Lucrarea 8

Aspecte Algoritmice ale Înmulțitoarelor Binare

Această lucrare prezintă o manieră algoritmică de soluționare eficientă a problemei overflow-ului la algoritmul Booth modificat radix-2. În plus, este introdus un cadru general, bazat pe limbajul VHDL, de verificare exhaustivă a soluției propuse.

1. Codificare Booth

Algoritmul lui Booth este o metodă clasică de înmulțire a numerelor în virgulă fixă reprezentate în complement de 2. Ea se bazează pe codificarea lui Booth, care face uz de pattern-uri specifice înmulțitorului. Pattern-urile dorite sunt cele ce includ şiruri contigue de 0-uri sau 1-uri. Oricum, admițând faptul că fiecare pas al algoritmului presupune o operație de shiftare, precedată sau nu de o operație aritmetică elementară (adunare sau scădere), putem observa că pentru anumite pattern-uri ale înmulțitorului – cum ar fi …010101…– overheadul computational dictat de operațiile aritmetice elementare vor fi mai mari pentru algoritmul lui Booth, prin comparație cu o soluție mai directă cum ar fi algoritmul lui Robertson. Acest lucru se întâmplă deoarece, în pattern-ul de mai sus nu avem un şir contiguu de 0-uri sau 1-uri. În Tabela 1 este prezentată o reprezentare canonică a codificării Booth. În plus, în Figura #8.1 sunt prezentate două exemple de înmulțitoare: în cazul (a) algoritmul lui Booth este mai bun decât Robertson, în schimb, cazul (b) descrie situația opusă.

x_{i+1}	x_i	Cod	x_i-x_{i+1}	Descriere
0	0	0	0	Nici o operație aritmetică
0	1	1	1	+ Deînmulțit
1	0	$\overline{1}$	-1	– Deînmulțit
1	1	0	0	Nici o operație aritmetică

Tabela #8.1 Codificare canonică Booth: x_{i+1} şi x_i sunt doi biți consecutivi ai înmulțitorului x_{n-1} x_{n-2} … x_1 x_0 x_{-1}, unde x_{-1}=0. Aici, bitul curent al înmulțitorului este x_{i+1}.

1.1 Algoritmul lui Booth Modificat

Deoarece algoritmul lui Booth nu reuşeşte să înbunătățească overhead-ul computațional revendicat de mult mai simplul algoritm al lui Robertson, atunci când sunt întâlnite pattern-uri ale înmulțitorului de forma …010101… , după cum se arată în Figura #8.1, vor fi necesare rafinări subsecvente ale algoritmului. Prin urmare, algoritmul lui Booth modificat acordă o semnificație deosebită cazurilor de '0' izolat şi '1' izolat.

$x = 11000111\vdots 0$ ← deînmulțit → $x = 11010101\vdots 0$

$x' = 0\,\overline{1}\,00100\,\overline{1}$ ← codificare canonică → $x' = 0\,\overline{1}\,1\,\overline{1}\,1\,\overline{1}\,1\,\overline{1}$

a) Booth este mai bun b) Robertson este mai bun

Figura #8.1 Doi înmulțitori exemplu şi codificarea lor canonică: numărul de operații aritmetice elementare revendicate de algoritmul lui Robertson este egal cu numarul de cifre '1' în reprezentarea C2 a înmulțitorului x, în timp ce algoritmul lui Booth revendică un număr de operatii aritmetice elementare egal cu numărul de 1-uri sau $\overline{1}$-uri în x'. Prin urmare, sunt 5 operații pentru Robertson, 3 pentru Booth în cazul (a), şi 5 pentru Robertson, 7 pentru Booth în cazul (b).

În acest moment, observația esențială este că atunci când x_i este bitul curent al înmulțitorului, gardat fiind de alți doi biți (x_{i+1} x_{i-1}) şi

formând astfel pattern-uri de forma '010' şi '101', avem cazul special de 1 şi 0 izolat. Aceasta înseamnă că, presupunând codificarea Booth, efectul per total al acţiunilor dictate de '010' şi '101' este cel descris de Ecuaţia 8.1.

$$[(x_i - x_{i-1}) \times 2^i + (x_{i+1} - x_i) \times 2^{i+1}] \times Multiplicand \qquad (8.1)$$

Prin urmare, '101' – situaţia de 0 izolat – va revendica $(1 \times 2^i - 1 \times 2^{i+1}) \times Multiplicand = -2^i \times Multiplicand$ şi 010 – adică 1 izolat – revendică $(-1 \times 2^i + 1 \times 2^i) \times Multiplicand = 2^i \times Multiplicand$, ceea ce înseamnă că este extrem de simplu de a acţiona în astfel de situaţii.

Celelalte situaţii sunt tratate pur şi simplu prin aplicarea proprietăţilor moştenite de la algoritmul lui Booth simplu. În algoritmul lui Booth simplu o tranziţie de la 0 la 1 atunci când se parcurge înmulţitorul (în notaţie poziţională) va avea $\overline{1}$ pe post de corespondent canonic, în timp ce tranziţia de la 1 la 0 va avea drept coresponent canonic 1. În algoritmul lui Booth modificat pentru a trata o tranziţie în acelaşi mod în care a fost tratată în cazul algoritmului lui Booth simplu, tranziţia trebuie să apară de la 0 la un şir contiguu de 1-uri (2 sau mai multe cifre '1') sau de la 1 la un şir contiguu de 0-uri. O altă proprietate moştenită de la algoritmul lui Booth simplu este următoarea: atunci când pasul curent al algoritmului are bitul curent al înmulţitorului într-un şir de 1-uri sau de 0-uri, acel pas va consta într-o simplă operaţie de shiftare.

În continuare, trebuie găsită o metodă de detectare a situaţiilor mai sus menţionate. În acest scop este introdus flag-ul *F*. Acest *F* semnalează, pentru bitul curent x_i dacă grupul binar din imediata vecinătate din stânga este un şir de 0-uri (*F*=0) sau un şir de 1-uri (*F*=1), excluzând aici cazurile de 0 şi 1 izolat. *F* este 0 atunci când porneşte algoritmul şi se poate schimba în 1 numai dacă se întâlneşte un şir de 1-uri, sau se poate reveni la valoarea 0 atunci când se întâlneşte un şir de 0-uri. Atunci când luăm în consideraţie bitul curent x_i, bitul din vecinătatea dreaptă x_{i+1} şi flagul *F*, toate situaţiile posibile sunt rezumate în Tabela #8.2.

Din Tabela #8.2 putem obţine expresia booleană a lui *F'*, presentată în Ecuaţia #8.2.

$$F' = F \cdot x_{i+1} + F \cdot x_i + x_{i+1}x_i = F \cdot (x_{i+1} + x_i) + x_{i+1} \cdot x_i \qquad (8.2)$$

F' este flagul ce va fi folosit în pasul următor (iterația următoare) al algoritmului, şi *x"* este codificarea canonică a operației ce trebuie întreprinsă în pasul curent.

x_{i+1}	x_i	F	x''	F'	Interpretarea
0	0	0	0	0	Interiorul unui şir de 0-uri
0	0	1	1	0	Începutul unui şir de 0-uri
0	1	0	1	0	Un 1 izolat
0	1	1	0	1	Interiorul unui şir de 1-uri
1	0	0	0	0	Interiorul unui şir de 0-uri
1	0	1	$\overline{1}$	1	Un 0 izolat
1	1	0	$\overline{1}$	1	Începutul unui şir de 1-uri
1	1	1	0	1	Interiorul unui şir de 1-uri

Tabela #8.2 Codificarea canonică Booth modificat, *x"*, având flagul curent *F* şi flagul pasului următor *F'*. Bitul curent al înmulțitorului este aici x_i.

$x = 1\vdots 11000111$ ← înmulțitor → $x = 1\vdots 11010101$

$x'' = 0\overline{1}00\overline{1}00\overline{1}$ ← codificare canonică → $x'' = 0\overline{1}010101$

$F = 10001110$ $\qquad$ $F = 10000000$

a) Booth modificat este mai bun $\qquad$ b) Booth modificat este mai bun

Figura #8.2. Înmulțitorii exemplu din Figura 1. Algoritmul lui Booth modificat revendică 3 operații aritmetice elementare (a) şi 4 operații aritmetice elementare (b). În ambele cazuri, algoritmul Booth modificat este mai bun.

1.2 Problema Overflow-ului

După cum se arată în Figura #8.2, algoritmul lui Booth modificat este o metodă foarte eficace de înmulțire. De fapt, prin aportul tehnicilor de optimizare hardware ce pot fi aplicate, acest algoritm este piatra de temelie a celop mai performante dispozitive de înmulțire. Cu toate acestea, algoritmul lui Booth modificat se confruntă cu problema

calculului în prezența overflow-ului, chiar dacă acest aspect a fost soluționat implicit în cazul algoritmului Booth modificat.

Atunci când vorbim de problema overflow-ului, vom face referire la erorile care apar la adunarea a doi operanzi în complement de 2, din cauza dimensiunii limitate a reprezentării. Această problemă este prezentată în Tabela 3. Situațiile de overflow sunt relativ simplu de detectat, deoarece ele rezultă din adunarea a două numere negative ce dă rezultat pozitiv sau din adunarea a două numere pozitive ce dă rezultat negativ.

Algoritmul de înmulțire procedează la adunarea unui număr la produsul parțial în fiecare pas algoritmic. Să presupunem că produsul parțial este reprezentat pe n biți şi numărul ce trebuie adunat la acest produs parțial este de asemenea reprezentat pe n biți. Atunci, dacă rezultatul este şi el reprezentat pe n biți, poate să apară o situație de overflow, în schimb dacă rezultatul este reprezentat pe $n+1$ biți, atunci problema overflow-ului este completamente evitată. Fără îndoială, aceasta este o soluție directă şi simplistă pentru problema overflow-ului, iar soluționarea sa se poate face printr-o soluție mai "deşteaptă", astfel încât să rezulte o implementare mai rapidă şi mai ieftină.

Atunci când se efectuează algoritmul lui Booth modificat radix-2, la fiecare pas, fie că avem o operație aritmetică elementară sau nu, va fi necesară efectuarea unei shiftări la dreapta a produsului parțial. Astfel, în vederea atingerii scopului nostru, problema overflow-ului poate fi redusă la urmatoarea întrebare: *atunci când se shiftează produsul parțial, care este valoarea noului bit care apare pe poziția cea mai semnificativă – eliberată prin shiftare – a reprezentării binare a produsului parțial*?

Problema formulată mai sus este soluționată dacă vom sti când trebuie introdus un '1' pe cea mai semnificativă poziție a produsului parțial shiftat, deoarece în toate celelalte situații vom introduce un '0'. Bitul introdus la shiftarea produsului parțial va fi '1' în două situații: atunci când nu avem overflow dar produsul parțial înainte de shiftare a rezultat ca negativ, sau atunci când apare overflow la adunarea unui produs parțial negativ cu un deînmulțit negativ iar rezultatul noului produs parțial înainte de shiftare este pozitiv. Aceste două situații sunt prezentate în Figura #8.3, fără a pierde din generalitate, prin intermediul a două exemple.

Aceasta înseamnă că, dacă stocăm cea mai semnificativă parte a produsului parțial în registrul A (jumătatea mai puțin semnificativă este stocată în registrul Q) de dimensiune n biți, atunci cea mai semnificativă

poziție pre-shiftare a produsului parțial este $A(n-1)$, cea mai semnificativă poziție post-shiftare a produsului parțial este $A'(n-1)$, şi relația dintre cele două este prezentată în Ecuația #8.3, unde OVR indică prezența overflow-ului.

$$A'(n-1) = A(n-1)\cdot\overline{OVR} + \overline{A(n-1)}\cdot OVR = A(n-1) \oplus OVR \qquad (8.3)$$

x_{n-1}	y_{n-1}	c_{n-1}	z_{n-1}	$c_n = c_{out}$	Flag de overflow
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	1	0
1	1	0	0	1	1
1	1	1	1	1	1

Tabela #8.3 Flag-ul semnalizează apariția overflow-ului pentru adunarea lui $X = x_{n-1}x_{n-2}\ldots x_1x_0$ cu $Y = y_{n-1}y_{n-2}\ldots y_1y_0$ şi obținerea lui $Z = z_{n-1}z_{n-2}\ldots z_1z_0$ ca rezultat ($x_i, y_i, z_i \in \{0,1\}; i = 0..n-1$). Lanțul de carry pentru această adunare este $C = c_nc_{n-1}c_{n-2}\ldots c_1c_0$, cu c_0 reprezentându-l pe carry in iar c_n pe carry out.

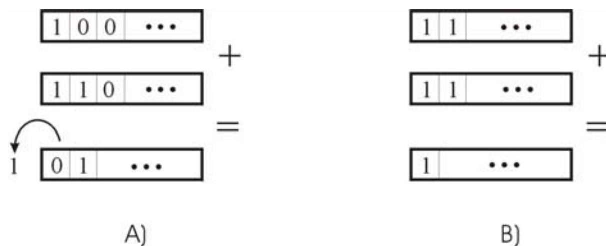


Figura 8.3 Situațiile în care *A'*(*n*-1)=1: A) situația cu overflow şi B) situația fără overflow.

Chiar dacă Ecuația #8.3 pare a fi foarte simplă şi eficientă în acelaşi timp, de fapt implementarea se va confrunta cu o penalitate de

performanță datorată faptului că trebuie să așteptăm efectuarea operației aritmetce elementare cu produsul parțial, pentru a putea genera flag-ul de overflow. Astfel, valoarea lui*A'*(*n*-1) va fi cunoscută după ce efectuarea operației aritmetice elementare – corespunzătoare pasului curent al algoritmului – se va fi terminat. Explicația constă, de fapt, în expresia lui *OVR*, care este prezentată în Ecuația #8.4.

$$OVR = c_n \oplus c_{n-1} \quad (8.4)$$

În Ecuația 2.4, c_n și c_{n-1} sunt biții din lanțul de carry ce au fost prezentați în Tabela #8.3. Pe cale de consecință, va fi util să exprimăm *A'*(*n*-1) în alți termeni, astfel încât expresia sa să depindă de variabile ce pot fi obținute mult mai rapidprin implementarea hardware. Următoarea secțiune arată cum se poate realiza acest obiectiv.

2. Detalii Algoritmice

Expresia lui *A'*(*n*-1) din Ecuația #8.3, pentru o mai bună interpretare, poate fi scrisă precum în Ecuația #8.5.

$$\begin{aligned} A'(n-1) = {} & A(n-1)\cdot\overline{OVR}\cdot M(n-1) + A(n-1)\cdot\overline{OVR}\cdot\overline{M(n-1)} + \\ & + \overline{A(n-1)}\cdot OVR\cdot M(n-1) + \overline{A(n-1)}\cdot OVR\cdot\overline{M(n-1)} \end{aligned} \quad (8.5)$$

În Ecuația #8.5 avem 4 mintermi, notați ca α, β, χ, δ. Prin urmare, avem expresiile $\alpha = A(n-1)\cdot\overline{OVR}\cdot M(n-1)$, $\beta = A(n-1)\cdot\overline{OVR}\cdot\overline{M(n-1)}$, $\chi = \overline{A(n-1)}\cdot OVR\cdot M(n-1)$, $\delta = \overline{A(n-1)}\cdot OVR\cdot\overline{M(n-1)}$ iar Ecuația #8.5 foate fi rescrisă în mod direct ca:

$$A'(n-1) = \alpha + \beta + \chi + \delta \quad (8.6)$$

De prima dată, vom încerca rescrierea termenilor α, β, χ, δ, astfel încât variabila *A*(*n*-1) să fie eliminată. Pentru a face acest lucru, va trebui să clarificăm două aspecte extrem de importante ce implică intrpretarea flagului *F* și evoluția valorii produselor parțiale. Mai mult, acest studiu va arăta cum tehnica "running over zeros" ne simplifică întregul demers.

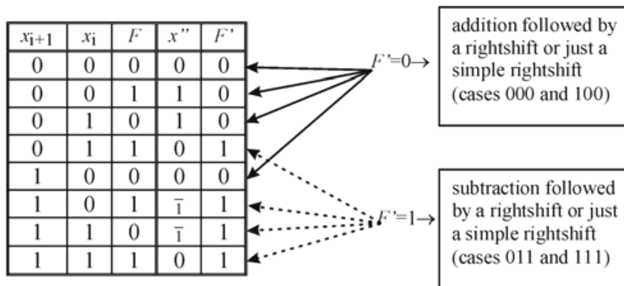
2.1 Interpretarea flagului *F*

Pentru a putea analiza implicațiile valorii lui *F* în contextul algoritmului lui Booth modificat, va fi necesară citirea Tabelei #8.2 în maniera indicată de Figura #8.4. În Figura #8.4 se prezintă faptul că, dacă în pasul curent al algoritmului avem adunare sau scădere, atunci acest lucru este indicat de asemenea prin valoarea lui *F'*. Această concluzie vine ca o consecință directă a modului în care interpretăm Tabela #8.2.

În următoarele noastre considerații, pentru a simplifica modul în care privim un pas al algoritmului, fără însă a pierde din generalitate, vom folosi următoarea taxonomie: un pas al algoritmului lui Booth modificat este un:

- *pas aritmetic* (trebuie efectuată o adunare sau scădere, urmată de o shiftare la dreapta)
- *pas simplu* (se efectuează doar shiftarea la dreapta).

De asemenea, pentu o mai bună înțelegere şi interpretare, vom lua în considerație ambele cazuri *F'*=0 şi *F'*=1 respectiv.



x_{i+1}	x_i	F	x''	F'
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	$\bar{1}$	1
1	1	0	$\bar{1}$	1
1	1	1	0	1

Figure 8.4. Valoarea lui *F'*, pe lângă semnificația ei inițială, ne dă o indicație despre natura pasului curent al algoritmului. Acest tip de interpretare este dat de cele două cutii din dreapta figurii (pentru *F'*=0 şi *F'*=1).

- ***F'*=1**

  În această situație putem avea un pas curent de tip aritmetic sau un pas simplu. Pasul aritmetic va implica întotdeauna o scădere. Mai mult, dacă pasul curent este un pas simplu, atunci el vine după un alt pas simplu sau după un pas ce implică operația aritmetică de scădere. Ambele cazuri în care pasul curent este un pas simplu şi *F'*=1 sunt cazuri de tipul "în interiorul unui şir de

1-uri". Deoarece algoritmul porneşte cu *F*=0, întotdeauna va fi un '0' la dreapta şirului nostru de 1-uri. Prin urmare, imediat înaintea paşilor dictaţi de situaţia "în interiorul unui şir de 1-uri" va fi întotdeauna un pas aritmetic de scădere (intrarea 110 în Tabela #8.2).
Concluzia este că, atunci când *F'*=1, *pasul curent este un pas aritmetic de scădere sau un pas simplu care vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de scădere.*

- ***F'*=0**
  În această situaţie avem concluzii similare, până la un punct extrem de important. În mod analog cu cazul precedent, aici pasul curent este fie un pas aritmetic de adunare fie un pas simplu. De asemenea prin analogie cu cazul precedent, putem spune că pasul simplu curent atunci când *F'*=0, se datorează unuia din cazurile "în interiorul unui şir de 0-uri". Prin urmare, dacă avem un '1' înaintea unui şir de 0-uri, putem spune că pasul simplu curent vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de adunare. Din nefericire, există un caz în care nu există un '1' înaintea şirului nostru curent de 0-uri, deoarece iniţial în algoritmul lui Booth modificat avem *F*=0: cazul în care înmulţitorul *X* începe (din dreapta) cu un şir de 0-uri.
  Această situaţie de excepţie poate fi înlăturată cu uşurinţă prin aplicarea tehnicii "running over zeros". Astfel, concluzia pentru *F'*=0 este: *pasul curent este un pas aritmetic de adunare sau un pas simplu ce vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de adunare numai dacă se aplică tehnica "running over zeros".*

2.2 Valoarea Produselor Parţiale

Înrebarea care-şi are răspunsul în această secţiune este următoarea: care sunt valorile maximă şi minimă a produselor parţiale ce apar pe parcursul execuţiei algoritmului lui Booth modificat? Oricum, pentru scopurile statuate în secţinile de mai sus, de interes este doar găsirea valorii maxime absolute (fără semn) a produsului parţial, după un număr finit de paşi ai algoritmului.

Lema 8.1 (Cel mai mare produs parțial) Cea mai mare valoare absolută pe care o poate avea produsul parțial, obținut cu algoritmul lui Booth modificat după un număr finit de paşi, este $|p_n| < |m|$, unde m este deînmulțitul şi p_n este produsul parțial după n paşi algoritmici.

Demonstrație

Produsul parțial este obținut prin aplicarea paşilor algoritmici (algorithm steps). De asemenea, produsul parțial este stocat într-un registru cu lungime fixă (vom lua în considerație implementarea secvențială). Vom considera că lungimea acestui registru este n biți, care este de asemenea lungimea operanzilor, deînmulțit şi înmulțitor. La fiecare pas algoritmic se obține un bit al rezultatului.

Pentru a obține valoarea absolută maximă pentru produsul parțial, care este – într-o implementare secvențială – stocată în registrul acumulator, va trebui să efectuăm la fiecare pas algoritmic un pas aritmetic (aşa cum a fost el descris în secțiunea precedentă). Mai mult, trebuie să fie aceiaşi operație aritmetică (adunare sau scădere) pe parcursul execuției algoritmului. Un pas simplu înseamnă efectuarea unei shiftări la dreapta sau, altfel spus – în termenii aritmeticii binare, a unei împărțiri cu 2 a produsului parțial. Astfel, un pas simplu va avea ca efect descreşterea considerabilă a produsului parțial ca valoare absolută. Fără a pierde din generalitate, vom considera că pasul aritmetic constă dintr-o adunare şi un shift la dreapta (rightshift). Evoluția valorii produsului parțial pe parcursul execuției algoritmului este prezentată în Ecuația #8.7.

$$
\begin{array}{ll}
\text{step } 0 & p_0 = 0 \\
\text{step } 1 & p_1 = \dfrac{0+m}{2} = \dfrac{m}{2} \\
\text{step } 2 & p_2 = \dfrac{\frac{m}{2}+m}{2} = \dfrac{3m}{4} \\
\text{step } 3 & p_3 = \dfrac{\frac{3m}{4}+m}{2} = \dfrac{7m}{8} \\
\vdots & \vdots \\
\text{step } n-1 & p_{n-1} = \dfrac{p_{n-2}+m}{2}
\end{array}
\tag{8.7}
$$

Oricum, deoarece $p_{n-2} = (2^{n-2} - 1)m / 2^{n-2}$, apare ca evident faptul că p_{n-1} are expresia din Ecuația #8.8.

$$p_{n-1} = \frac{\frac{(2^{n-2} - 1)m}{2^{n-2}} + m}{2} = \frac{(2^{n-1} - 1)m}{2^{n-1}} \tag{8.8}$$

Prin urmare, datorită faptului că $2^{n-1} - 1 / 2^{n-1}$ pentru un *n* finit, rezultă că $|p_n| < |m|$.

Consecința 8.1 (Semnul produsului parțial) Datorită concluziei Lemei 8.1, într-o manieră directă, rezultă faptul că adunarea deînmulțitului *m*≠0 la produsul parțial, semnul noului produs parțial va fi semnul deînmulțitului; pe când în situația în care se scade deînmulțitul nenul din produsul parțial, semnul noului produs parțial va fi opus semnului deînmulțitului.

3. Rezolvarea Problemei

Rezultatele Consecinței 8.1 ajută în rezolvarea problemei overflow-ului. Pentru a putea aduce o soluție elegantă acestei probleme, trebuie ă acceptăm două precondiții: prima – ce reiese în mod evident – este că trebuie să tratăm separat, ca excepții, cazurile în care cel puțin unul dintre operanzi este zero, si a doua – faptul că aplicăm tehnica "running over zeros" pentru dispozitivul de înmulțire.

În continuare, vom lua Ecuația #8.6 şi o vom analiza termen cu termen. Fiecare termen trebuie interpretat în aşa fel încât să utilizăm concluziile secțiunilor 2.1 şi 2.2.

- $\alpha = A(n-1) \cdot \overline{OVR} \cdot M(n-1)$

  Acest termen are următoarea semnificație:
  - nu există overflow
  - produsul parțial, înainte de rightshift, este negativ
  - deînmulțitul *m* este negativ

  Astfel, datorită consecinței 8.1, rezultă faptul că pasul curent este *un pas aritmetic de adunare* sau *un pas simplu care vine după ce cel mai*

recent pas aritmetic a fost un pas aritmetic de adunare. Prin urmare, având în minte considerațiile din secțiunea 2.1, α poate fi rescris ca:

$$\alpha = \overline{F'} \cdot \overline{OVR} \cdot M(n-1) \qquad (8.9)$$

- $\beta = A(n-1) \cdot \overline{OVR} \cdot \overline{M(n-1)}$

  Acest termen are următoarea semnificație:
  - nu există overflow
  - produsul parțial, înainte de rightshift, este negativ
  - deînmulțitul *m* este pozitiv

  Datorită consecinței 8.1, pasul curent este fie *un pas aritmetic de scădere* sau *un pas simplu care vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de scădere.* Astfel, ca o consecință a concluziilor secțiunii 2.1, β poate fi rescris ca:

$$\beta = F' \cdot \overline{OVR} \cdot \overline{M(n-1)} \qquad (8.10)$$

- $\chi = \overline{A(n-1)} \cdot OVR \cdot M(n-1)$

  Acest termen are următoarea semnificație:
  - există overflow
  - produsul parțial, înainte de rightshift, este pozitiv şi incorect din cauza overflow-ului
  - deînmulțitul *m* este negativ

  Datorită consecinței 8.1 şi a faptului că overflow-ul schimbă semnul produsului parțial, rezultă că pasul curent este *un pas aritmetic de adunare* sau *un pas simplu care vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de adunare*. Astfel, χ poate fi rescris ca:

$$\chi = \overline{F'} \cdot OVR \cdot \overline{M(n-1)} \qquad (8.11)$$

- $\delta = \overline{A(n-1)} \cdot OVR \cdot \overline{M(n-1)}$

  Acest termen are următoarea semnificație:
  - există overflow
  - produsul parțial, înainte de rightshift, este pozitiv datorită overflow-ului; în mod normal produsul parțial ar trebui să fie negativ
  - deînmulțitul *m* este pozitiv

Aici, datorită consecinței 8.1 şi a prezenței overflow-ului, rezultă faptul că pasul curent este *un pas aritmetic de scădere* sau *un pas simplu care vine după ce cel mai recent pas aritmetic a fost un pas aritmetic de scădere*. Prin urmare, luând în considerație concluziile secțiunii 2.1, δ este rescris ca în Ecuația 8.12.

$$\delta = F' \cdot OVR \cdot \overline{M(n-1)} \qquad (8.12)$$

Soluția, presupunând că $m \neq 0$ şi aplicarea tehnicii “running over zeros”, atunci când luăm în considerație Ecuația (8.6) şi Ecuațiile (8.9) ÷ (8.12) este directă:

$$\begin{aligned}
A'(n-1) &= \overline{F'} \cdot \overline{OVR} \cdot M(n-1) + F' \cdot \overline{OVR} \cdot \overline{M(n-1)} + \\
&+ \overline{F'} \cdot OVR \cdot \overline{M(n-1)} + F' \cdot OVR \cdot \overline{M(n-1)} = \\
&= \overline{OVR} \cdot (\overline{F'} \cdot M(n-1) + F' \cdot \overline{M(n-1)}) + \\
&+ OVR \cdot (\overline{F'} \cdot M(n-1) + F' \cdot \overline{M(n-1)}) = \\
&= \overline{OVR} \cdot (F' \oplus M(n-1)) + OVR \cdot (F' \oplus M(n-1)) = \\
&= F' \oplus M(n-1) \qquad (8.13)
\end{aligned}$$

Calculul în prezența overflow-ului ste astfel soluționată de Ecuația #8.13, prin implicarea variabilelor ce pot fi obținute într-un timp mai scurt, din punct de vedere al implementării hardware.

4. Verificare

Tema 8.1

Se cere verificarea rezultatului obținut în secțiunea 3 a lucrării, printr-o scanare exhaustivă a tuturor combinațiilor de intrare. Procesul de verificare formală se va face în VHDL, cu operanzi pe 4 biți. Rezultatul înmulțirii va fi reprezentat pe 8 biți. De prima dată se va implementa dispozitivul de înmulțire Booth modificat (entitatea booth_impl) într-o manieră comportamentală (arhitectura bi_a a entității booth_impl), după cum se arată în Figura #8.5.

```
entity booth_impl is
        port(x,y:in operand;p:out result);
end booth_impl;
architecture bi_a of booth_impl is
        begin
          process (x,y)
           begin
                p<=modified_booth(x,y);
           end process;
end bi_a;
```

Figura 8.5 Implementarea comportamntală a înmulțitorului Booth modificat, unde se foloseşte funcția modified_booth(x,y)**.**

```
function modified_booth(x,y:operand)return result is
        begin
        if (one operand is zero) then
                return "00000000";
        else    if it is the case shift_over_zeros (k positions);
                else
                for i in 0 to 3-k  loop
                if (x(1)='0' and x(0)='0' and F='1') or (x(1)='0' and x(0)='1'
                and F='0') then
                pp:=add_operand(pp,y); F′:='0';
                elsif (x(1)='1' and x(0)='0' and F='1') or (x(1)='1' and x(0)='1'
                and F='0') then
                pp:=sub_operand(pp,y); F′:='1';
                elsif (x(1)='0' and x(0)='0' and F='0') or (x(1)='1' and x(0)='0'
                and F='0') then
                        F′:='0';
                elsif (x(1)='0' and x(0)='1' and F='1') or (x(1)='1' and x(0)='1'
                and F='1') then
                        F′:='1';
                end if;
                w:=y(3) xor fn;
                pp&x:= shift_pp(pp&x,w);
                end if;
                return pp&x;
        end if;
        end modified_booth;
```

Figure 8.6 Pseudocod inspirat de limbajul VHDL, ce prezintă noua versiune a algoritmului lui Booth modificat pentru operanzi pe 4 biți, unde pp **este produsul parțial.**

După implementarea dispozitivului de înmulțire, acesta se testează în manieră exhaustivă. De prima dată, se generează toate combinațiile posibile de operanzi pe 4 biți $o_1, o_2 \in \mathbf{B}^4$, unde $\mathbf{B}=\{0,1\}$. Rezultatul va fi $r \in \mathbf{B}^8$. Prin urmare dispozitivul de înmulțire va funcționa după cum se arată în Ecuația 8.14.

$$\text{device}(o_1, o_2) = r \in \mathbf{B}^8 \tag{8.14}$$

Apoi ne va trebui o funcție de conversie vect_to_int(string:bit_string)return integer ce returnează echivalentul număr întreg în zecimal al şirului binar (string $\in \mathbf{B}^n$, $\forall n \in \mathbf{N}$). Cu această funcție, cu o arhitectură de testare VHDL, se crează semnalul correct. Valoarea acestui semnal se asignează după cum se arată în Ecuația 8.15.

$$\text{correct} = \begin{cases} 1 & 1 \ \text{if} \ \text{vect_to_int}(\text{device}(o_1, o_2)) = \text{vect_to_int}(o_1) \times \text{vect_to_int}(o_2); \\ 0 & \text{altminteri;} \end{cases} \tag{8.15}$$

Dacă implementarea descrisă în această lucrare funcționează corect, atunci pentru toate combinațiile posibile de operanzi, dispozitivul simulat trebuie să se comporte în aşa fel încât semnalul correct să rămână pe '1'.