IMPLEMENTAREA ALGORITMILOR DE TIP CORDIC ÎN FPGA

Iacob Petrescu **Dragoş Popescu** **Adrian Petrescu**

iacobp@quadrilogic.ro *dragos@csit-sun.pub.ro* *adrian.petrescu@cs.pub.ro*

Universitatea POLITEHNICA, Bucureşti

Rezumat: Lucrarea explorează posibilităţile de implementare a algoritmilor de tip CORDIC (COordinate Rotation DIgital Computer) cu ajutorul structurilor reconfigurabile de tip FPGA (Field Pragrammable Gate Arrays). Au fost avute în vedere funcţiile trigonometrice: sin, cos, arcsin, arccos, arctg, transformarea coordonatelor polare în coordonate carteziene şi invers. Pentru acestea, au fost dezvoltaţi algoritmi de tip CORDIC. Implementarea în FPGA a avut la bază conceptul de procesor CORDIC, văzut ca un "black box", căruia i se aplică la intrare un argument, pentru a se obţine la ieşire funcţia căutată. Procesoarele CORDIC se plasează în două categorii: iterative şi neiterative. În vederea simulării şi implementării a fost descris în Verilog algoritmul CORDIC iterativ, pentru calculul funcţiilor *sin* şi *cos*. Pentru simularea algoritmului, a fost utilizată aplicaţia ModelSim XE III 6.1 e, iar pentru implementarea algoritmului s-a folosit platforma Xilinx Spartan-3 AN Starter Kit, bazată pe circuitul FPGA- XC3S700AN-FGG484.

Cuvinte cheie: CORDIC, sin, cos, arctg, conversie polar-cartezian, ModelSim XE III 6.1 e, FPGA Xilinx Spartan-3 AN Starter Kit.

Abstract: The paper explores CORDIC type algorithms FPGA implementation possibilities. The reasearch focused on trigonometric functions: sine, cosine, arcsine, arccosine, arctan, polar/cartesian coordinate transforms. For these functions suitable CORDIC algorithms were developed.
The FPGA implementation is based on a CORDIC processor seen as a black box whos input/output is an argument and the desired function, repectively. CORDIC processors fall into two cathegories: iterative and non-iterative. An iterative sine/cosine functions CORDIC algorithm was developed in order to be simulated and implemented.. The Verilog algorithm description was simulated and implemented using tools like:ModelSim XE III 6.1 e and FPGA Xilinx Spartan-3 AN Starter Kit.

Key Words: CORDIC, sine, cosine, atan , polar-cartezian transform, ModelSim XE III 6.1 e, FPGA Xilinx Spartan-3 AN Starter Kit.

1. Introducere

Implementarea în hardware, ca soluţii dedicate, a algoritmilor constituie una dintre tendinţele actuale ce se manifestă în domeniul prelucrării semnalelor. În acest scop, se depun numeroase eforturi pentru elaborarea unor algoritmi eficienţi atât în privinţa creşterii vitezei de execuţie, cât şi a reducerii costurilor. Având în vedere răspândirea extrem de largă a microprocesoarelor, în ultimul sfert de veac, s-a manifestat, cu precădere, o dominaţie a algoritmilor orientaţi-software. Din păcate, algoritmii de prelucrare a semnalelor orientaţi-software nu se pot aplica în cazul soluţiilor de implementare hardware. Printre algoritmii eficienţi, din punctul de vedere al implementării hardware, se regăsesc algoritmii iterativi, bazaţi pe operaţiile de deplasare şi adunare, algoritmi care sunt utilizaţi pentru calculul funcţiilor trigonometrice şi a altor funcţii transcedentale.

Pentru calculul funcţiilor trigonometrice, este folosit algoritmul CORDIC (COordinate Rotation DIgital Computer). Algoritmul asigură pentru fiecare iteraţie o creştere suplimentară a preciziei de un bit. Implementarea funcţiilor trigonometrice are la bază rotirea unui vector, în timp ce alte funcţii, cum ar fi rădăcina pătrată, se implementează pornind de la exprimarea incrementală a funcţiei dorite [1].

Funcţiile trigonometrice *sin*, *cos*, *tg* etc. se regăsesc în numeroase probleme care apar în domenii tehnice cum ar fi: Robotica (predicţia deplasării, calcule privind geometria mediului), Sistemele liniare (control), Procesoarele de semnal (transformări, filtre, aplicaţii finale: Radar) s.a. Tehnicile de calcul al funcţiilor trigonometrice pot avea la bază: dezvoltări în serie Taylor, aproximări polinomiale, tabele asociative, algoritmi de tip CORDIC etc. În comparaţie cu alte metode, CORDIC prezintă o serie de avantaje cum ar fi: prezenţa numai a operaţiilor de adunare şi deplasare, absenţa operaţiei de înmulţire, uşurinţa implementării în hardware reconfigurabil (FPGA), complexitate comparabilă cu cea a operaţiilor de înmulţire sau extragere a rădăcinii pătrate.

Primii algoritmi de tip CORDIC au fost elaboraţi pentru soluţionarea în timp real a problemelor de navigaţie, cu ajutorul calculatoarelor numerice [2], [3], [4]. Aceşti algoritmi şi-au găsit utilizări în procesoarele de semnale radar [5], coprocesorul 8087, calculatorul de buzunar HP-35 [6], robotică ş.a. Pentru calculul transformatelor: Fourier Discretă /Cosinus Discretă [7], Hartley Discretă [8], Chirp-Z [9], cât şi pentru soluţionarea sistemelor lineare [10], a fost propusă implementarea CORDIC a funcţiilor trigonometrice, bazată pe rotirea unui vector. Implementări ale algoritmilor de tip CORDIC în FPGA sunt prezentate în lucrările: [11], [12], [13].

2. Bazele algoritmului *rotator* CORDIC

Evaluarea funcţiilor trigonometrice *sin*, *cos*, *tg*, *arcsin*, *arccos* şi *arctg* se bazează pe operaţia de rotire generalizată a unui vector. Operaţia de rotire se implementează printr-o succesiune de rotiri ale vectorului, cu *unghiuri date*, al căror sens este ales, astfel încât, după un număr de iteraţii, suma algebrică a unghiurilor cu care au fost efectuate rotirile să coincidă exact sau cu o eroare acceptabilă cu unghiul dorit. *Unghiurile date* se bucură de proprietatea că tangentele lor sunt puteri ale lui 2, ceea ce, după cum se va constata în continuare, reduce operaţia de înmulţire la o operaţie de deplasare.

Algoritmul CORDIC stă şi la baza *conversiilor de la coordonatele polare la cele Carteziene şi invers*, *Transformărilor Discrete Fourier* (TDF) şi *Cosinus* (TDC), calculul *modulul unui vector* etc.

Algoritmul pentru rotirea unui vector, cu un unghi arbitrar θ, are un caracter iterativ şi foloseşte numai operaţiile de deplasare şi adunare, ceea ce îl recomandă pentru implementarea lui direct în hardware.

Fie un vector unitar având coordonatele (x_0,y_0) şi acelaşi vector unitar, rotit cu un unghi θ, cu coordonatele (x_1,y_1) ca în figura 1.

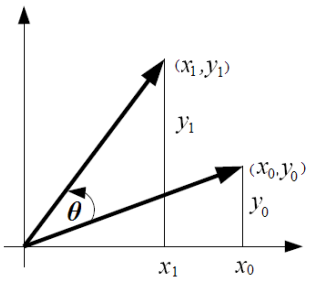

Figura 1. Rotirea unui vector unitar cu un unghi θ

Noile coordonate (x_1,y_1) vor putea fi calculate, în funcţie de vechile coordonate (x_0,y_0) şi de unghiul θ, cu ajutorul expresiilor de mai jos:

$$x_1 = x_0 \,.\, \cos(\theta) - y_0 \,.\, \sin(\theta)$$

$$y_1 = x_0 \,.\, \sin(\theta) + y_0 \,.\, \cos(\theta) \qquad (1)$$

Ecuaţiile 1 pot fi rescrise astfel:

$$x_1 = \cos(\theta).\,[\, x_0 - y_0 \,.\, \mathrm{tg}(\theta)\,]$$

$$y_1 = \cos(\theta).\,[\, y_0 + x_0 \,.\, \mathrm{tg}(\theta)\,] \qquad (2)$$

În cazul în care unghiurile de rotire θ satisfac relaţia:

$$\mathrm{tg}(\theta) = +/- 2^{-i} \qquad (3)$$

înmulţirea cu termenul $\mathrm{tg}(\theta)$ se reduce la o operaţie de deplasare.

Astfel, rotirea cu un unghi arbitrar θ se va reduce la o succesiune de rotiri elementare, cu unghiuri θ_i, al căror sens trebuie stabilit la fiecare iteraţie i.

Întrucât $\cos(\theta_i) = \cos(-\theta_i)$, pe baza relaţei (3), ecuaţiile (2) capătă următorul aspect, în care operaţiile principale sunt adunarea şi deplasarea:

$$x_{i+1} = K_i \,.\, (\, x_i - y_i.(d_i.\, 2^{-i}\,))$$

$$y_{i+1} = K_i \,.\, (\, y_i + x_i.(d_i.\, 2^{-i}\,)) \qquad (4)$$

unde:

$$K_i = \cos(\mathrm{arctg}(2^{-i})) = 1/(1 + 2^{-2i})^{1/2}$$

$$d_i = +/-1 \qquad (5)$$

Făcând abstracţie de constanta de scalare K_i, algoritmul se reduce la o succesiune de operaţii de deplasări şi adunări. Produsul constantelor K_i, care se calculează la fiecare iteraţie, poate fi examinat ca făcând parte din amplificarea sistemului. Pe măsură ce numărul de iteraţii tinde spre infinit, produsul ia valoarea 0,607252935. Astfel, algoritmul de rotire are o amplificare G_n, care depinde de numărul de iteraţii conform expresiei (6).

$$G_n = \prod_{i=1}^{n} (1 + 2^{-2i})^{1/2} \quad (6)$$

Valoarea aproximativă a lui G_n este: 1,646760258121...

Unghiul, rezultat în urma iterațiilor, este definit în mod unic de secvența sensurilor rotirilor elementare, secvență care poate fi reprezentată de componentele unui vector de decizie. Mulțimea tuturor vectorilor de decizie reprezintă un sistem de măsuri unghiulare bazat pe „*arctangente binare*". Valorile unghiulare se pot furniza cu ajutorul unei tabele (Tabelul 1), care are ca intrare numarul *i* al iterației şi ca ieşire arctg(2^{-i}), adică unghiul θ_i în grade:

Tabelul 1. arctg(2^{-i}) ca funcție de *i*

i	arctg(2^{-i}) – **in grade-**
0	45,0
1	26,6
2	14,0
3	7,1
4	3,6
5	1,8
6	0,9
7	0,4
8	0,2
9	0,1

Pentru a roti un vector cu 28° se va efectua următoarea succesiune de rotiri:

$45{,}0^o - 26{,}6^o + 14{,}0^o - 7{,}1^o + 3{,}6^o - 1{,}8^o + 0{,}9^o - 0{,}4^o + 0{,}2^o + 0{,}1^o = 27{,}9^o$

Eroarea înregistrată este de $0{,}1^o$.

Componentele vectorului de decizie, amintit mai sus, se pot asocia cu semnele rotirilor pentru a genera rotirea de un unghi dat. În vederea asigurării unei precizii de *n* biți a rezultatului, trebuie efectuate *n* iterații CORDIC.

Din cele arătate mai sus, se constată că gestiunea unghiului rezultat, pentru rotirea curentă, necesită o resursă hardware *sumator/scăzător/acumulator*, ceea ce conduce la o a treia ecuație, care se asociază celor doua ecuații (4).

$$z_{i+1} = z_i - d_i.\text{arctg}\,(2^{-i}) \quad (7)$$

În cazul în care unghiul este utilizat ca *arctg*, resursa hardware, amintită mai sus, nu mai este necesară.

Practic, *rotatorul* CORDIC poate opera în modurile: *rotire* şi *vector*.

În modul *rotire*, se roteşte un vector dat cu unghiul dorit, care joacă rol de argument, acumulatorul fiind inițializat cu unghiul de rotire dorit. La fiecare iterație, decizia referitoare la sensul rotirii se bazează pe semnul unghiului rezidual, după cum s-a arătat mai sus.

Pentru modul *rotire,* ecuațiile CORDIC sunt următoarele:

$$x_{i+1} = x_i - y_i.\,(d_i.\,2^{-i})$$

$$y_{i+1} = y_i + x_i.\,(d_i.\,2^{-i})$$

$$z_{i+1} = z_i - d_i.\text{arctg}\,(2^{-i}) \quad (8)$$

unde:

$d_i = +1$ daca $z_i < 0$ şi -1 dacă $z_i \geq 0$,

ceea ce conduce la următorul rezultat:

$$x_n = G_n.[x_0.\cos(\theta) - y_0.\sin(\theta)]$$

$$y_n = G_n.[x_0.\sin(\theta) + y_0.\cos(\theta)]$$

$$z_n = 0 \quad (9)$$

În modul *vector*, se roteşte vectorul dat cu unghiul necesar pentru alinierea vectorului cu axa *x*, obţinându-se, în final, un unghi de rotire, cât şi mărimea scalată a vectorului iniţial, respectiv – componenta *x* a rezultatului. Procesul se bazează pe minimizarea componentei *y*, care rezultă la fiecare rotire, semnul acesteia fiind utilizat pentru determinarea sensului următoarei rotiri. Dacă acumulatorul a fost iniţializat cu zero, la sfârşitul procesului, acesta va conţine unghiul căutat. Ecuaţiile utilizate în modul *vector* sunt următoarele:

$$x_{i+1} = x_i - y_i.(d_i. 2^{-i})$$

$$y_{i+1} = y_i + x_i.(d_i.2^{-i})$$

$$z_{i+1} = z_i - d_i.\text{arctg}\ (2^{-i}) \qquad (10)$$

unde:

$d_i = +1$ dacă $y_i < 0$ *şi* -1 dacă $y_i \geq 0$

Pe baza ecuaţiilor (10) se pot scrie relaţiile de mai jos:

$$x_n = G_n.(x^2 + y^2)^{1/2}$$

$$y_n = 0$$

$$\theta_n = \theta + \text{arctg}(x/y) \qquad (11)$$

Algoritmii CORDIC, pentru modurile *rotire* şi *vector*, prezentaţi mai sus, funcţionează pentru un interval de unghiuri cuprins între: $-\pi/2$ şi $+\pi/2$, datorită folosirii la prima iteraţie a valorii 2^0, pentru tangenta. În cazurile unor unghiuri mai mari decât $\pi/2$, este necesară o rotire iniţială cu $(+/-)\pi/2$. Iteraţia de corecţie a fost propusă în lucrarea [3].

3. Implementarea unor funcţii trigonometrice

3.1. Sin şi Cos

Utilizarea algoritmului CORDIC în modul *rotire* conduce la calculul simultan al funcţiilor *sin* şi *cos* de un unghi θ_0, iniţial dat. Prin egalarea cu zero a componentei y_0 a vectorului de intrare şi stabilirea $x_0 = 1/ G_n$, ecuaţiile (9) devin:

$$x_n = \cos(\theta_0)$$

$$y_n = \sin(\theta_0) \qquad (12)$$

Astfel, *rotatorul* CORDIC va genera valorile nescalate ale funcţiilor *sin* şi *cos* de un argument dat θ_0. În general, ieşirile *rotatorului* CORDIC sunt scalate cu factorul de amplificare G_n. . Dacă acest lucru nu este de dorit, *rotatorul* CORDIC poate fi precedat de un înmulţitor cu constanta $1/G_n$. Complexitatea *rotatorului* CORDIC are acelaşi ordin ca şi complexitatea unui singur înmulţitor pentru numere de aceeaşi lungime.

3.2. Arcsin

Calculul valorii funcţiei *arcsin* presupune utilizarea algoritmului în modul *vector*, în condiţiile prezenţei unui vector unitar aliniat la axa *x*, care este rotit astfel încât, componenta *y* să egaleze argumentul de intrare *a*. Valoarea *arcsin* căutată reprezintă unghiul subîntins, care asigură egalitatea între componenta *y* a vectorului rotit şi argumentul de intrare *a.* Componenta d_i a vectorului de decizie rezultă ca urmare a comparaţiei, la fiecare iteraţie *i*, între argumentul de intrare *a* şi componenta curentă y_i.

$$x_{i+1} = x_i - y_i.(d_i. 2^{-i})$$

$$y_{i+1} = y_i + x_i.(d_i.2^{-i})$$

$$z_{i+1} = z_i - d_i.\text{arctg}\ (2^{-i}) \qquad (13)$$

unde:

$d_i = +1$ daca $y_i < a$ *şi* -1 daca $y_i \geq a$

a = argumentul de intrarea.

În urma rotirilor, se obţin următoarele rezultate:

$x_n = ((G_n \,.\, x_0)^2 - a)^{1/2}$

$y_n = a$

$z_n = z_0 - \arcsin(a/G_n.x_0)$ (14)

Procedura de mai sus conduce la rezultate corecte pentru intrări care îndeplinesc condiția:
$-1 < (a/G_n.x_0) < 1$. Pe măsură ce intrarea se apropie de +/-1 (mai exact> |0,98 |) eroarea creşte rapid datorită amplificării *rotatorului*, care nu asigură luarea unor decizii corecte pentru unghiuri apropiate de axa *y*, întrucât vectorul rotit este mai mic decât argumentul de intrare. O soluție pentru această problemă este dată în lucrarea [9], care conduce însă la creşterea complexității.

3.3. Arccos

Pentru *arccos* se pot utiliza aceleaşi ecuații (13) ca şi în cazul *arcsin*, în condițiile folosirii diferenței între componenta curentă x_i şi argumentul de intrare *a*, la fiecare iterație *i*, pentru evaluarea componentei d_i a vectorului de decizie:

$d_i = +1$ daca $x_i < a$ şi -1 dacă $x_i \geq a$, (15)

a = argumentul de intrarea.

Algoritmul operează corect pentru intrări *a*, care satisfac condiția:

$-1 < (a/G_n.y_0) < 1$ (16)

Arccos poate fi calculat, de asemenea, şi cu formula:

$\arccos(\theta) = \pi/2 - \arcsin(\theta)$ (17)

3.4. Arctg

Funcția arctg(x/y) poate fi calculată cu ajutorul *rotatorului* CORDIC în modul *vector*, dacă acumulatorul pentru unghi este setat la zero. Argumentul prezentat sub forma unui raport (x/y) favorizează reprezentarea infinitului prin fortarea $x = 0$. Rezultatul este obținut în acumulatorul pentru unghi:

$z_n = z_0 - \text{arctg}(y_0/.x_0)$ (18)

3.5. Modulul vectorului

Modulul vectorului este generat cu ocazia calculului funcției arctg(x/y). În finalul operației, vectorul este aliniat cu axa *x* şi, prin urmare, modulul vectorului este egal cu componenta *x* a vectorului rotit. Rezultatul este conform cu ecuația pentru *rotatorul* în modul *vector*:

$x_n = G_n.(x_0^2 + y_0^2)^{1/2}$ (19)

Modulul este scalat cu amplificarea G_n a procesorului CORDIC. Implementarea dată are complexitatea unui înmulțitor, care operează cu acelaşi număr de biti, iar fiecare iterație creşte precizia rezultatului cu 2 biți.

3.6. Transformarea coordonatelor Carteziene în coordonate Polare

În cazul utilizării modului *vector*, în condițiile unui unghi inițial $z_0 = 0$, unghiul din acumulator z_n şi componenta x_n ale rezultatului vor fi:

$z_n = \text{arctg}(y_0/.x_0)$

$x_n = G_n.(x_0^2 + y_0^2)^{1/2}$ (20)

care corespund valorii unghiului şi valorii modulului scalată cu amplificarea G_n.

3.7. Transformarea coordonatelor Polare în coordonate Carteziene

Transformarea coordonatelor Polare în coordonate Carteziene constituie o extensie a calculelor funcțiilor *sin* şi *cos*.

Fie *r* şi θ mărimea polară şi, respectiv, faza polară. Dacă în ec. (9) se consideră:

$x_0 = r$

$z_0 = \theta$ (21)

$y_0 = 0$

după *n* iterații, rezultatele *x* și *y* reprezintă coordonatele Carteziene:

$x_n = r.\cos(\theta)$

$y_n = r.\sin(\theta)$ (22)

4. Implementarea algoritmului CORDIC în FPGA

Înainte de implementarea propriu-zisă a algoritmului CORDIC trebuie avute în vedere unele aspecte privitoare la reprezentarea unghiurilor, a valorilor numerice şi a preciziei.

Atunci când *unghiurile* sunt date în radiani, de regulă, se are în vedere intervalul $[-\pi/2, \pi/2]$. Pentru valori aflate în afara intervalului, se vor utiliza identitățile trigonometrice, ceea ce asigura plasarea unghiului în intervalul $[-\pi/2, \pi/2]$.

Valorile numerice aparțin domeniului numerelor reale şi au o gamă limitată, de exemplu: $0 \le x < 1$, fapt care recomandă reprezentarea în virgulă fixă şi în complementul față de 2. Reprezentarea în virgulă fixă simplifică cuplarea cu senzori de tipul convertoarelor A/N şi N/A.

Este indicat să se estimeze *precizia* necesară, care se traduce prin numarul de biți utilizați în reprezentarea valorilor numerice, ceea ce determină numărul iterațiilor în algoritmul CORDIC.

Implementările algoritmului CORDIC în FPGA pot urmări, fie performanțe maxime, fie minimizarea hardware-lui.

4.1. Procesoare CORDIC iterative

Arhitectura iterativă rezultă direct din ecuațiile CORDIC, implementate cu ajutorul blocului de bază prezentat în figura 2.

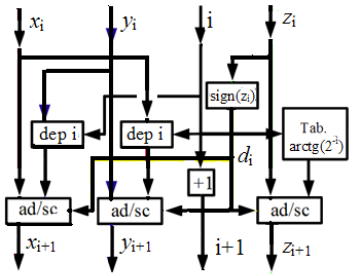

Figura 2. Bloc de bază CORDIC

Acest bloc de bază, completat cu resursele hardware necesare (registre, multiplexoare), după cum se arată în figura 3, se constituie în secțiunea de execuție a unui procesor CORDIC iterativ. Funcția de decizie d_i se obține cu ajutorul semnelor registrelor *y* sau *z*, corespunzător modului de operare: *rotire* sau *vector*.

Valorile inițiale x_0, y_0, 0, z_0 sunt încărcate inițial în registrele corespunzătoare. În continuare, la fiecare ciclu de ceas *i* conținuturile registrelor sunt transferate prin circuitele de deplasare şi prin circuitele sumatoare/scăzătoare, pentru a fi returnate în registrele surse corespunzătoare. Circuitele de deplasare primesc, la fiecare iterație *i*, constanta de deplasare necesară. De asemenea, se furnizează adresa corespunzătoare Tabelei/memoriei în care sunt stocate valorile pentru *arctg*. Ultima iterație *n* generează direct rezultatele de la ieşirile circuitelor sumator/scăzător. Arhitectura examinată reprezintă unitatea de execuție a procesorului CORDIC. Pentru completarea arhitecturii procesorului este nevoie de o unitate simplă de comandă, care va gestiona numărul iterației curente, constanta de deplasare şi adresa datei curente care va fi citită din Tabel.

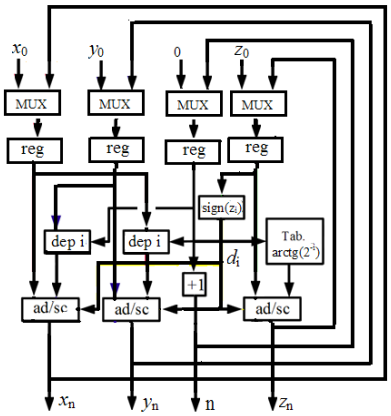

Figura 2. Arhitectura iterativă CORDIC

Arhitectura descrisă este de tip paralel la nivel de bit şi se caracterizează prin următoarele:

- complexitate spațială: $O(m)$, unde m reprezintă lungimea cuvântului; $O(m)$ poate fi asociat şi cu întârzierea în sumatorul/scăzătorul pentru numere de m biți;
- complexitatea temporală: $O(m.n)$, unde n este numărul iterațiilor;
- complexitatea temporală, în condițiile utilizării unor sumatoare/scăzătoare performante: $O(n.\log(m))$;
- latența: $n*(t_{bloc} + t_{reg})$, unde t_{bloc} este întârzierea combinațională pe bloc, iar t_{reg} reprezintă întârzierea în registru ($t_{setup} + t_{clock\ to\ Q}$).

În concluzie, soluția iterativă/secvențială, paralelă la nivel de bit, constituie un bun compromis între spațiul ocupat şi viteză.

O soluție mai economică [1], în ceea ce priveşte hardware-ul, se bazează pe o arhitectură serială, care constă în trei sumatoare/scăzătoare seriale, trei registre de deplasare şi o memorie serială, pentru implementarea tabelei de valori ale *arctg*. În vederea selectării prizelor necesare ale registrelor de deplasare, se folosesc porți sau multiplexoare, după cum se arată în figura 3.

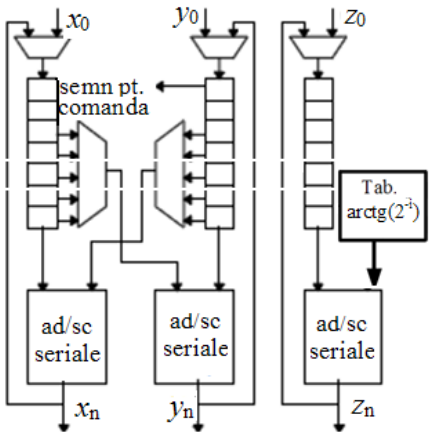

Figura 3. Arhitectura CORDIC serială, la nivel de bit

În această variantă de implementare, fiecare dintre cele n iterații se desfăşoară pe parcursul a m perioade de ceas. La inițializare, registrele de deplasare pot fi încărcate în paralel sau în serie. Înaintea fiecărei iterații, secțiunea de comandă citeşte semnul registrului y sau al registrului z, pentru a activa, fie operația de adunare, fie operația de scădere. De asemenea, se selectează priza corespunzătoare a registrului de deplasare. Pe parcursul celei de a n-a iterații, rezultatele pot fi citite de la ieşirile sumatoarelor seriale. Întrucât întârzierea într-un sumator serial este mai mică decât cea introdusă de către un sumator paralel, se poate opera la frecvențe de ceas mai ridicate, decât în soluția paralelă la nivel de bit.

Arhitectura serială la nivel de bit se caracterizează prin următoarele:

- complexitate spațială: $O(1)$, unde 1 este asociat cu lungimea cuvântului prelucrat de sumator/scăzător pe durata unui ciclu de ceas;
- complexitatea temporală: $O(m.n)$, unde n este numarul iterațiilor, iar m - numărul biților dintr-un cuvânt;

– latența: $n*m*(t_{bloc} + t_{reg})$, unde t_{bloc} este întârzierea combinațională pe sumator/scăzător serial, iar t_{reg} reprezintă întârzierea pe un bistabil al unui registru de deplasare ($t_{setup} + t_{clok\ to\ Q}$).

4.2 Procesoare CORDIC neiterative

Procesorul neiterativ se obține prin plasarea în cascadă a unor blocuri de bază CORDIC (figura 2), al căror număr este egal cu numărul *n* al iterațiilor, ceea ce poate fi interpretat ca o desfășurare spațială a *n* cicluri CORDIC iterative. Unitatea de execuție a unui asemenea procesor este ilustrată în figura 4.

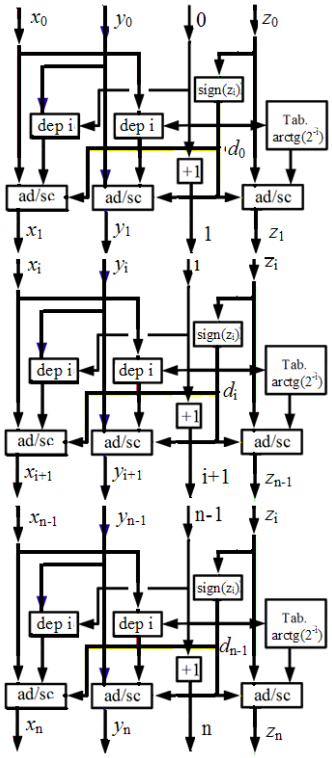

Figura 4. Unitatea de execuție a unui procesor CORDIC neiterativ

O asemenea implementare prezintă o serie de avantaje referitoare la realizarea efectivă a circuitelor de deplasare şi a tabelelor de valori pentru *arctg*. Astfel, la nivelul fiecărui bloc de baza, circuitul de deplasare efectuează deplasarea cu un număr fixat de poziții, ceea ce îl recomandă pentru o implementare cablată, mai simplă. De asemenea, în cadrul fiecărui bloc, valorile pentru *arctg* se pot distribui sub forma unor constante cablate, nemaifiind necesară o memorie pentru stocarea acestor valori.

Implementarea se caracterizează prin următoarele:
– complexitate spatială: $O(m.n)$;
– complexitatea temporală: $O(m.n)$;
– complexitatea temporală, în condițiile utilizării unor sumatoare/scăzătoare performante: $O(n.\ \log(m))$;
– latența: $n*t_{bloc}$.;
– productivitatea: $1/\ n*t_{bloc}$.

Blocurile CORDIC de bază se pot interconecta direct, fără a fi separate prin registre tampon, ceea ce conduce la o reducere a timpului de prelucrare, față de varianta iterativă. Desigur, aceasta soluție se remarcă prin prezența unui lanț combinațional de prelucrare, caracterizat printr-o întârziere apreciabilă, întârziere care contribuie la reducerea frecvenței ceasului sistemului şi implicit la o *productivitate* (*throughput*) mai redusă. Se poate însă remarca faptul că această productivitate este sensibil mai mare decât în cazul soluției iterative, deoarece în cazul din urmă, intervine la fiecare iterație latența registrului în care se acumulează rezultatele operației curente. Această latență, vizualizată ca o *regie* (*overhead*) de sistem, este de *n* ori mai mare în cazul procesorului iterativ, decât în cel al procesorului neiterativ.

Pentru a mări productivitatea, schema din figura 4 se poate completa cu registre, la nivelul fiecărui bloc de bază, generându-se, astfel, o structură de tip *bandă de asamblare*. Introducerea registrelor în blocurile de bază contribuie la creşterea regiei, la nivelul blocului de bază, oferind, în schimb, avantajul creşterii productivității, în sensul că, la fiecare ciclu de ceas, în banda de asamblare se află în curs de evaluare, în faze diferite, mai multe rezultate şi că, în fiecare ciclu de ceas, se generează un rezultat.

5. Rezultate experimentale

Implementarea algoritmului CORDIC în FPGA, care a fost precedată de simularea cu ajutorul aplicației ModelSim, s-a efectuat pentru varianta iterativă. Schema bloc a procesorului CORDIC este prezentată în figura 5. Mărimile de intrare sunt unghiul *theta* (16 biți), semnul unghiului, *sign* (1 bit), semnalul de ceas, *clock*, şi semnalul *reset*. Rezultatele, reprezentate pe 17 biți, au fost notate cu *CosX* şi *SinX*.

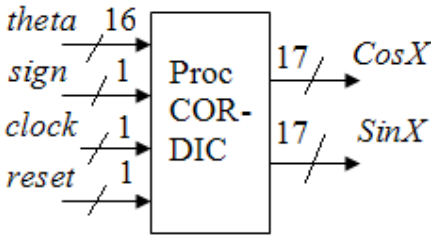

Figura 5. Schema bloc a procesorului CORDIC

5.1. Simularea

S-a descris în Verilog modulul cordic, în cadrul căruia au fost instanțiate modulele corespunzătoare resurselor hardware necesare „mecanizării" algoritmului: sumatoare, circuite de deplasare, multiplexoare, memorie. Pentru ca proiectul să se poată extinde la cerințe diferite, în ceea ce priveşte precizia, dimensiunea registrelor a fost parametrizată. Un fragment din descrierea Verilog a algoritmului CORDIC iterativ este prezentată mai jos:

```
module cordic_control(CosX,SinX,theta,sign,clock,reset);
  output [16:0] CosX,SinX;
          input [15:0] theta;
          input  sign,clock,reset;

          reg [16:0] CosX,SinX;
          wire [16:0] cos,sin;
          reg [4:0] cnt;
          reg rst;

  cordic mycordic(cos,sin,theta,Sign,clock,rst);

          always@(posedge clock)
           if (reset) begin
            cnt <= 0;
            CosX <= 0;
            SinX <= 0;
            rst <= 1;
           end else if (cnt<=16)
            begin
             cnt <= cnt + 1;
             CosX <= cos;
             SinX <= sin;
                    rst <= 0;
            end else rst <= 1;
endmodule

`define REG_SIZE 15
//Dimensiunea reală a registrului este: REG_SIZE+1.
module cordic(CosX,SinX,theta,sign,clock,reset);

          output [`REG_SIZE+1:0] CosX,SinX;
```

```
        input [`REG_SIZE:0] theta;
        input  sign,clock,reset;

        reg AngleCin,Xsign,Ysign;
        reg [`REG_SIZE:0] X,Y,Angle;
        reg [3:0] iteration;

        wire [`REG_SIZE:0] tanangle;
        wire [`REG_SIZE:0] BS1,BS2;
        wire [`REG_SIZE:0] SumX,SumY,SumAngle;
        wire CarryX,CarryY,AngleCout;

    shifter SH1(BS1,Y,iteration);
    Adder AddX(SumX,CarryX,Xsign,X,BS1,~AngleCin);
    shifter SH2(BS2,X,iteration);
    Adder AddY(SumY,CarryY,Ysign,Y,BS2,AngleCin);
    Adder Add0(SumAngle,AngleCout,AngleCin,Angle,tanangle,~AngleCin);
    assign CosX={CarryX,SumX};
    assign SinX={CarryY,SumY};
```

Modulul de memorie MEM, descris în Verilog, este ilustrat în continuare:

```
module MEM (iteration, tanangle);
  output [`REG_SIZE:0] tanangle;
  input [3:0] iteration;
  reg [`REG_SIZE:0] tanangle;

   always @ (iteration)
   case (iteration)
  4'b0000: tanangle <= 16'b00101101_00000000 ; //  1.000000 |45.000000
  4'b0001: tanangle <= 16'b00011010_11111111 ; //  0.500000 |26.565051
  4'b0010: tanangle <= 16'b00001110_00001111 ; //  0.250000 |14.036243
  4'b0011: tanangle <= 16'b00000111_00111111 ; //  0.125000 |7.125016
  4'b0100: tanangle <= 16'b00000011_11111111 ; //  0.062500 |3.576334
  4'b0101: tanangle <= 16'b00000001_11111111 ; //  0.031250 |1.789911
  4'b0110: tanangle <= 16'b00000000_11111111 ; //  0.015625 |0.895174
  4'b0111: tanangle <= 16'b00000000_01111111 ; //  0.007812 |0.447614
  4'b1000: tanangle <= 16'b00000000_00111111 ; //  0.003906 |0.223811
  4'b1001: tanangle <= 16'b00000000_00011111 ; //  0.001953 |0.111906
  4'b1010: tanangle <= 16'b00000000_00001111 ; //  0.000977 |0.055953
  4'b1011: tanangle <= 16'b00000000_00000111 ; //  0.000488 |0.027976
  4'b1100: tanangle <= 16'b00000000_00000011 ; //  0.000244 |0.013988
  4'b1101: tanangle <= 16'b00000000_00000001 ; //  0.000122 |0.006994
  4'b1110: tanangle <= 16'b00000000_00000000 ; //  0.000061 |0.003497
  4'b1111: tanangle <= 16'b00000000_00000000 ; //  0.000031 |0.001749

  endcase
endmodule
```

Rezultatele obţinute în urma simulării, pentru unghiuri *theta* cuprinse între 0 şi 45 grade, sunt date parţial în Tabelul 2.

```
#            1020 CosX = 0.993820 SinX = 0.004074  theta=+  0
#            2640 CosX = 0.994125 SinX = 0.019501  theta=+  1
#            4260 CosX = 0.993729 SinX = 0.035126  theta=+  2
#            5880 CosX = 0.993073 SinX = 0.050629  theta=+  3
.....................................................................................
.....................................................................................
#           46380 CosX = 0.881531 SinX = 0.472672  theta=+ 28
#           48000 CosX = 0.874146 SinX = 0.486252  theta=+ 29
#           49620 CosX = 0.866409 SinX = 0.499969  theta=+ 30
```

```
.....................................................................................
#          70680 CosX = 0.732651 SinX = 0.681000  theta=+ 43
#          72300 CosX = 0.722076 SinX = 0.692245  theta=+ 44
#          73920 CosX = 0.703491 SinX = 0.711105  theta=+ 45
```

Analiza rezultatelor simulării arată, în unele cazuri, mici discrepanţe între valorile rezultate din simularea execuţiei algoritmului descris în Verilog şi valorile funcţiilor *sin* şi *cos* obţinute prin alte metode. Diferenţele se pot explica prin numărul relativ mic de biţi utilizaţi pentru reprezentarea datelor.

5.2. Implementarea algoritmului CORDIC iterativ în FPGA

Implementarea algoritmului s-a realizat cu ajutorul plachetei **Xilinx Spartan-3 AN Starter Kit** (figura 6), bazată pe circuitul FPGA- XC3S700AN-FGG484, circuit capabil să satisfacă proiecte care necesită până la 700.000 porţi logice. Placheta posedă un generator propriu de ceas, cu frecvenţa de 50 MHz, şi interfeţe: USB, VGA, PS/2, RS-232 port serial, Ethernet, A/N (2), N/A (4) etc. Astfel, pentru experimentarea proiectelor bazate pe circuite FPGA, s-a proiectat şi realizat o platformă, constând în placheta, echipată cu circuitul XC3S700AN-FG484, tastatura PS/2 şi monitorul VGA.

Figura 6. Placheta Xilinx Spartan-3 AN Starter Kit

Au fost proiectate, descrise în Verilog şi implementate modulele de interconectare pentru tastatura PS/2 şi monitor VGA, la un procesor generic ale cărui structură şi funcţii pot diferi de la caz la caz. În situaţia procesorului CORDIC, de la tastatură se introduce unghiul *theta* (în hexa), iar pe ecranul monitorului se obţin rezultatele CosX şi SinX (în hexa). Platforma experimentală este ilustrată în figura 7. Timpul de calcul pentru funcţiile CosX / SinX este dat de numărul de iteraţii înmulţit cu perioada de ceas. Ceasul utilizat în cadrul experimentului

Figura 7. Platforma experimentală

este de 50MHz, numărul de iteraţii fiind 16. Astfel, un rezultat, inclusiv afişarea, necesită un interval de timp egal cu 320 ns. Dacă se ia în consideraţie numai modulul CORDIC, **XST** (Xilinx® Synthesis Technology) raportează: "*Minimum period*: 13.549ns (*Maximum Frequency*: 73.806MHz)" pentru FPGA-ul ales. În tabelul 2, se prezintă, în rezumat, utilizarea resurselor hardware ale FPGA-ului XC3S700AN-FGG484, pentru implementarea interfeţelor cu tastatura PS/2, cu monitorul VGA cât şi a procesorului CORDIC.

Tabelul 2. Utilizarea resurselor hardware ale FPGA-ului XC3S700AN-FGG484

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	191	11,776	1%
Number of 4 input LUTs	582	11,776	4%
Logic Distribution			
Number of occupied Slices	366	5,888	6%
Number of Slices containing only related logic	366	366	100%
Number of Slices containing unrelated logic	0	366	0%
Total Number of 4 input LUTs	627	11,776	5%
Number used as logic	581		
Number used as a route-thru	45		
Number used as Shift registers	1		
Number of bonded IOBs	27	372	7%
Number of BUFGMUXs	1	24	4%

6. Concluzii

Cu toate că algoritmii CORDIC sunt cunoscuți de către specialiștii în calcul numeric, cât şi de către cei care lucrează în domeniul calculului de înaltă performanță, implementarea lor în FPGA nu şi-a epuizat pe deplin posibilitățile. Dintre modalitățile de implementare: secvențială, paralelă, paralelă cu sumatoare cu transport rapid şi banda de asamblare, se poate selecta aceea care corespunde mai bine cerințelor de proiectare: arie ocupată (număr de tabele look-up, porți), putere consumată, întârziere pe iterație/întârziere totală. Lucrarea a demonstrat posibilitățile de realizare facilă în FPGA a algoritmilor de tip CORDIC, în condițiile existenței unor unelte performante de simulare, sinteză şi implementare cum sunt **ModelSim XE III 6.1** şi **FPGA Xilinx Spartan-3 AN Starter Kit.**

Bibliografie

1. **ANDRAKA, R. J.**: A Survey of CORDIC Algorithms for FPGA Based Computers. ACM 0-89791-978-5/1998/01.
2. **VOLDER, J**.: Binary Computation Algorithms for Coordinate Rotation and Function Generation, Convair Report IAR-1 148 Aeroelectrics Group, June 1956.
3. **VOLDER, J**.: The CORDIC Trigonometric Computing Technique. IRE Trans. Electronic Computing, Vol EC-8, Sept 1959, pp330-334.
4. **WALTHER, J.S**.: A unified algorithm for elementary functions. Proc. of the Spring Joint Computer Conf.,1971, pp. 379-385.
5. **ANDRAKA, R. J.**: Building a High Performance Bit-Serial Processor în an FPGA. Proc. of Design SuperCon '96, January 1996. pp. 5.1 - 5.21
6. **DUPRAT, J. J.M. MULLER**: The CORDIC Algorithm: New Results for Fast VLSI Implementation. IEEE Transactions on Computers, Vol. 42, 1993, pp. 168-178.
7. **DEPRETTERE, E., P. DEWILDE, R. UDO**: Pipelined CORDIC Architecture for Fast VLSI Filtering and Array Processing. Proc. ICASSP'84, 1984, pp. 41.A.6.1- 41.A.6.4.
8. Marchesi, M., G. Orlandi, F. Piazza: Systolic Circuit for Fast Hartley Transform. Proc. of IEEE International Symposium on Circuits and Systems, Espoo, Finland, June 1988, pp. 2685-2688.
9. **HU, Y.H., S. NAGANATHAN**: A Novel Implementation of Chirp Z-Transformation Using a CORDIC Processor. IEEE Transactions on ASSP, Vol. 38, 1990, pp. 352-354.
10. **AHMED, H. M., J.M. DELOSME, M. MORF**: Highly Concurrent Computing Structure for Matrix Arithmetic and Signal Processing. IEEE Comput. Mag., Vol. 15, 1982, pp. 65-82.
11. **SINNEN, O**.: Reconfigurable Computing - CORDIC algorithms. Electrical and Computer Engineering The University of Auckland. http://www.cs.auckland.ac.nz/~jmor159/reconfig/lectures.html
12. **www.altera.com**. Implementation of CORDIC-Based QRD-RLS Algorithm on Altera Stratix FPGA with Embedded Nios Soft Processor Technology. White Paper. Copyright © 2003 Altera Corporation.
13. **ANGARITA F., A. PEREZ-PASCUAL, T. SANSALONI, J. VAILS**: Efficient FPGA implementation of Cordic algorithm for circular and linear coordinates. Field Programmable Logic and Applications, 2005. Int. Conf. on Volume, Issue, 24-26 August, 2005 pp. 535 – 538.