stabilă, care menține semnalul pt. măcar un clock)
  - stare următoare: D

- Secvențial: always @ (posedge clk, posedge rst)
  begin
  reg <= next;
  end

- Arhitectură cu ——— unitate de înmulțire
  ——— // ——— împărțire
  ——— // ——— memorie

Scanned by CamScanner