Universitatea Tehnică din Cluj-Napoca
Departamentul de Calculatoare

Structura Sistemelor de Calcul

Proiect: Proiectarea unei unități aritmetico-logice (UAL)

Student: Lazea Dragoș-Bogdan
Grupa: 30233

Profesor îndrumător: Hangan Anca

An universitar: 2021 – 2022

Cuprins

I. Propunere de proiect

Se dorește proiectarea și implementarea unei unități aritmetice și logice (UAL) care să opereze cu întregi reprezentați pe 32 de biți în complement față de 2 (C2) și să permită efectuarea următoarelor operații aritmetice: adunare, scădere în complement față de 2, incrementare, decrementare și negare și logice: ȘI-logic pe biți, SAU-logic pe biți, NU-logic pe biți, rotire logică spre stânga și rotire logică spre dreapta. Unitatea aritmetico-logică va prelua unul dintre operanzi dintr-un registru acumulator, iar rezultatul operației va fi stocat în același registru acumulator. De asemenea, va fi utilizat un circuit suplimentar pentru înmulțire și împărțire. În plus, se va proiecta o unitate de control responsabilă de specificarea operației care urmează a fi efectuată de către unitatea aritmetică și logică. Implementarea UAL se va realiza în maniera structurală în limbajul de descriere hardware VHDL.

II. Studiu bibliografic

Unitatea aritmetico-logică (UAL) este un bloc esențial al unității centrale de prelucrare a fiecărui sistem de calcul, reprezentând un circuit combinațional care efectuează operații aritmetice și operații logice pe biți, având drept operanzi numere întregi. În general, UAL beneficiază de un registru acumulator din care preia unul dintre operanzi și în care furnizează rezultatele operațiilor efectuate.

- **Aritmetica binară**

Pentru a conferi flexibilitate implementării operațiilor aritmetice și logice, atât operanzii, cât și rezultatul operațiilor efectuate de UAL sunt reprezentați în complement față de 2, reprezentarea numerelor negative în acest format obținându-se prin negarea tuturor biților componenți ai reprezentării binare corespunzătoare valorii absolute a numărului și incrementarea valorii obținute, în timp ce numerele pozitive sunt reprezentate ca în formatul mărime și semn. Astfel, gama valorilor ce pot fi reprezentate în complement față de 2 pe *n* biți se întinde de la valoarea -2^{n-1} până la $2^{n-1}-1$ (în cadrul proiectării unității aritmetico-logice de față se va alege $n = 32$). Prin urmare, valoarea zecimală a unui număr binar, reprezentat în complement față de 2 pe *n* biți după cum urmează:

$$b_{n-1}b_{n-2}\dots b_1b_0$$

este:

$$-b_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

Avantajul folosirii acestei reprezentări îl constituie faptul că scăderea folosește aceeași logică binară ca adunarea, opusul unui număr în complement față de 2 obținându-se prin adunarea cu 1 a valorii complementului față de 1 corespunzătoare numărului:

$$x - y = x + (-y) = x + (\bar{y} + 1)$$

Acest avantaj se reflectă în etapa de proiectare a unității aritmetice și logice prin reducerea numărului de componente necesare efectuării operațiilor aritmetice, adunarea și scăderea făcând astfel apel la aceleași componente hardware elementare.

- **Strategia de proiectare – componentele UAL**

Pentru a putea efectua operații aritmetice și logice pe operanzi reprezentați pe 32 de biți, UAL interconectează 32 de componente care vor realiza operațiile corespunzătoare pe numere reprezentate pe 1 bit, generând rezultate intermediare care vor fi asociate astfel încât să conducă la obținerea rezultatului final pe 32 de biți. Blocurile constructive de bază ale fiecărei astfel de componente UAL ce realizează operațiile pe 1 bit sunt reprezentate de porți logice și multiplexoare.

Registrul acumulator are rolul de a stoca atât valoarea unuia dintre operanzii de intrare, cât și cea a rezultatului generat în urma efectuării operațiilor de către UAL.

Unitatea de control (UC) este destinată specificării operației realizate de UAL din setul disponibil, prin generarea unor semnale de control dedicate fiecărei operații în parte care au scopul de a caracteriza starea în care se află unitatea proiectată și de a asigura tranziția corespunzătoare de stări..

Circuitul suplimentar de înmulțire și împărțire se bazează pe implementarea unor algoritmi de înmulțire și împărțire hardware, fiind așadar proiectat sub forma unui automat cu stări finite, comandat, de asemenea, de unitatea de control. Înmulțirea se realizează prin implementarea algoritmului de înmulțire de tip ***Shift-and-Add Multiplication***, în timp ce împărțirea urmărește un algoritm de tip ***Restoring Division***, care vor fi detaliați ulterior, în cadrul descrierii etapelor de proiectare.

III. Planificare

- **Săptămâna 3 (11 oct. 2021)** – Propunere de proiect, studiu bibliografic și planificare

- **Săptămâna 5 (25 oct. 2021)** – Proiectare și implementare componente pentru UAL pe 1 bit, proiectare și implementare componente pentru UAL pe 32 de biți prin interconectarea a 32 componente pentru UAL pe 1 bit

- **Săptămâna 7 (8 noi. 2021)** – Sinteză și implementare UC pentru UAL, proiectare circuite suplimentare pentru înmulțire/împărțire

- **Săptămâna 9 (22 noi. 2021)** – Implementare înmulțitor și împărțitor

- **Săptămâna 11 (6 dec. 2021)** – Interconectare componente, testare, rezultate experimentale

IV. Analiză

Etapa de analiză se bazează pe descrierea listei de funcționalități menționate în specificația proiectului și are drept scop identificarea algoritmilor ce urmează a fi implementați, în vederea identificării elementelor de design, cât și a macro-componentelor ce stau la baza proiectării unității aritmetice și logice.

1. *Operații aritmetice de bază*

a. *Adunarea*

Operația de adunare este o operație fundamentală, întrucât majoritatea operațiilor aritmetice au la bază operația de adunare: scăderea este o adunare cu opusul scăzătorului, înmulțirea reprezintă o adunare repetată, iar împărțirea o succesiune de scăderi și adunări.

Punctul de plecare în descrierea operației de adunare a numerelor reprezentate pe 32 de biți îl reprezintă adunarea numerelor reprezentate pe un singur bit. Ecuațiile ce descriu această operație sunt:

$$S = X_1 \oplus X_2 \oplus C_{in}$$
$$C_{out} = X_1 X_2 + (X_1 \oplus X_2) C_{in}$$

Algoritmul folosit pentru descrierea și implementarea operației de adunare pe numere reprezentate pe 32 de biți este *Ripple-Carry Add*, având la bază ideea de propagare a transportului de ieșire generat la un rang inferior drept transport de intrare pentru rangul imediat următor. Practic algoritmul urmează logica adunării zecimale cu propagarea transportului spre rangul imediat superior. Prin urmare, ecuațiile pentru adunarea biților corespondenți de pe poziția *i* devin:

$$S_i = X_{1_i} \oplus X_{2_i} \oplus C_{in_i}$$
$$C_{out_i} = X_{1_i} X_{2_i} + \left(X_{1_i} \oplus X_{2_i}\right) C_{in_i}$$

În cazul în care rezultatul operației de adunare este prea mare pentru a putea fi reprezentat pe 32 de biți apare o depășire a domeniului de reprezentare. O astfel de depășire este se înregistrează, în cadrul operației de adunare, doar dacă:

- **operanzii au același semn** (în cazul semnelor diferite, domeniul de reprezentare nu poate fi depășit, operația reprezentând de fapt o scădere)
- **semnul rezultatului diferă de semnele operanzilor** (s-a depășit gama disponibilă pentru reprezentare)

X₁	***X₂***	***X₁ + X₂***	***Depășire***
≥ 0	≥ 0	≥ 0	NU
≥ 0	≥ 0	< 0	DA
≥ 0	< 0	≥ 0	NU
≥ 0	< 0	< 0	NU
< 0	≥ 0	≥ 0	NU
< 0	≥ 0	< 0	NU
< 0	< 0	≥ 0	DA
< 0	< 0	< 0	NU

Tabel IV-1 1 Cazuri de depășire la adunare

Prin urmare, detectarea și semnalizarea depășirii se realizează folosind elementele logicii combinaționale de bază după cum urmează:

$$overflow_+ = X_{1_{31}} X_{2_{31}} \overline{S_{31}} + \overline{X_{1_{31}}}\ \overline{X_{2_{31}}} S_{31}$$

Practic, depășirea apare atunci când transportul de intrare și cel de ieșire de la rangul cel mai semnificativ diferă:

$$overflow_+ = C_{out_{31}} \oplus C_{in_{31}} = C_{out_{31}} \oplus C_{out_{30}}$$

b. *Scăderea*

Operația de scădere exploatează avantajele reprezentării numerelor binare în complement față de 2, pornind de la premisa că opusul unui număr în complement față de 2 este obținut în urma incrementării complementului față de 1, obținut prin negarea tuturor biților, corespunzător numărului:

$$-x = \bar{x} + 1$$

Așadar, operația de scădere se reduce la o simplă adunare cu opusul scăzătorului, astfel obținut:

$$X_1 - X_2 = X_1 + (\overline{X_2} + 1)$$

Extensia operației de scădere de la 1 bit la 32 de biți se realizează similar cu cea a operației de adunare, scăderea reprezentând, în fapt, o adunare.

În ceea ce privește depășirea domeniului, în cadrul operației de scădere, aceasta are o semnificație opusă depășirii întâlnite la operația de adunare, putând fi înregistrată doar atunci când:

- **operanzii au semn diferit** (în cazul semnelor identice, rezultatul domeniul nu poate fi depășit la scădere)
- **semnul rezultatului este identic cu semnul scăzătorului** (valoarea absolută a scăzătorului este mai mare decât valoarea absolută a descăzutului)

X₁	***X₂***	***X₁ + X₂***	***Depășire***
≥ 0	≥ 0	≥ 0	NU
≥ 0	≥ 0	< 0	NU
≥ 0	< 0	≥ 0	NU
≥ 0	< 0	< 0	DA
< 0	≥ 0	≥ 0	DA
< 0	≥ 0	< 0	NU
< 0	< 0	≥ 0	NU
< 0	< 0	< 0	NU

Tabel IV-1 2 Cazuri de depășore la scădere

Astfel, ecuația logică prin care poate fi detectată depășirea generată în urma unei operații de scădere este:

$$overflow\_ = X_{1_{31}}\overline{X_{2_{31}}}\,\overline{S_{31}} + \overline{X_{1_{31}}}\,X_{2_{31}}S_{31}$$

c. *Incrementarea*

Incrementare reprezintă operația prin care operandul considerat își crește valoarea cu o unitate.

$$x \leftarrow x + 1$$

Operația de incrementare se reduce, de asemenea, la o operație de adunare, în care cel de-al doilea operand are valoarea egală cu 1, algoritmii, regulile și cazurile de depășire detaliate în cadrul descrierii operației de adunare păstrându-și în totalitate valabilitatea.

d. *Decrementarea*

În mod total analog cu operația de incrementare, decrementarea se definește ca fiind operația prin care operandul considerat își micșorează valoarea cu o unitate.

$$x \leftarrow x - 1$$

Așadar, operația de incrementare reprezintă în fapt o operație de scădere în care scăzătorul are valoarea egală cu 1. De asemenea, algoritmii și cazurile de depășire prezente la operația de scădere sunt moștenite și de operația de decrementare.

e. *Negarea*

Operația de negare are drept scop obținerea opusului unui număr. Aceasta folosește proprietățile reprezentării în formatul complement față de 2, care se bazează pe ideea că opusul unui număr se obține prin complementarea tuturor biților din reprezentarea binară a numărului și incrementarea valorii obținute. Practic, se obține complementul față de 1 al numărului căruia i se dorește aflarea valorii opuse, iar apoi acestuia i se adună valoarea 1, conform următorului procedeu:

$$\bar{x} = \overline{x_{31}}\,\overline{x_{30}} \ldots \overline{x_2}\,\overline{x_1}\,\overline{x_0}$$
$$-x = \bar{x} + 1$$

2. *Operații logice pe biți*

a. *ȘI-logic*

Operația ȘI-logic (AND) este o operație logică care produce un rezultat având valoarea 1-logic doar atunci când ambii biți ce reprezintă operanții au valoarea 1-logic.

Operația AND la nivel de bit se definește conform următorului tabel de adevăr:

A	*B*	*A and B*
0	0	0
0	1	0
1	0	0
1	1	1

Tabel IV-2 1 Tabel de adevăr ȘI-logic

Rezultatul operației AND având operanzii reprezentați pe 32 de biți se obține prin aplicarea operației AND pe fiecare pereche de biți aflați pe aceeași poziție în reprezentarea operanzilor, astfel:

$$Y_i = X_{1_i} and\ X_{2_i}$$

b. *SAU-logic*

Operația SAU-logic (OR) se definește ca fiind operația ce generează un 1-logic atunci când cel puțin unul dintre biții operanzi are valoarea 1-logic.

Tabelul de adevăr aferent funcției SAU-logic este:

A	*B*	*A or B*
0	0	0
0	1	1
1	0	1
1	1	1

Tabel IV-2 2 Tabel de adevăr SAU-logic

Analog cu operația ȘI-logic, SAU-logic efectuată pe operanzi reprezentați în binar pe 32 de biți se realizează prin aplicarea operației pe fiecare pereche de biți de pe aceeași poziție:

$$Y_i = X_{1_i} or\ X_{2_i}$$

c. *NU-logic*

Operația NU-logic (NOT) este o operație logică având un singur operand, căruia îi neagă valoarea, conform următorului tabel de adevăr:

A	*not A*
0	1
1	0

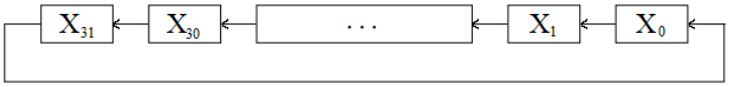
Tabel IV-2 3 Tabel de adevăr NU-logic

Prin negarea fiecărui bit a unui număr reprezentat în binar pe 32 de biți de obține, de fapt, complementul față de 1 al numărului respectiv:

$$y = \bar{x} = \overline{x_{31}}\,\overline{x_{30}} \dots \overline{x_2}\,\overline{x_1}\,\overline{x_0}$$

d. *Rotire logică spre stânga*

Operația de rotire logică spre stânga (Shift-Left Logical) realizează o deplasare circulară spre stânga a biților componenți ai operandului cu o poziție binară. Pe poziția bitului cel mai puțin semnificativ, eliberată prin rotire, va ajunge bitul cel mai semnificativ, care iese din operand:



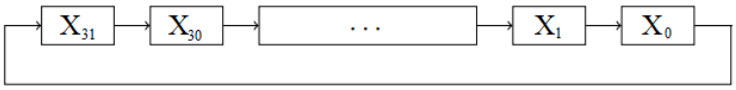
Figură IV-2 1 Rotire logică spre stânga

Rezultatul operației este, așadar, următorul:

$$y = x_{30}x_{29} \dots x_1x_0x_{31}$$

e. *Rotire logică spre dreapta*

Rotirea logică spre dreapta (Shift-Right Logical) reprezintă operația opusă rotirii logice spre stânga, aceasta realizând deplasarea circulară a operandului spre dreapta cu o poziție binară:



Figură IV-2 2 Rotire logică spre dreapta

În urma efectuării operației pe un operand, *x* reprezentat în binar pe 32 de biți, se obține următorul rezultat:

$$y = x_0x_{31} \dots x_3x_2x_1$$

3. *Operații aritmetice suplimentare (înmulțire și împărțire)*

Operațiile de înmulțire și împărțire se dovedesc a fi cele mai costisitoare operații efectuate la nivelul unității aritmetice și logice, acestea constând dintr-o succesiune de adunări și scăderi, după caz. Astfel, pentru proiectarea și implementarea acestora este necesară studierea și descrierea unor algoritmi specifici prin implementarea cărora pot fi obținute rezultatele dorite.

a. *Înmulțirea*

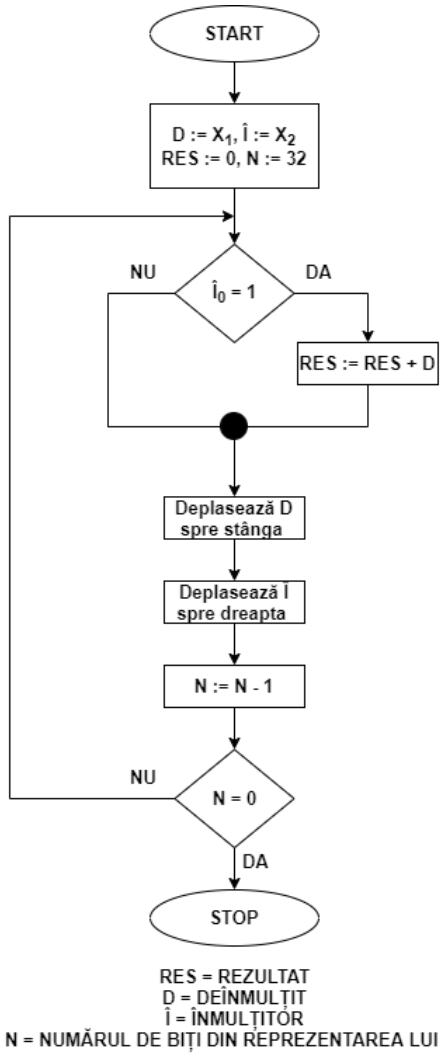
Operația de înmulțire este la bază o adunare repetată, cel mai simplu algoritm de înmulțire binară fiind Shift-And-Add Multiplication. Această abordare se bazează pe adunarea operandului X_1 la el însuși de X_2 ori, unde X_1 reprezintă deînmulțitul, iar X_2 înmulțitoruș. Practic, strategia folosește aceeași logică precum înmulțirea zecimală, putând fi descrisă de următoarele etape:

1. Se consideră cifrele înmulțitorului de la stânga la dreapta;

2. Se înmulțește fiecare cifră considerată cu fiecare cifră a deînmulțitului;
3. Se plasează rezultatul intermediar obținut cu o poziție mai spre stânga față de anteriorul rezultat obținut;
4. Se adună rezultatele intermediare obținute, respectându-se indentarea obținută prin deplasarea cu o poziție spre stânga a rezultatelor intermediare succesive.

În cazul înmulțirii binare, fiecare etapă a algoritmului este simplificată, întrucât singurele cifre din sistemul de numerație binar sunt 0 și 1. Prin urmare, dacă cifra considerată a înmulțitorului este 1 se va copia deînmulțitul în tocmai, cu o poziție mai spre stânga față de rezultatul anterior obținut, iar dacă cifra considerată este 0, se vor adăuga, de asemenea, mai spre stânga, un număr de 0-uri egal cu numărul de cifre am deînmulțitului.

Succesiunea de pași urmați de algoritmul Shift-And-Add Multiplication poate fi regăsită în schema logică de mai jos:



Figură IV-3 1 Algoritmul de înmulțire Shift-And-Add Multiplication

D și *Î* se vor reprezenta pe un număr de biți egal cu dublul numărului de biți din reprezentarea lui X_1 ($2N = 64$ *biți*), astfel încât să

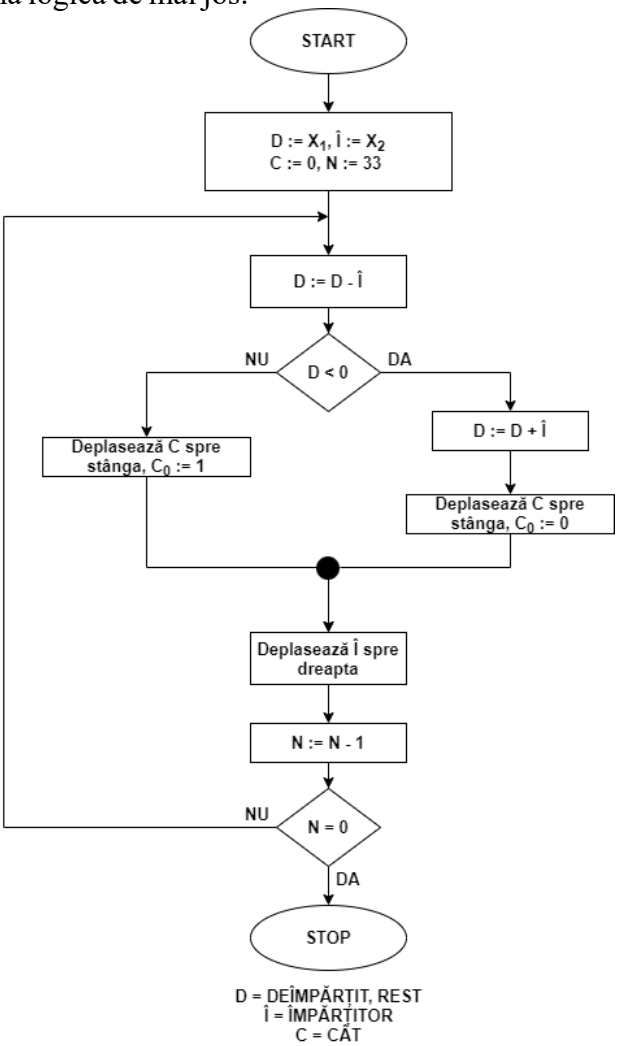
permită reprezentarea rezultatelor parțiale, deplasate spre stânga cu câte o poziție față de rezultatul parțial anterior.

b. *Împărțirea*

Operația de împărțire este de departe cea mai costisitoare, unul dintre algoritmii de bază pentru împărțirea binară fiind *Restoring Division (Împărțire cu refacerea restului)*. În fiecare pas al algoritmului are loc o deplasare a împărțitorului spre dreapta cu o poziție și o deplasare a câtului spre stânga cu o poziție binară, algoritmul ajungând la final în momentul în care împărțitorul ajunge mai mic decât restul împărțirii.

Ideea fundamentală a algoritmului este următoarea: pentru a verifica dacă împărțitorul este mai mic decât restul, se scade împărțitorul din valoarea curentă a restului; dacă rezultatul operației este unul negativ pasul următor îl reprezintă restabilirea valorii precedente prin adunarea împărțitorului la valoarea restului și deplasarea câtului spre stânga cu o poziție, astfel încât bitul C_0 să aibă valoarea 0.

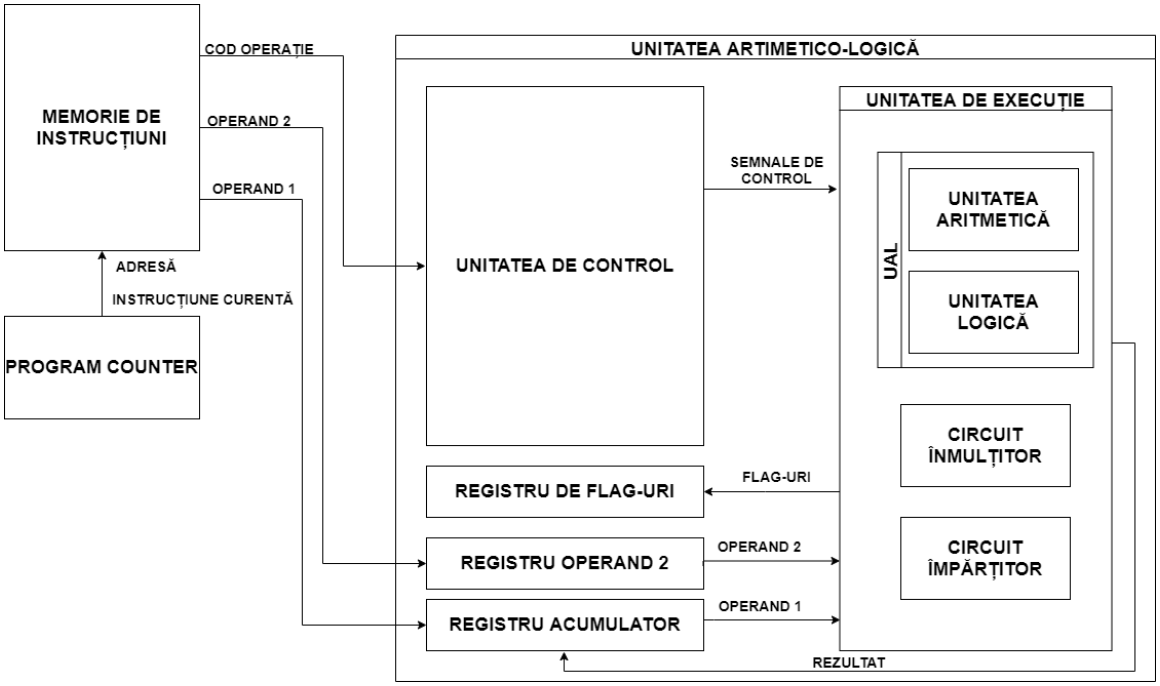
Succesiunea pașilor care descriu algoritmul sunt descriși în schema logică de mai jos:



Figură IV-3 2 Algoritmul de împărțire Restoring Division

V. *Design*

Luând în considerare lista funcționalităților ce trebuie satisfăcute de către unitatea ce urmează a fi proiectată, se pot identifica macro-componentele circuitului, vizibile în schema bloc mai jos prezentată:



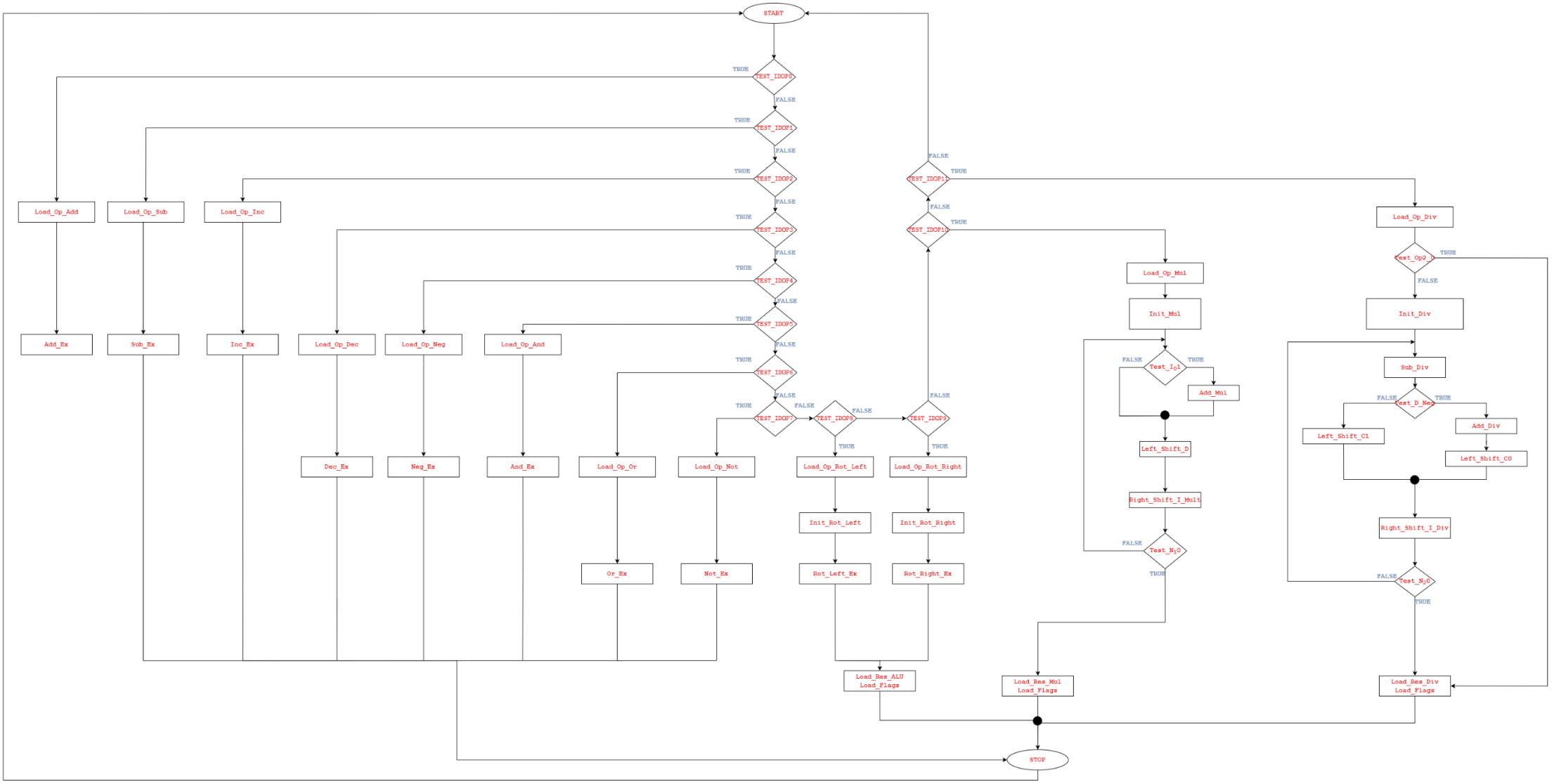
Figură V 1 Schema bloc a unității aritmetice și logice

Comunicarea între componentele prezente în cadrul circuitului se realizează prin intermediul magistralelor de date, după cum poate fi observat în schema bloc.

Așadar, componentele principale ale unității aritmetice și logice sunt:

1. **Unitatea de execuție (UE):** este destinată efectuării propriu zise a operațiilor din lista de funcționalități. Aceasta constă dintr-o **unitate logică**, definită la nivelul cel mai de bază folosind porți logice interconectate pentru a putea realiza operații pe numere reprezentate pe 32 de biți, o **unitate aritmetică**, destinată operațiilor de natură aritmetică și construită folosind multiplexoare și sumatoare definite, de asemenea, cu ajutorul porților logice și, nu în ultimul rând, două **circuite suplimentare pentru operațiile de înmulțire și împărțire**. Aceasta preia unul dintre operanzi din registrul acumulator, cel de-al doilea operand fiind preluat dintr-un registru dedicat, efectuează operația corespunzătoare combinației valorilor semnalelor de control generate de unitatea de control, scrie rezultatul obținut în registrul acumulator și setează flag-uri care furnizează informații referitoare la rezultatul obținut și eventualele excepții înregistrate.

2. **Unitatea de control (UC):** aceasta este responsabilă de generarea semnalelor de control și transmiterea acestora către unitatea de execuție în vederea selectării operației de efectuat și stabilirii succesiunii de stări prin care circuitul va urma să treacă pentru a obține rezultatul operației curente, fiind proiectată sub forma unui automat cu stări finite.



Figură V 2 Schema logică pentru unitatea de control

3. **Registrul de flag-uri (de stare):** conține valorile flag-urilor generate în urma operațiilor efectuate de către unitatea de execuție:
   - **Z** ***Z****ero flag:* setat la valoarea 1 când rezultatul operației din UAL este 0
   - **C** ***C****arry flag:* setat la 1 când se obține un transport la rangul cel mai semnificativ în cadrul rezultatului operației din UAL;
   - **P** ***P****arity flag:* setat la valoarea 1 când rezultatul obținut reprezintă un număr impar;
   - **O** ***O****verflow flag:* setat la 1 atunci când se înregistrează o depășire a domeniului de reprezentare;
   - **A** ***A****uxiliary Carry flag:* setat la valoarea 1 atunci când se înregistrează un transport de la rangul 4 (cei mai puțin semnificativi 4 biți) la rangul imediat următor;
   - **S** ***S****ign flag:* setat la valoarea 1 atunci când rezultatul obținut în urma operației este un număr negativ;
   - **D** ***D****ivide By Zero flag:* setat la 1 în cazul unei operații de împărțire în care împărțitorul are valoarea 0.

   Structura registrului de stare, reprezentată de succesiunea flag-urilo mai sus precizate, este următoarea:

S	O	P	Z	D	A	C

Figură V 3 Structura registrului de stare

4. **Registrul acumulator:** este responsabil cu memorarea unuia dintre operanzii implicați și reținerea rezultatului operației indicate.

5. **Registrul pentru al doilea operand:** stochează cel de-al doilea operand, pe 32 de biți.

6. **Memoria de instrucțiuni:** este o memorie de tip ROM, care conține instrucțiuni hardcodate, destinate testării unității aritmetice și logice. Instrucțiunile reținute în memorie sunt construite din câmpuri pentru operanzi și un câmp pentru codul operației de efectuat, folosit de către unitatea de control pentru generarea semnalelor de control corespunzătoare.

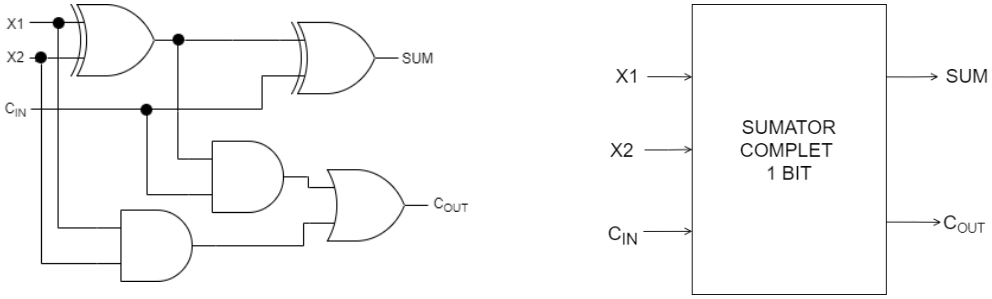
7. **Contorul de program:** reprezintă un numărător care va asigura trecerea de la instrucțiunea curentă din memoria de instrucțiuni la instrucțiunea imediat următoare, indicând locația de la care se va face citirea din memorie.

În continuare se vor prezenta componentele fundamentale ce stau la baza fiecărei operații din lista de funcționalități, conform descrierii prezentate în cadrul capitolului de analiză.

1. *Operații aritmetice de bază*

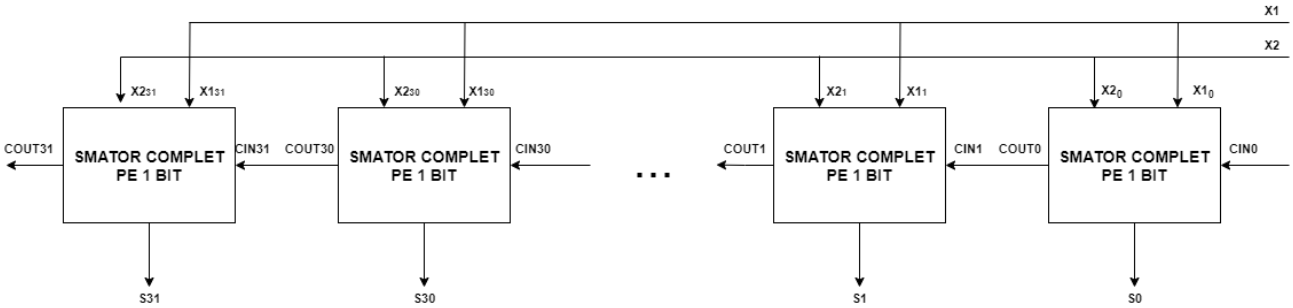
a. *Adunarea*

Adunarea a două numere reprezentate pe 1 bit se realizează folosind un sumator complet pe 1 bit, realizat cu porți logice, conform ecuațiilor logice descrise în secțiunea de analiză, după cum vor fi prezentate în figurile de mai jos.



Figură V-1 1 Sumator complet pe 1 bit

Pentru a putea realiza operația de adunare pe numere de 32 de biți, conform algoritmului descris, se vor interconecta serial 32 de astfel de sumatoare complete pe 1 bit, ieșirea de transport de la fiecare rang fiind conectată la intrarea de transport de la rangul imediat următor:



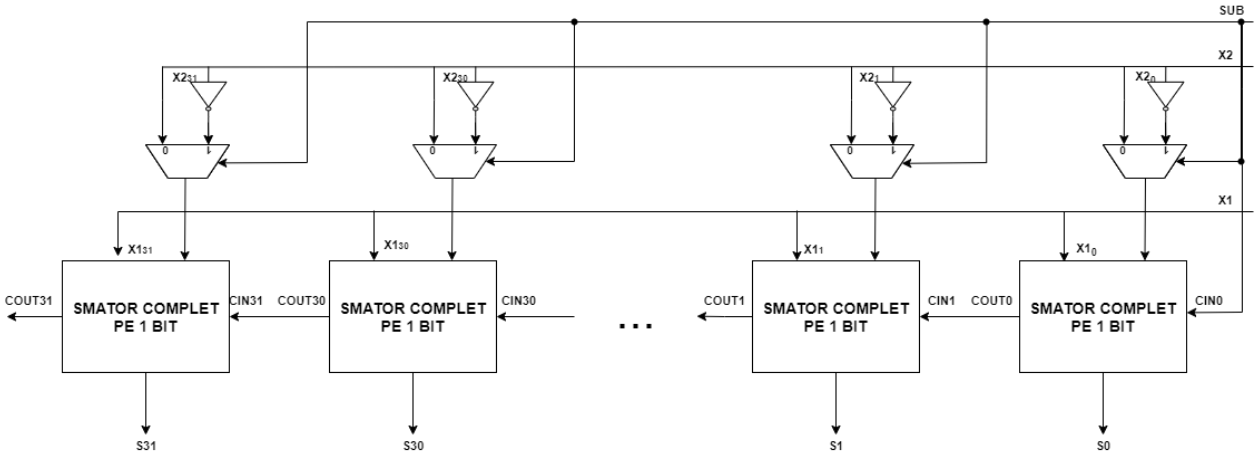
Figură V-1 2 Sumator complet pe 32 de biți cu propagarea transportului

b. *Scăderea*

Operația de scădere se va realiza folosind aceeași componentă precum adunarea, exploatând astfel avantajele reprezentării numerelor în formatul complementului față de 2.

Diferența față de operația de adunare va fi faptul că cel de-al doilea operand se va nega mai întâi, iar abia apoi valoarea acestuia va fi adăugată valorii primului operand. Astfel, se vor folosi 32 de inversoare pentru negarea fiecărui bit în parte al scăzătorului și totodată 32 de multiplexoare 2:1, ale căror selecție este reprezentată de semnalul de control *SUB*, care va avea valoarea 1 logic în momentul execuției operației de scădere. Totodată, semnalul *SUB* va fi propagat și pe intrarea de Carry a sumatorului corespunzător pozițiilor cel mai puțin semnificative pentru a asigura incrementarea valorii complementului față

de 1, obținut prin negarea tuturor biților scăzătorului, astfel încât să se obțină complementul față de 2 al acestuia.



Figură V-1 3 Sumator scăzător pe 32 de biți cu propagarea transportului

c. *Incrementarea*

Operația de incrementare reprezintă o adunare cu valoarea 1 a primului operand, așadar logica operației va rămâne neschimbată față de cea a adunării, acestea realizându-se folosind aceeași componentă, mai sus descrisă, cu deosebirea că este necesară setarea registrului pentru cel de-al doilea operand la valoarea 1, reprezentată pe 32 de biți. Starea se va realiza folosind semnalul de control dedicat, *LOAD_OP2_OR_1*, care, ajuns pe intrarea de selecție unui multiplexor dacă în registrul corespunzător celui de-al doilea operand va fi încărcat operandul efectiv sau valoarea *00000001H*.

d. *Decrementarea*

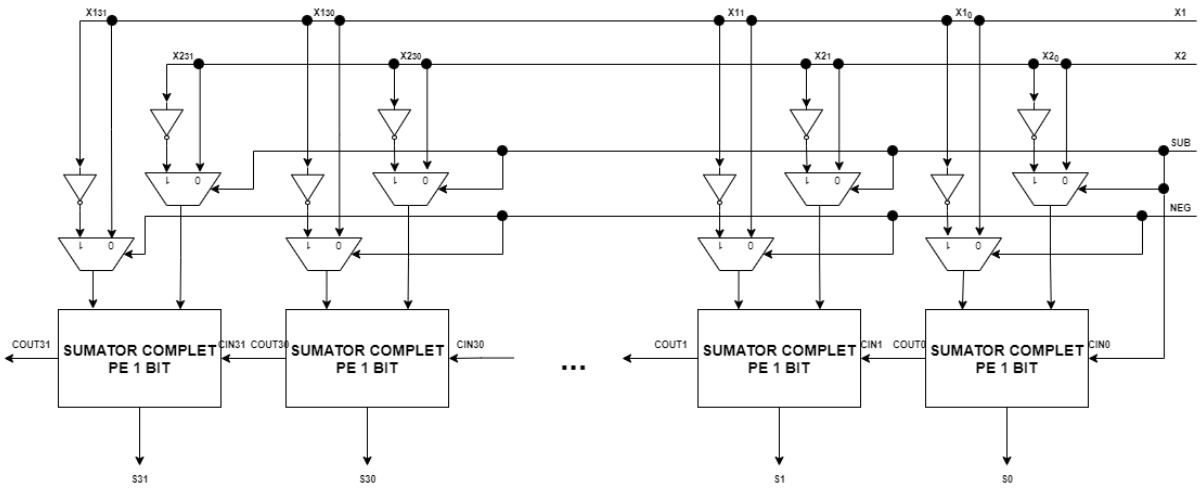
Operația de decrementare se definește în mod analog cu operația de incrementare, fiind la bază o scădere în care scăzătorul are valoarea 1.

Setarea valorii pentru cel de-al doilea operand la *00000001H* se va realiza, de asemenea, cu același semnal de control folosit și în cadrul operației de incrementare, deosebirea constând în faptul că în cazul operației de decrementare semnalul de control *SUB* va avea valoarea 1 logic.

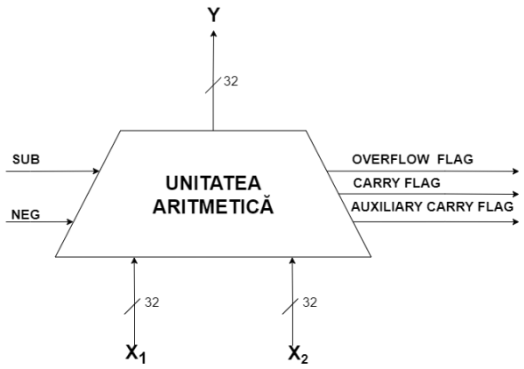
e. *Negarea*

Operația de negare se va realiza prin negarea tuturor biților și incrementarea valorii obținute. Aceasta urmează un mecanism similar cu cel al scăderii, cu deosebirea că operațiile efectuate pe primul operand. Așadar, se va aplica un mecanism similar, prin adăugarea a 32 de multiplexoare 2:1, care vor selecta dacă pe intrările sumatoarelor vor ajunge biții operandului X1 sau valorile negate ale acestora. Selecția se va face prin intermediul semnalului de control dedicat *NEG*, care va avea valoarea 1 în timpul execuției operației de negare. De asemenea, semnalul de control *LOAD_OP2_OR_1* va avea valoarea 1 logic pentru a asigura încărcarea valorii *00000001H* în registrul pentru cel de-al doilea operand, întrucât valoarea complementului față de 1 obținut prin

negarea tuturor biților operandului trebuie incrementată pentru a obține opusul acestuia.



Figură V-1 4 Unitatea aritmetică pe 32 de biți

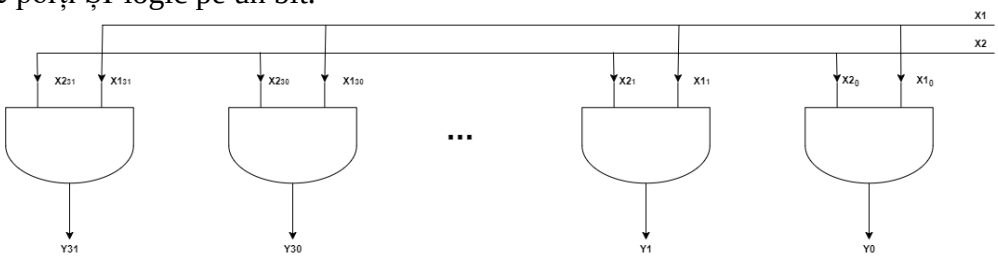


Figură V-1 5 Bloc unitatea aritmetică pe 32 de biți

2. *Operații logice pe biți*

a. *ȘI-logic pe biți*

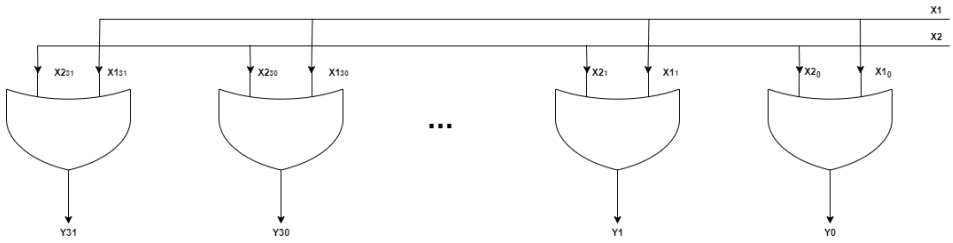
După cum a fost descrisă în cadrul secțiunii de analiză, operația ȘI-logic se va executa pe perechile de biți de pe aceeași poziție, folosind, prin urmare, 32 de porți ȘI-logic pe un bit.



Figură V-2 1 Porți ȘI pentru operația ȘI-logic pe 32 de biți

b. *SAU-logic pe biți*

În mod total analog, operația SAU-logic pe biți va implica utilizarea a 32 de porți SAU-logic, pe intrările cărora vor ajunge biții aflați pe aceleași poziții în reprezentările operanzilor.



Figură V-2 2 Porți SAU pentru operația SAU-logic pe 32 de biți

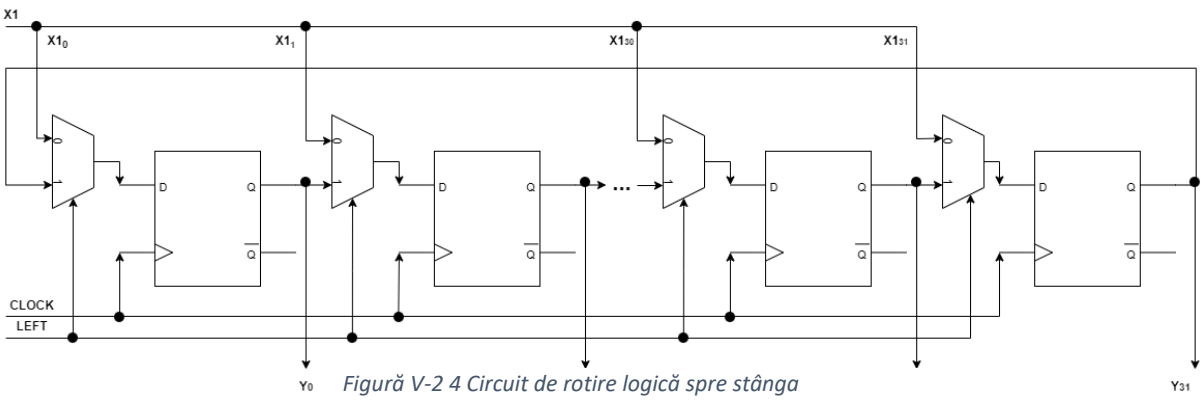
c. *NU-logic pe biți*

Operația de NU-logic pe biți va presupune negarea tuturor biților componenți ai primului operand, folosind 32 de inversoare pe 1 bit.

Figură V-2 3 Porți NU pentru operația NU-logic pe 32 de biți

d. *Rotire logică spre stânga*

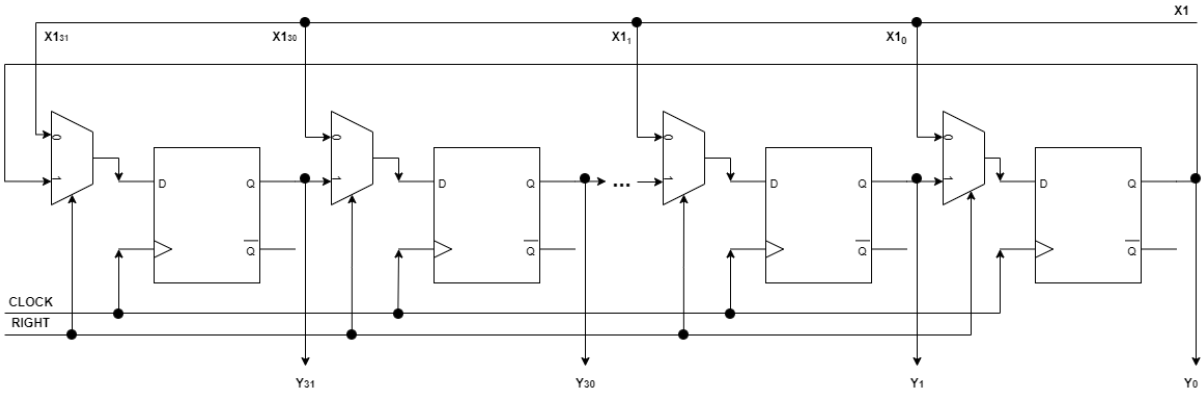
Rotirea logică spre stânga se va realiza prin interconectarea a 32 de celule de memorie reprezentate prin circuite bistabile de tip D, ale căror ieșiri seriale se vor lega la intrările seriale ale bistabilelor destinate pozițiilor imediat mai semnificative, cu excepția ieșirii bistabilului corespunzător poziției celei mai semnificative, care va fi conectată la intrarea bistabilului aferent poziției celei mai puțin semnificative. Pe intrările paralele ale circuitului de rotire vor ajunge biții din reprezentarea operandului, iar selecția dintre intrările seriale și paralele se va realiza folosind 32 de multiplexoare 2:1, controlate de semnalul de control *LEFT*, activ pe 0 la încărcarea biților operandului în celulele de memorie. Astfel, la sosirea impulsului de ceas, dacă semnalul *LEFT* este activ pe 1 logic, va avea loc rotirea operandului cu o poziție spre stânga.



Figură V-2 4 Circuit de rotire logică spre stânga

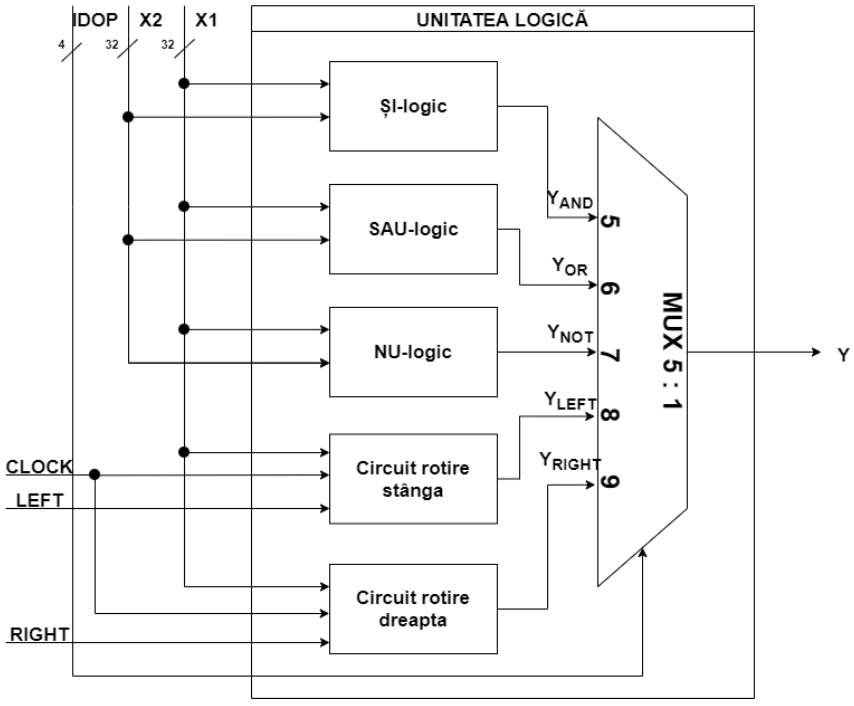
e. *Rotire logică spre dreapta*

Rotirea logică spre dreapta abordează o strategie asemănătoare cu cea a rotirii logice spre dreapta, cu deosebirea că sensul de deplasare se va inversa. Astfel ieșirile seriale se vor conecta la intrările seriale ale bistabilelor corespunzătoare pozițiilor imediat mai puțin semnificative, iar ieșirea bistabilului pentru poziția cel mai puțin semnificativă se va lega la intrarea serială a bistabilului pentru poziția cea mai semnificativă. Selecția între intrările seriale și paralele se va realiza folosind 32 de multiplexoare 2:1, controlate de semnalul *RIGHT*, raționamentul rămânând neschimbat față de rotirea logică spre stânga.



Figură V-2 5 Circuit de rotire logică spre dreapta

Astfel, schema bloc a unității logice pe 32 de biți poate fi reprezentată după cum urmează:



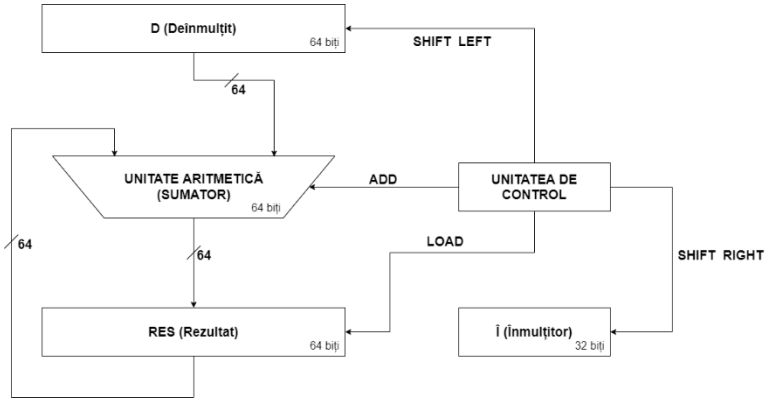
Figură V-2 6 Schema bloc pentru unitatea logică

3. *Operații suplimentare (înmulțire și împărțire)*

a. *Înmulțirea*

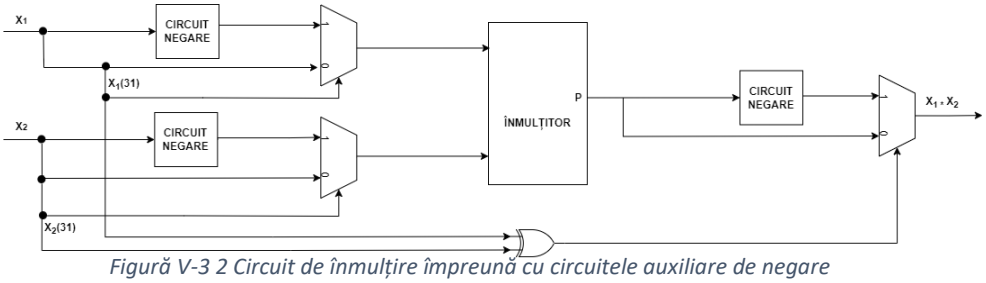
Urmărind etapele algoritmului *Shift-And-Add Multiplication*, descrise în cadrul capitolului de analiză, traducerea execuției algoritmului în limbaj hardware este una imediată: inițial, registrul RES pe 64 de biți, corespunzător rezultatului este setat la 0. Registrul RES este reprezentat folosind 64 de biți pentru a putea susține ideea de bază a algoritmului, care, la fiecare pas, deplasează spre stânga conținutul registrului D, definit tot pe 64 de biți și destinat memorării deîmpărțitului. Inițial X_1, reprezentând deîmpărțitul va fi stocat în jumătatea inferioară a registrului D. Conținutul registrului Î, corespunzător înmulțitorului va fi inițializat cu valoarea X_2, iar în numărătorul N va fi încărcată valoarea 32. Bitul cel mai puțin semnificativ al registrului Î, și anume $\hat{I}_0$, va determina practic dacă valoarea deînmulțitului va fi sau nu adăugată la valoarea stocată în registrul RES, corespunzător produsului. Deplasarea spre stânga a conținutului registrului înmulțitor va avea efectul deplasării produselor intermediare spre stânga cu o poziție binară, iar deplasarea spre dreapta a conținutului registrului D va permite luarea în considerare a fiecărei cifre a deînmulțitului, de la stânga la dreapta, similar algoritmului de înmulțire zecimală.

Legăturile între componentele mai sus menționate pot fi vizualizate în schema logică de mai jos:



Figură V-3 1 Circuit de înmulțire

Întrucât algoritmul de înmulțire ales operează doar pe numere întregi pozitive, se dovedește a fi necesară utilizarea unor circuite suplimentare de negare care să asigure faptul că la înmulțitor ajung valorile absolute ale operanzilor și că rezultatul împărțirii este un număr negativ, dacă cei doi termeni ai produsului au semne diferite, după cum urmează:

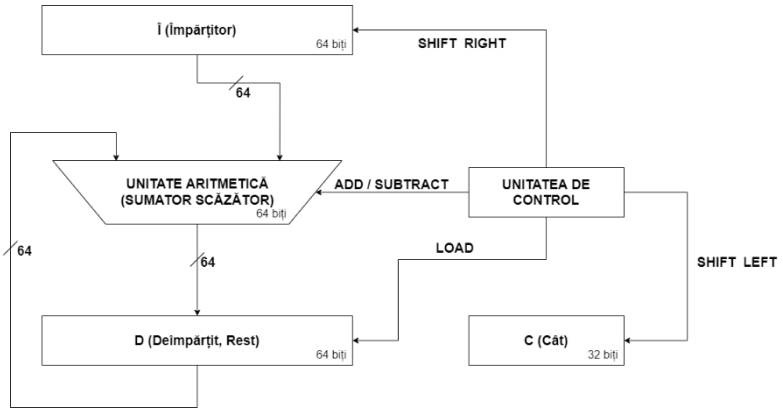


Figură V-3 2 Circuit de înmulțire împreună cu circuitele auxiliare de negare

b. *Împărțirea*

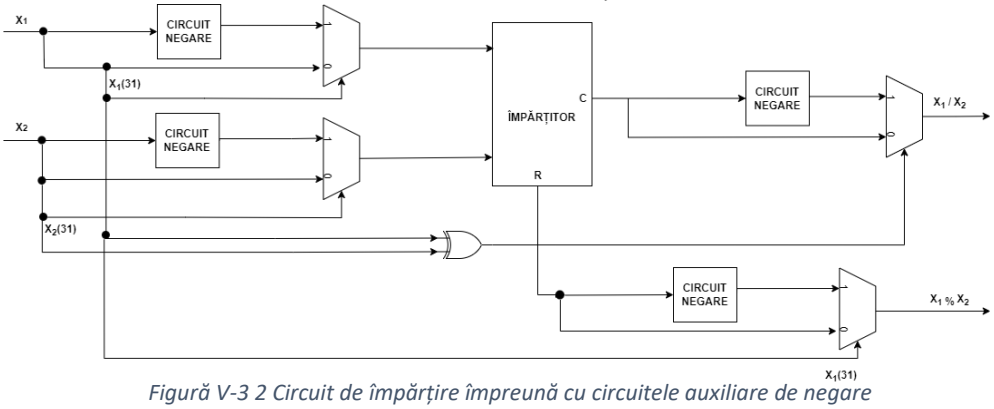
Implementarea algoritmului de împărțire *Restoring Division* necesită identificarea componentelor hardware necesare satisfacerii etapelor de execuție ale algoritmului, traduse în sfera hardware astfel: inițial, deîmpărțitul este încărcat în registrul D, iar împărțitorul în registrul Î, ambele definite pe 64 de biți. Registrul pe 32 de biți C, corespunzător câtului împărțirii va fi setat la 0, iar numărătorul la valoarea 33. Pentru a putea determina dacă împărțitorul este sau nu mai mic decât restul parțial, conținutul registrului Î va fi scăzut din conținutul registrului D, în care se stochează resturile parțiale obținute. Dacă rezultatul este unul negativ, pasul următor îl reprezintă restabilirea valorii precedente prin adăugarea valorii stocate în Î la valoarea restului parțial și generarea unui 0 pe poziția cel mai puțin semnificativă în registrul C, C_0. Dacă rezultatul operației de scădere este unul negativ, se va genera o valoare de 1 pe poziția C_0. În etapa următoare, împărțitorul este deplasat spre dreapta cu o poziție binară, având astfel loc alinierea sa cu conținutul registrului deînmulțit D pentru următoarea iterație a algoritmului.

Așadar, componentele hardware ale circuitului de împărțire binară, cât și legăturile dintre acestea arată în felul următor:



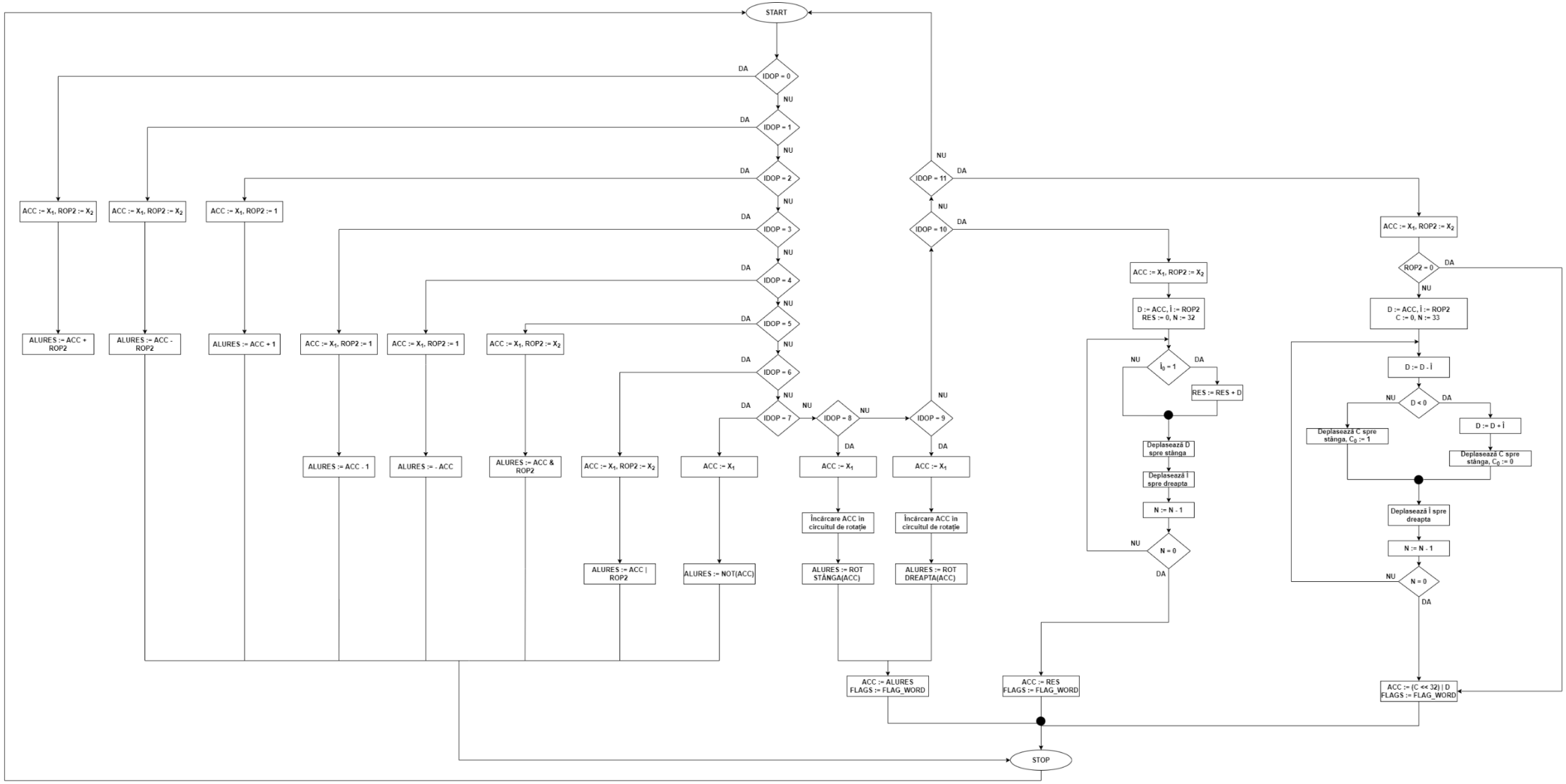
Figură V-3 3 Circuit de împărțire

Similar cu problema întâlnită la proiectarea circuitului de înmulțire, și în cazul operației de împărțire este necesară utilizarea unor circuite de negare care să asigure faptul că operanzii sunt pozitivi, iar rezultatele obținute au semnele corespunzătoare, câtul fiind negativ dacă deîmpărțitul și împărțitorul sunt de semne contrare, iar restul având semnul deîmpărțitului:



Figură V-3 2 Circuit de împărțire împreună cu circuitele auxiliare de negare

În figura de mai jos pot fi observate operațiile efectuate de unitatea de execuție în cadrul fiecărei stări posibile în care circuitul se poate afla:



Figură V 4 Operațiile efectuate de unitatea de execuție în cadrul fiecărei stări definite de către unitatea de control

VI. Implementare

Implementarea componentelor unității aritmetice și logice urmărește întocmai descrierile circuitelor respective din cadrul capitolului de analiză, abordând o manieră structurală în ceea ce privește circuitele ce descriu operațiile din setul de funcționalități și o manieră comportamentală pentru componentele auxiliare, precum unitatea de control și memoria de instrucțiuni.

Codul VHDL ce descrie principalele componente ale unității aritmetice și logice poate fi consultat în cele ce urmează.

1. Unitatea aritmetică

Unitatea aritmetică este circuitul responsabil de efectuarea operațiilor aritmetice, și anume adunarea, scăderea, incrementarea și decrementarea, implementate conform specificațiilor detaliate în capitolul de design, având următoarea listă de porturi:

```
entity arithmetic_unit is
    Port ( x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           sub : in STD_LOGIC;
           neg : in STD_LOGIC;
           y : out STD_LOGIC_VECTOR (31 downto 0);
           flags : out STD_LOGIC_VECTOR(6 downto 0));
end arithmetic_unit;
```

Figură VI 1 Codul VHDL pentru entitatea unității aritmetice

Porturile unității aritmetice, mai sus menționate, se împart în trei categorii importante, porturi pentru intrări de date, porturi pentru intrări de control, porturi pentru ieșiri de date, având următoarele semnificații:

- *x1* – intrare pentru primul operand, reprezentat pe 32 de biți;
- *x2* – intrare pentru cel de-al doilea operand, reprezentat pe 32 de biți;
- *sub* – intrare de control, activ pe 1-logic în cadrul operațiilor care presupun efectuarea unei scăderi de către unitatea aritmetică (scădere, decrementare);
- *neg* – intrare de control, care asigură efectuarea operației de negare (determinarea opusului) a primului operand;
- *y* – rezultatul operației aritmetice;
- *flags* – cuvântul ce conține valorile flag-urilor setate în urma efectuării operației aritmetice.

Arhitectura unității aritmetice este descrisă în manieră structurală, conținând declarații și instanțieri ale componentelor necesare, conform schemei prezentate în *Figura V-1 4*, acestea fiind implementate conform descrierilor prezentate în cadrul etapei de proiectare.

2. *Unitatea Logică*

Unității logice îi revine sarcina efectuării operațiilor logice descrise în specificațiile de proiectare: ȘI-logic, SAU-logic, NU-logic, rotire logică spre stânga și rotire logică spre dreapta.

Structura intrărilor și ieșirilor asociate unității logice este următoarea:

```
entity logic_unit is
    Port ( x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           idop : in STD_LOGIC_VECTOR (3 downto 0);
           left : in STD_LOGIC;
           right : in STD_LOGIC;
           clk : in STD_LOGIC;
           y : out STD_LOGIC_VECTOR (31 downto 0);
           flags : out STD_LOGIC_VECTOR(6 downto 0));
end logic_unit;
```

Figură VI 2 Codul VHDL pentru entitatea unității logice

Porturile unității logice sunt, de asemenea, clasificate în intrări de date, intrări de control și ieșiri de date:

- *x1* – intrare pentru primul operand, reprezentat pe 32 de biți;
- *x2* – intrare pentru cel de-al doilea operand, reprezentat pe 32 de biți;
- idop – intrare de control reprezentând identificatorul unic al operației ce se va efectua în unitatea logică, asigurând selecția rezultatului produs în urma operației respective;
- left – intrare de control responsabilă de efectuarea operației de rotire spre stânga;
- right – intrare de control ce asigură rotirea logică spre dreapta a celui de-al doilea operand;
- clk – intrare pentru semnalul de ceas necesar bistabilelor de tip D ce alcătuiesc circuitele pentru operațiile de rotire logică spre stânga și spre dreapta;
- *y* – rezultatul operației logice;
- *flags* – cuvântul ce conține valorile flag-urilor setate în urma efectuării operației logice.

Asemeni unității aritmetice, arhitectura unității logice prezintă o descriere structurală, declarând și instanțiind componentele responsabile de efectuarea operațiilor logice: circuit care efectuează operația ȘI-logic pe biți, circuit pentru SAU-logic pe biți, circuit pentru negarea tuturor biților din reprezentarea operandului, circuite de rotire logică spre stânga și spre dreapta. Rezultatul furnizat de unitatea logică este selectat folosind un multiplexor 5 : 1, cu calea de date pe 32 de biți, conform schemei din *Figura V-2 6*. Componentele unității aritmetice au fost implementate conform descrierilor și schemelor detaliate în cadrul capitolului de *Design*.

3. *Înmulțitorul*

Circuitul pentru înmulțire implementează algoritmul *Shift-and-Add Multiplication* de înmulțire binară a două numere pozitive, reprezentate pe 32 de biți, produsul obținut fiind un număr pozitiv, reprezentat pe 32 de biți, dispunând de următoarea structură a intrărilor și a ieșirilor:

```
entity multiply_circuit is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           clr_res_mul : in STD_LOGIC;
           ld_res_mul : in STD_LOGIC;
           ld_i_mul : in STD_LOGIC;
           ld_d_mul : in STD_LOGIC;
           right_shift_i_mul : in STD_LOGIC;
           left_shift_d_mul : in STD_LOGIC;
           y : out STD_LOGIC_VECTOR (63 downto 0);
           i0 : out STD_LOGIC;
           flags : out STD_LOGIC_VECTOR(6 downto 0));
end multiply_circuit;
```

Figură VI 3 Codul VHDL pentru entitatea circuitului de înmulțire binară

Semnificațiile porturilor circuitului pentru înmulțire binară, în funcție de destinația fiecăruia, sunt următoarele:

- *clk* – intrare pentru semnalul de ceas, necesar registrelor pentru operanzi și rezultat;
- *clr* – intrare de control pentru resetarea valorilor registrelor pentru operanzi din componența înmulțitorului;
- *x1* – intrare pentru primul operand, reprezentat pe 32 de biți;
- *x2* – intrare pentru cel de-al doilea operand, reprezentat pe 32 de biți;
- *clr_res_mul* – intrare de control pentru resetare pentru registrul în care se va stoca produsul;
- *ld_res_mul* – intrare de control ce asigură încărcarea paralelă a registrului pentru stocarea rezultatului;
- *ld_i_mul* – intrare de control pentru încărcarea paralelă a registrului destinat înmulțitorului;
- *ld_d_mul* – intrare de control pentru validarea încărcării paralele a registrului pentru deînmulțit;
- *right_shift_i_mul* – intrare de control ce asigură deplasarea logică spre dreapta a valorii din registrul ce stochează înmulțitorul;
- *left_shift_d_mul* – intrare de control pentru deplasarea logică spre stânga a valorii din registrul destinat deînmulțitului;
- *y* – rezultatul operației de înmulțire;
- *i0* – ieșire ce conține bitul cel mai puțin semnificativ al valorii curente din registrul pentru înmulțitor, necesară în unitatea de control în cadrul stării de decizie `Test_I0`;
- *flags* – cuvântul ce conține valorile flag-urilor setate în urma efectuării operației de înmulțire.

Înmulțitorul abordează, de asemenea, o implementare structurală, arhitectura constând din instanțierea registrelor pentru deînmulțit, împărțitor și rezultat, a sumatorului pe 64 de biți și interconectarea acestora, conform schemei din *Figura V-3 1*. Funcționarea și tranziția stărilor intermediare, prin care circuitul trece în vederea obținerii rezultatului, sunt asigurate prin intermediul semnalelor de control generate de unitatea de control.

4. *Împărțitorul*

Împărțitorul implementează algoritmul de împărțire binară cu refacerea restului, Restoring Division, conform schemei logice prezentate în capitolul de analiză și are următoare structură externă de porturi:

```
entity division_circuit is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           clr_c_div : in STD_LOGIC;
           ld_d_div : in STD_LOGIC;
           sub_div : in STD_LOGIC;
           ld_i_div : in STD_LOGIC;
           c0 : in STD_LOGIC;
           right_shift_i_div : in STD_LOGIC;
           left_shift_c_div : in STD_LOGIC;
           y : out STD_LOGIC_VECTOR (63 downto 0); -- (C << 32) | R
           flags : out STD_LOGIC_VECTOR(6 downto 0);
           dn : out STD_LOGIC);
end division_circuit;
```

Figură VI 4 Codul VHDL pentru entitatea circuitului de împărțire binară

Porturile circuitului de împărțire binară sunt definite după cum urmează:

- *clk* – intrare pentru semnalul de ceas, necesar registrelor pentru operanzi și rezultat;
- *clr* – intrare de control pentru resetarea valorilor registrelor pentru operanzi din componența înmulțitorului;
- *x1* – intrare pentru deîmpărțit, reprezentat pe 32 de biți;
- *x2* – intrare pentru împărțotpr, reprezentat pe 32 de biți;
- *clr_c_mul* – intrare de control pentru resetare pentru registrul în care se va stoca câtul împărțirii;
- *ld_d_div* – intrare de control pentru validarea încărcării paralele a registrului pentru deîmpărțit;
- *sub_div* – intrare de control care comandă sumatorul-scăzător din componența înmulțitorului astfel încât să realizeze o operație de scădere;
- *ld_i_div* – intrare de control pentru încărcarea paralelă a registrului destinat împărțitorului;
- *c0* – intrare de control, care indică valoarea celui mai puțin semnificativ bit din reprezentarea curentă a câtului, necesară în cadrul stărilor `Left_Shift_C0` și `Left_Shift_C1`;
- *right_shift_i_div* – intrare de control ce asigură deplasarea logică spre dreapta a valorii din registrul ce stochează împărțitorul;

- *left_shift_c_div* – intrare de control pentru deplasarea logică spre stânga a valorii din registrul destinat câtului împărțirii;
- *y* – rezultatul operației de înmulțire, conținând în jumătatea *HIGH* câtul și în cea *LOW*, restul împărțirii;
- *flags* – cuvântul ce conține valorile flag-urilor setate în urma efectuării operației de împărțire;
- *dn* – ieșire de control, corespunzătoare bitului de semn al deînmulțitului, a cărui valoare va fi testată în unitatea de control în cadrul stării `Test_D_Neg`.

Circuitul de împărțire este descris structural, arhitectura acestuia constând din instanțierea registrelor pentru stocarea deîmpărțitului și restului, a împărțitorului și a câtului și a sumatorului-scăzător, interconectate conform *Figurii V-3 2* și controlate prin intermediul semnalelor de control.

5. *Unitatea de execuție*

Unitatea de execuție, fiind implementată structural, cuprinde toate componentele mai sus descrise, furnizând rezultatul operației curente sub forma unui număr binar reprezentat pe 64 de biți în complement față de 2.

Structura porturilor unității de execuție este următoare, semnificațiile acestora fiind deja detaliate în cadrul descrierii componentelor anterior expuse:

```
entity execution_unit is
    Port ( x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           res_sel : in STD_LOGIC_VECTOR (1 downto 0);
           idop : in STD_LOGIC_VECTOR (3 downto 0);
           sub : in STD_LOGIC;
           neg : in STD_LOGIC;
           left : in STD_LOGIC;
           right : in STD_LOGIC;
           -- semnale de control pentru fsm_ctrl inmultitor
           clr_res_mul : in STD_LOGIC;
           ld_res_mul : in STD_LOGIC;
           ld_i_mul : in STD_LOGIC;
           ld_d_mul : in STD_LOGIC;
           right_shift_i_mul : in STD_LOGIC;
           left_shift_d_mul : in STD_LOGIC;
           -- semnale de control pentru fsm_ctrl impartitor
           clr_c_div : in STD_LOGIC;
           ld_d_div : in STD_LOGIC;
           sub_div : in STD_LOGIC;
           ld_i_div : in STD_LOGIC;
           c0 : in STD_LOGIC;
           right_shift_i_div : in STD_LOGIC;
           left_shift_c_div : in STD_LOGIC;
           dn : out STD_LOGIC;
           i0 : out STD_LOGIC;
           flags : out STD_LOGIC_VECTOR(6 downto 0);
           y : out STD_LOGIC_VECTOR (63 downto 0));
end execution_unit;
```

Figură VI 5 Codul VHDL pentru entitatea unității de execuție

Întrucât unitatea aritmetică și unitatea logică furnizează ca rezultate numere reprezentate pe 32 de biți, iar înmulțitorul și împărțitorul au nevoie de 64 de biți pentru reprezentarea rezultatului, unitatea de execuție conține suplimentar și circuite de extindere cu 0 și cu semn, folosite pentru a extinde rezultatele pe 32 de biți la reprezentarea echivalentă, pe 64 de biți.

```
entity zero_extend is
    Port ( data_in : in STD_LOGIC_VECTOR (31 downto 0);
           extended_data : out STD_LOGIC_VECTOR (63 downto 0));
end zero_extend;
```

Figură VI 6 Codul VHDL pentru entitatea circuitului de extindere cu 0

```
entity sign_extend is
    Generic ( initial_size : integer := 32;
              extended_size : integer := 64);
    Port ( data_in : in STD_LOGIC_VECTOR (initial_size - 1 downto 0);
           extended_data : out STD_LOGIC_VECTOR (extended_size - 1 downto 0));
end sign_extend;
```

Figură VI 7 Codul VHDL pentru entitatea circuitului de extindere cu semn

De asemenea, luând în considerare faptul că algoritmii de înmulțire și împărțire binară utilizați operează doar pe numere pozitive, unitatea de execuție folosește mai multe circuite de negare, astfel încât să se asigure faptul că operanzii ajunși la circuitele de înmulțire și împărțire sunt pozitivi, iar rezultatele obținute au semnul corespunzător: produsul și câtul sunt pozitive dacă operanzii au același semn, în caz contrar fiind negative, iar restul împărțirii are același semn cu deîmpărțitul.

```
entity complement_circuit is
    Generic (n : integer := 32);
    Port ( x : in STD_LOGIC_VECTOR (n - 1 downto 0);
           y : out STD_LOGIC_VECTOR (n - 1 downto 0));
end complement_circuit;
```

Figură VI 8 Codul VHDL pentru entitatea circuitului de negare

6. *Unitatea de control*

Unitatea de control este implementată într-o manieră comportamentală, arhitectura sa conținând două procese: un proces pentru generarea stării următoare, în funcție de starea curentă și, unde este cazul, în funcție de intrări, și un proces pentru generarea combinației semnalelor de control asociate fiecărei stări. Tranzițiile de stări pe care circuitul o urmează este reprezentată în *Figura V 2*, mulțimea tuturor stărilor posibile fiind definită ca un tip enumerat.

Structura externă a unității de control poate fi vizualizată în figura de mai jos, semnificațiile porturilor acesteia fiind detaliate în descrierea entităților unității aritmetice, a unității logice, a înmulțitorului și împărțitorului:

```
entity control_unit is
    Port ( idop : in STD_LOGIC_VECTOR (3 downto 0);
           clk : in STD_LOGIC;
           start : in STD_LOGIC;
           clr : in STD_LOGIC;
           clr_div : in STD_LOGIC;
           clr_mul : in STD_LOGIC;
           i0 : in STD_LOGIC; -- pentru inmultire (inmultitor(0))
           dn : in STD_LOGIC; -- pentru impartite (deimpartit(31))
           zero : in STD_LOGIC;
           -- semnale de control pentru operatii logoce si aritmetice
           sub : out STD_LOGIC;
           neg : out STD_LOGIC;
           left : out STD_LOGIC;
           right : out STD_LOGIC;
           load_acc : out STD_LOGIC;
           load_res_or_op : out STD_LOGIC;
           load_rop2 : out STD_LOGIC;
           load_op2_or_1 : out STD_LOGIC;
           load_flags : out STD_LOGIC;
           stop : out STD_LOGIC;
           -- semnale de control pentru fsm_ctrl inmultitor
           clr_res_mul : out STD_LOGIC;
           ld_res_mul : out STD_LOGIC;
           ld_i_mul : out STD_LOGIC;
           ld_d_mul : out STD_LOGIC;
           right_shift_i_mul : out STD_LOGIC;
           left_shift_d_mul : out STD_LOGIC;
           -- semnale de control pentru fsm_ctrl impartitor
           clr_c_div : out STD_LOGIC;
           ld_d_div : out STD_LOGIC;
           sub_div : out STD_LOGIC;
           ld_i_div : out STD_LOGIC;
           c0 : out STD_LOGIC;
           right_shift_i_div : out STD_LOGIC;
           left_shift_c_div : out STD_LOGIC;
           res_sel : out STD_LOGIC_VECTOR (1 downto 0));
end control_unit;
```

Figură VI 9 Codul VHDL pentru entitatea unității de control

7. *Unitatea aritmetico-logică*

Unitatea aritmetico-logică instanțiază și interconectează componentele pentru unitatea de execuție și unitatea de control, registrul acumulator (64 biți), registrul pentru cel de-al doilea operand (32 biți) și registrul de flag-uri (7 biți), dispunând de următoarea listă de porturi:

```
entity alu32 is
    Port ( x1 : in STD_LOGIC_VECTOR (31 downto 0);
           x2 : in STD_LOGIC_VECTOR (31 downto 0);
           idop : in STD_LOGIC_VECTOR (3 downto 0);
           start : in STD_LOGIC;
           clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           stop : out STD_LOGIC;
           flags_word : out STD_LOGIC_VECTOR (6 downto 0);
           y : out STD_LOGIC_VECTOR (63 downto 0));
end alu32;
```

Figură VI 10 Codul VHDL pentru entitatea unității de control

8. *Contorul de program*

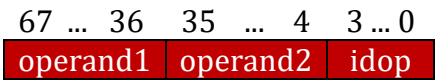
Contorul de program reprezintă un numărător pe 32 de biți ce indică adresa instrucțiunii curente din memoria de instrucțiuni, acesta fiind implementat comportamental, folosind un singur proces secvențial și având următoarele porturi de intrare-ieșire:

```
entity counter32 is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           en : in STD_LOGIC;
           q : out STD_LOGIC_VECTOR (31 downto 0));
end counter32;
```

Figură VI 11 Codul VHDL pentru entitatea contorului de program

9. *Memoria de instrucțiuni*

Memoria de instrucțiuni este o memorie de tip Read Only Memory (ROM) ce are scopul de a stoca instrucțiuni destinate testării funcționalităților implementate prin operațiile asociate unității aritmetice și logice. Astfel, a fost definit un format de instrucțiune pe 68 de biți, având următoarele câmpuri binare:



Figură VI 12 Formatul instrucțiunii de test

Prin urmare, a fost implementată o memorie ROM de dimensiune 256 × 68, ce conține instrucțiuni din repertoriul unității aritmetice implementate și are următoarea structură externă:

```
entity rom256x68 is
    Port ( address : in STD_LOGIC_VECTOR (31 downto 0);
           data_out : out STD_LOGIC_VECTOR (67 downto 0));
end rom256x68;
```

Figură VI 13 Codul VHDL pentru entitatea memoriei de instrucțiuni

Porturile memoriei de instrucțiuni sunt porturile fundamentale întâlnite la orice memorie de tip ROM:

- *address* – intrare de adresă, reprezentând indexul instrucțiunii curente de test;
- *data_out* – ieșire de date, reprezentând instrucțiunea curentă de test, definită, conform figurii VI 5, pe 68 de biți.

Conținutul curent al memoriei de instrucțiuni de test este următorul:

```
signal rom: rom_array := (
    x"0000000A_00000010_0",          -- add (10, 16) = 26 = 0000001Ah
    x"00000006_0000000A_0",          -- add (6, 10) = 16 = 00000010h
    x"FFFFFFFA_FFFFFFF0_0",          -- add (-6, -16) = -22 = FFFFFFEAh
    x"7FFFFFFF_6FFFFFFF_0",          -- add (2147483647, 1879048191) = -268435458 = EFFFFFFEh (overflow)
    x"0000000A_00000010_1",          -- sub (10, 16) = -6 = FFFFFFFAh
    x"0000000A_00000012_1",          -- sub (10, 18) = -8 = FFFFFFF8h
    x"80000000_00000002_1",          -- sub (-2147483648, 2) = 2147483646 = 7FFFFFFEh (overflow)
    x"0000000A_00000000_2",          -- inc (10) = 11 = Bh -bun
    x"80000002_00000000_2",          -- inc (-2147483646) = -2147483645 = 80000003h
    x"FFFFFFF8_00000000_3",          -- dec (-8) = -9 = FFFFFFF7h
    x"80000000_00000000_3",          -- dec (-2147483648) = 2147483647 = 7FFFFFFFh
    x"0000000A_00000000_4",          -- neg (10) = -10 = FFFFFFF6h
    x"FFFFFFF6_00000000_4",          -- neg (-10) = 10 = 0000000Ah
    x"0000000A_00000009_5",          -- and (10, 9) = 8h -bun
    x"ABC500F1_0A536221_5",          -- and (ABC500F1h, 0A536221h) = 0A410021h
    x"0000000A_00000009_6",          -- or (10, 9) = 11 = Bh
    x"ABC500F1_0A536221_6",          -- and (ABC500F1h, 0A536221h) = ABD762F1h
    x"0000000A_00000000_7",          -- not (10) = -11 = FFFFFFF5h
    x"FFFFFFF5_00000000_7",          -- not (-11) = 10 = 0000000Ah
    x"80000000_00000000_8",          -- rot_left (80000000) = 00000001h
    x"0000000B_00000000_8",          -- rot_left (11) = 22 = 00000016h
    x"00000010_00000000_9",          -- rot_right (16) = 8 = 00000008h
    x"0000000B_00000000_9",          -- rot_right (11) = 80000005h
    x"0000000A_00000006_A",          -- mul (10, 6) = 60 = 0000003Ch
    x"FFFFFFF6_00000006_A",          -- mul (-10, 6) = -60 = FFFFFFC4h
    x"0000000A_FFFFFFFA_A",          -- mul (10, -6) = -60 = FFFFFFC4h
    x"FFFFFFF6_FFFFFFFA_A",          -- mul (-10, -6) = 60 = 0000003Ch
    x"0000000A_00000000_A",          -- mul (10, 0) = 00000000h
    x"7FFFFFFF_00000003_A",          -- mul(2147483647, 3) = 7FFFFFFDh (overflow)
    x"0000000C_00000006_B",          -- div (12, 6) = 0000000200000000h (c = 2, r = 0)
    x"0000000C_00000005_B",          -- div (12, 5) = 0000000200000002h (c = 2, r = 2)
    x"FFFFFFF4_00000005_B",          -- div (-12, 5) = FFFFFFFEFFFFFFFEh (c = -2, r = -2)
    x"0000000C_FFFFFFFB_B",          -- div (12, -5) = FFFFFFFE00000002h (c = -2, r = 2)
    x"FFFFFFF4_FFFFFFFB_B",          -- div (-12, -5) = 000000002FFFFFFEh (c = 2, r = -2)
    x"0000000C_00000000_B",          -- div (12, 0) = 0000000000000000h (division by 0)
    others => x"00000000_00000000_0"    -- no operation
);
```

Figură VI 13 Conținutul memoriei ROM de instrucțiuni

10. Entitatea principală (top-level)

Entitatea principală a unității aritmetice și logice este implementată structural, conform schemei bloc din *Figura V 1* din cadrul capitolului de design, adăugând unității aritmetico-logice contorul de program și memoria de instrucțiuni pentru test.

Structura entității top-level este următoarea:

```
entity top_level is
    Port ( next_instr_en : in STD_LOGIC;
           clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           reset : in STD_LOGIC;
           start : in STD_LOGIC;
           stop : out STD_LOGIC;
           result_high : out STD_LOGIC_VECTOR (31 downto 0);
           result_low : out STD_LOGIC_VECTOR (31 downto 0);
           flags : out STD_LOGIC_VECTOR (6 downto 0));
end top_level;
```

Figură VI 14 Codul VHDL pentru entitatea top-level

Semnificațiile porturilor de intrare-ieșire pentru entitatea top-level:

- *next_instr_en* – intrare ce validează trecerea la instrucțiunea următoare din memoria de instrucțiuni de test;
- *clk* – intrare pentru semnalul de ceas;
- *clr* – intrare de control pentru resetarea valorilor registrelor pentru operanzi;
- *reset* – intrare ce resetează contorul de program, astfel încât acesta să indice prima instrucțiune de test din memoria ROM;
- *start* – intrare ce validează efectuarea operației;
- *stop* – ieșire ce semnalează finalizarea efectuării operației asociate instrucțiunii curente de test;
- *result_high* – ieșire reprezentând jumătatea mai semnificativă a rezultatului (cei mai semnificativi 32 de biți);
- *result_low* – ieșire reprezentând jumătatea mai puțin semnificativă a rezultatului (cei mai puțin semnificativi 32 de biți);
- *flags* – ieșire conținând valorile flag-urilor.

11. Componente hardware auxiliare

Întrucât finalitatea proiectului o reprezintă programarea unui dispozitiv FPGA, se dovedește necesară implementarea unor componente auxiliare, care să asigure funcționarea corespunzătoare a dispozitivului și să permită vizualizarea rezultatelor de către utilizator.

Dispozitivul ales pentru a fi programat este reprezentat de plăcuța Basys 3, bazată pe familia FPGA Artix 7, componentele hardware auxiliare necesare programării acesteia fiind:

a. Generatorul de monoimpuls sincron

Generatorul de monoimpuls sincron este de a permite o singură activare a unui semnal de, de obicei de tip *enable*, la o apăsare a butonului asociat acestuia, dovedindu-se util în special în cazul butoanelor uzate fizic și excluzând posibilitatea unor activări multiple ale semnalului la apăsarea butonului.

Structura entității generatorului de monoimpuls sincron este următoarea, arhitectura acestuia constând dintr-un numărător și 3 registre pe 1 bit, responsabile de validarea frontului crescător al semnalului de ceas cu scopul de a face posibilă trasarea și testarea fluxului de date și de control al circuitului implementat:

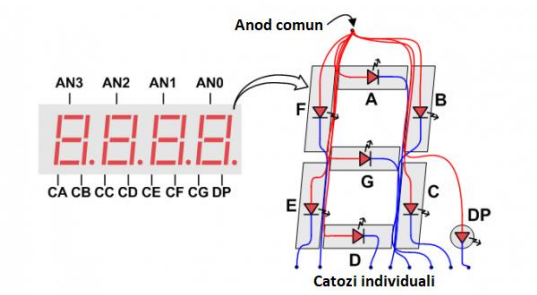
```
entity mpg is
  Port ( clk : in std_logic;
         btn : in std_logic;
         en : out std_logic );
end mpg;
```

Figură VI 15 Codul VHDL pentru entitatea generatorului de monoimpuls sincron

b. *Afișorul pe 7 segmente cu 4 cifre*

Afișorul pe 7 segmente cu 4 cifre de care dispune plăcuța Basys 3, permite vizualizarea fluxului de date și a rezultatelor operațiilor implementate de către unitatea aritemtico-logică, oferind posibilitatea afișării a 4 cifre hexazecimale concomitent.

Această interfață folosește șapte leduri pentru fiecare cifră, iar fiecare cifră este activată de un semnal de anod. Toate semnalele interfeței afișorului pe 7 segmente (7 semnale comune de anod și 4 semnale distincte de catod) sunt active pe 0. Semnalele de catod controlează ledurile care se aprind de pe acele cifre care au semnalul de anod activ.



Figură VI 18 Afișorul pe 7 segmente cu 4 cifre

(sursa: https://digilent.com/reference/programmable-logic/basys-3/reference-manual)

Codul VHDL pentru entitatea afișorului pe 7 segmente cu 4 cifre este următorul:

```
entity display_7seg is
    Port ( digit0 : in STD_LOGIC_VECTOR (3 downto 0);
           digit1 : in STD_LOGIC_VECTOR (3 downto 0);
           digit2 : in STD_LOGIC_VECTOR (3 downto 0);
           digit3 : in STD_LOGIC_VECTOR (3 downto 0);
           clk : in STD_LOGIC;
           cat : out STD_LOGIC_VECTOR (6 downto 0);
           an : out STD_LOGIC_VECTOR (3 downto 0));
end display_7seg;
```

Figură VI 16 Codul VHDL pentru entitatea afișorului pe 7 segmente cu 4 cifre

VII. *Simulare și testare*

Cazurile de testare ale circuitului implementat sunt asociate instrucțiunilor stocate în memoria ROM destinată testării și au fost selectate astfel încât să surprindă cât mai multe situații posibile privind operanzii, urmărindu-se testarea validității rezultatului furnizat de unitatea aritmetico-logică implementată.

Cazurile de testare, corespunzătoare instrucțiunilor de test din memoria ROM, sunt prezentate în tabelul următor:

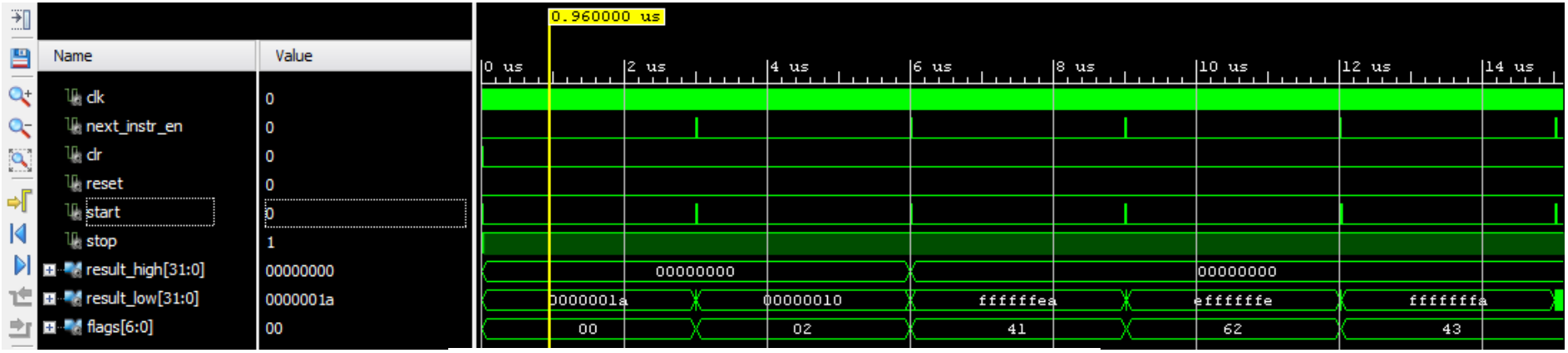
Operand 1		*Operand 2*		*Operație*	*Rezultat așteptat*		*Rezultat obținut*	
Baza 10	***Baza 16***	***Baza 10***	***Baza 16***		***Baza 10***	***Baza 16***	***Baza 10***	***Baza 16***
10	A	16	10	+	26	1A	26	0000001A
6	6	10	A	+	16	10	16	10
-6	FFFFFFFA	-16	FFFFFFF0	+	-22	FFFFFFEA	-22	FFFFFFEA
2147483647	7FFFFFFF	1879048191	6FFFFFFF	+	-268435458	EFFFFFFE	-268435458	EFFFFFFE
10	A	16	10	-	-6	FFFFFFFA	-6	FFFFFFFA
10	A	18	12	-	-8	FFFFFFF8	-8	FFFFFFF8
-2147483648	80000000	2	2	-	2147483646	7FFFFFFE	2147483646	7FFFFFFE
10	A	-	-	(+ 1)	11	B	11	0000000B
-2147483646	80000002	-	-	(+ 1)	-2147483645	80000003	-2147483645	80000003
-8	FFFFFFF8	-	-	(- 1)	-9	FFFFFFF7	-9	FFFFFFF7
-2147483648	80000000	-	-	(- 1)	2147483647	7FFFFFFF	2147483647	7FFFFFFF
10	A	-	-	× (- 1)	-10	FFFFFFF6	-10	FFFFFFF6
-10	FFFFFFF6	-	-	× (- 1)	10	A	10	0000000A
10	A	9	9	and	8	8	8	8

2881814769	ABC500F1	173236769	A536221	and	172032033	A410021	172032033	A410021
10	A	9	9	or	11	B	11	B
2881814769	ABC500F1	173236769	A536221	or	2883019505	ABD762F1	2883019505	ABD762F1
10	A	-	-	not	-11	FFFFFFF5	-11	FFFFFFF5
-11	FFFFFFF5	-	-	not	10	A	10	A
2147483648	80000000	-	-	rotate left	1	1	1	1
11	B	-	-	rotate left	22	16	22	16
16	10	-	-	rotate right	8	8	8	8
11	B	-	-	rotate right	2147483653	80000005	2147483653	80000005
10	A	6	6	×	60	3C	60	3C
-10	FFFFFFF6	6	6	×	-60	FFFFFFC4	-60	FFFFFFC4
10	A	-6	FFFFFFFA	×	-60	FFFFFFC4	-60	FFFFFFC4
-10	FFFFFFF6	-6	FFFFFFFA	×	60	3C	60	3C
10	A	0	0	×	0	0	0	0
2147483647	7FFFFFFF	3	3	×	2147483645	7FFFFFFD	2147483645	7FFFFFFD
12	C	6	6	/	c = 2, r = 0	2 00000000	c = 2, r = 0	2 00000000

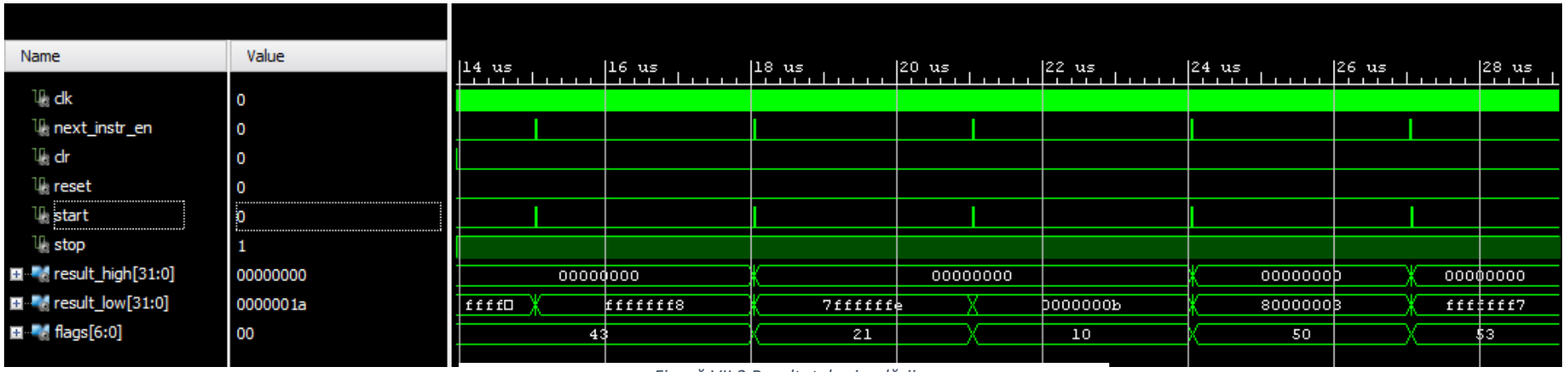
12	C	5	5	/	c = 2, r = 2	2 00000002	c = 2, r = 2	2 00000002
-12	FFFFFFF4	5	5	/	c = -2, r = -2	FFFFFFFE FFFFFFFE	c = -2, r = -2	FFFFFFFE FFFFFFFE
12	C	-5	FFFFFFFB	/	c = -2, r = 2	FFFFFFFE 00000002	c = -2, r = 2	FFFFFFFE 00000002
-12	FFFFFFF4	-5	FFFFFFFB	/	c = 2, r = -2	2 FFFFFFFE	c = 2, r = -2	2 FFFFFFFE
12	C	0	0	/	0 (division by 0)	0 (division by 0)	0 (division by 0)	0 (division by 0)

Tabel VII 1 Cazuri de testare și rezultatele simulării

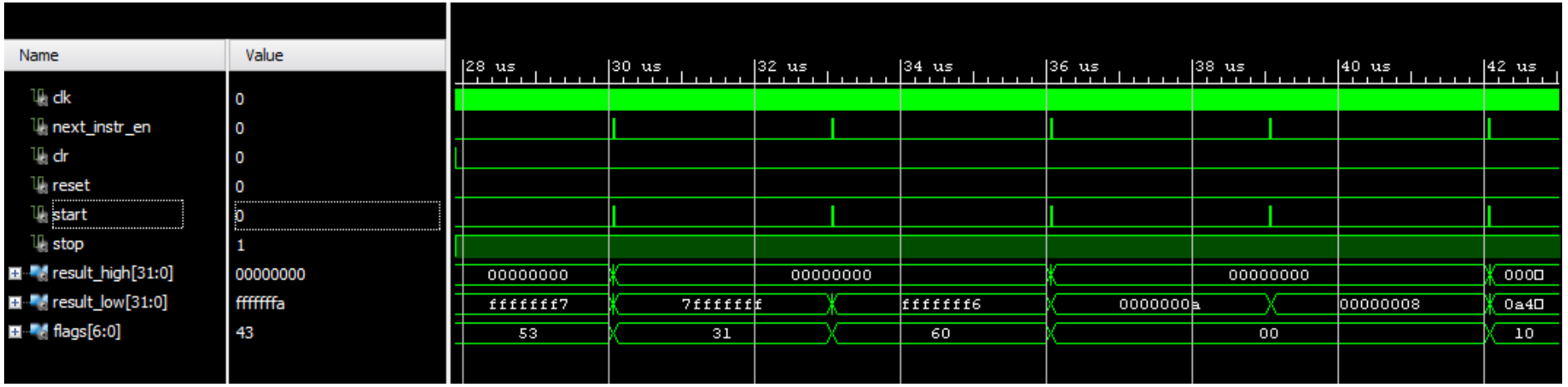
Rezultatele simulării pot fi vizualizate în figurile de mai jos:



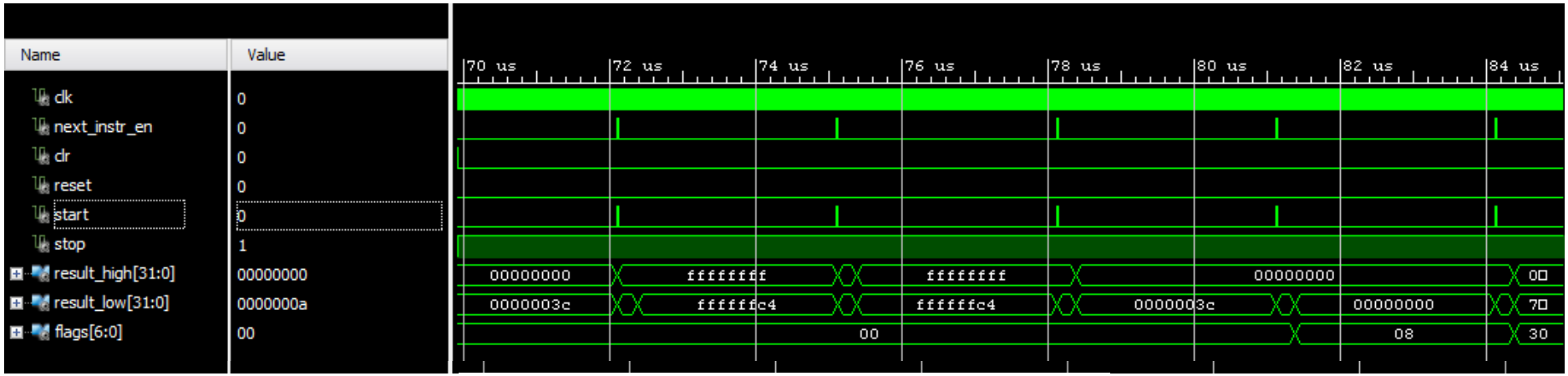
Figură VII 1 Rezultatele simulării



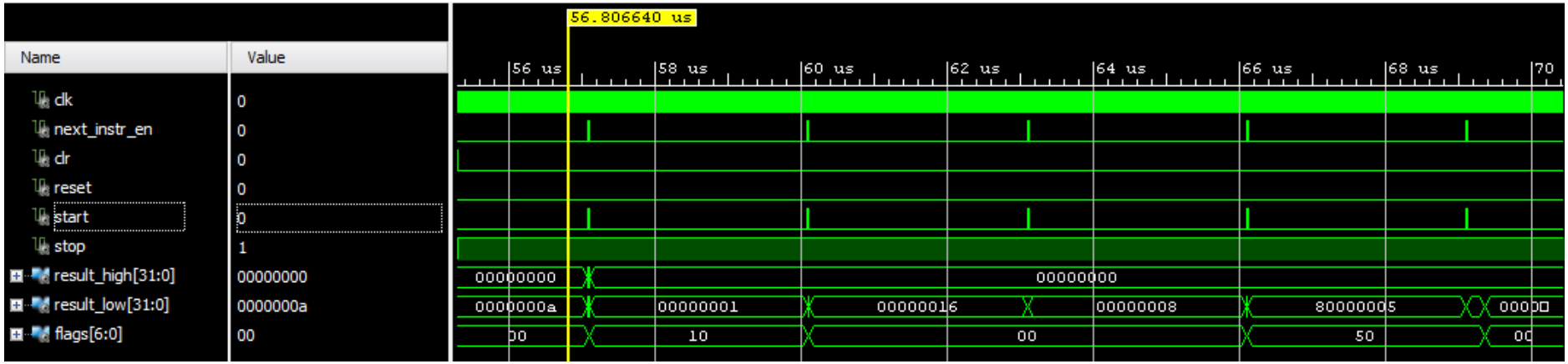
Figură VII 2 Rezultatele simulării



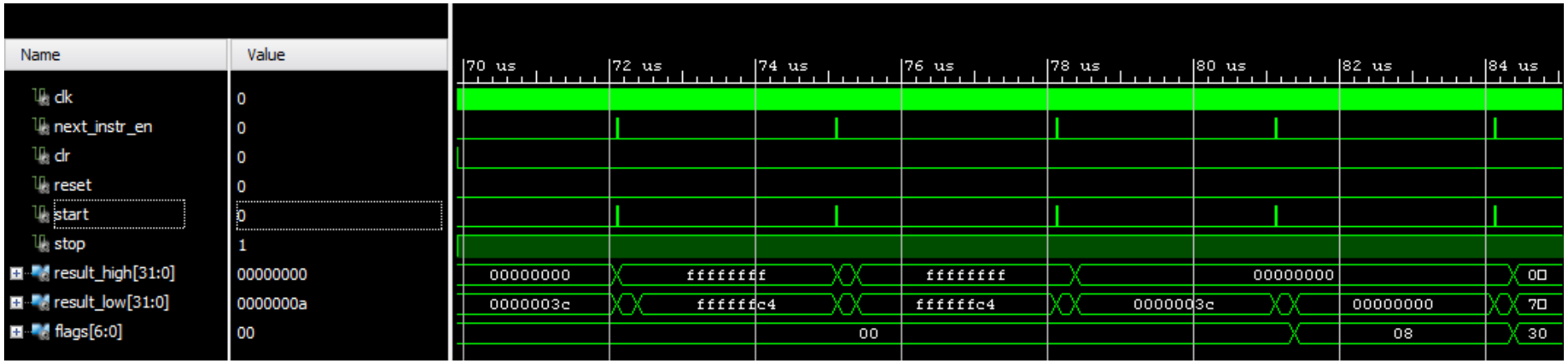
Figură VII 3 Rezultatele simulării



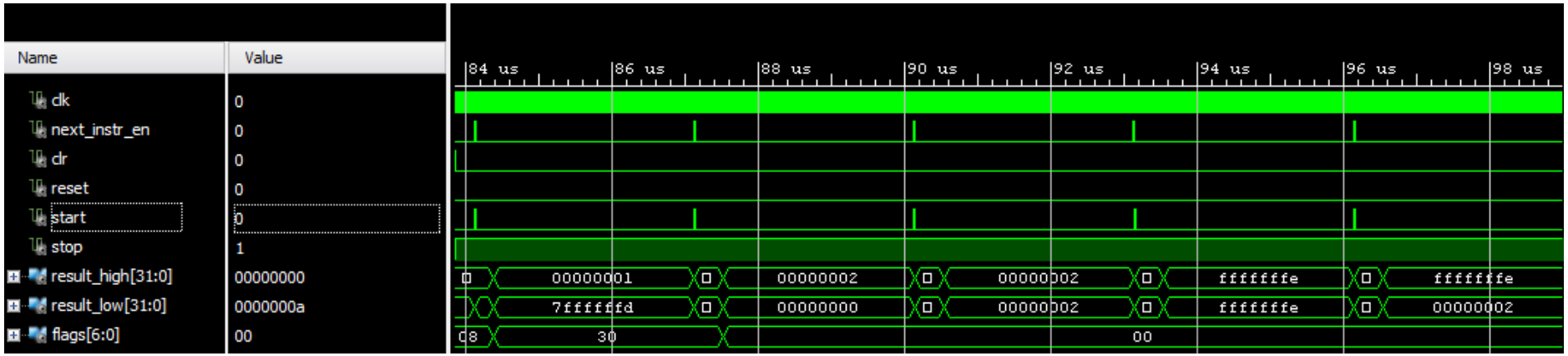
Figură VII 4 Rezultatele simulării



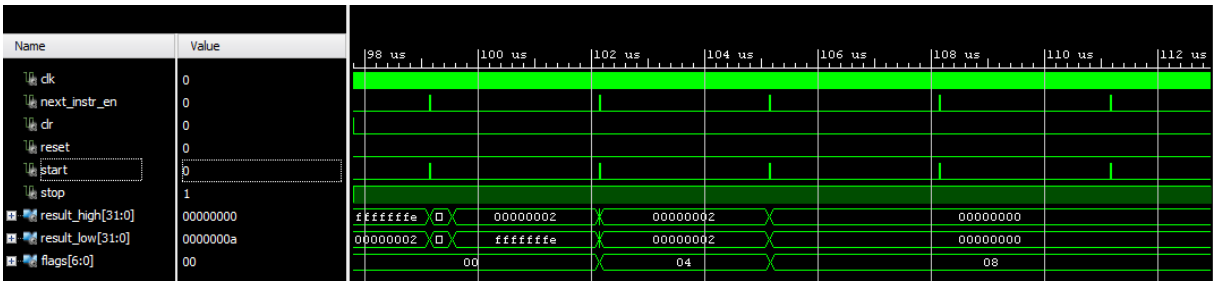
Figură VII 5 Rezultatele simulării



Figură VII 6 Rezultatele simulării



Figură VII 7 Rezultatele simulării



Figură VII 8 Rezultatele simulării

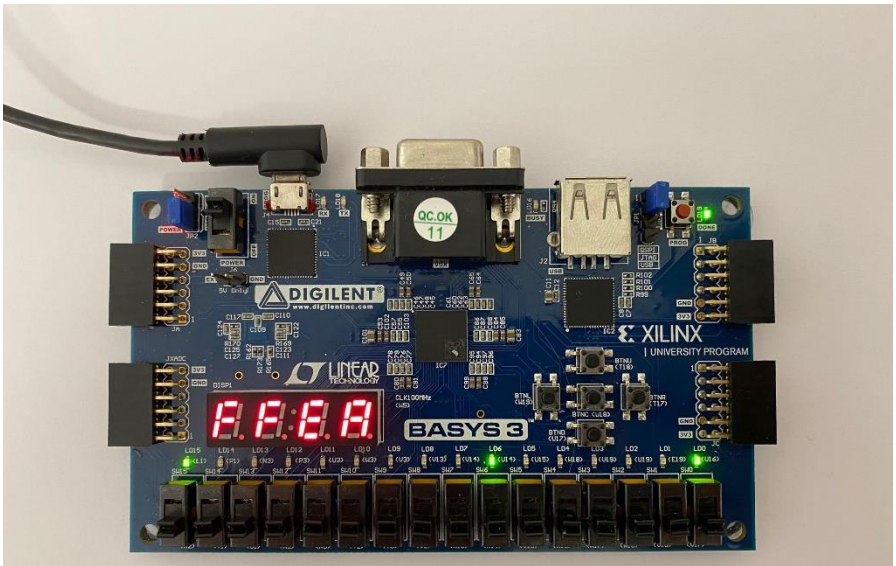
După cum poate fi observat în rezultatele simulării, circuitul implementat răspunde pozitiv tuturor cererilor, furnizând rezultatele așteptate pentru fiecare dintre operațiile implementate.

VIII. Concluzii

Concluzionând, se poate afirma că acest proiect vine să soluționeze una dintre cele mai întâlnite probleme de către orice sistem de calcul, și anume efectuare de operații pe numere întregi, prin proiectarea și implementarea unei componente principale a unității centrale de prelucrare: unitatea aritmetico-logică.

În urma implementării circuitului, s-a ajuns la concluzia că operațiile uzuale asupra numerelor întregi reprezintă o problemă complexă pe care unitatea centrală a oricărui sistem de calcul o întâmpină, și astfel, unitatea aritmetico-logică trebuie să asigure flexibilitate și corectitudine în furnizarea rezultatelor de care celelalte componente au nevoie.

Pentru dezvoltatorul proiectului, implementarea unității aritmetico-logice s-a dovedit utilă, având drept rezultat o mai bună însușire a manipulării numerelor întregi reprezentate în complement față de 2, a conceptelor de bază necesare proiectării și implementării componentelor hardware folosind limbajul VHDL, dar și a programării unor dispozitive de tip FPGA, în cazul proiectului de față fiind folosită o placă Basys 3, bazată pe familia FPGA Artix 7.



Figură VIII 1 Vizualizarea rezultatelor pe placa Basys 3, programată folosind unitatea aritmetico-logică implementată

Resurse bibliografice:

[1] Hennessy, J. L., Patterson, D. A., *Computer Organization and Design – The Hardware/Software Interface*, Ediția a V-a, 2014, [Online], Disponibilă: https://ict.iitk.ac.in/wp-content/uploads/CS422-Computer-Architecture-ComputerOrganizationAndDesign5thEdition2014.pdf?fbclid=IwAR1-CVXkP9pNwnbNNlHyDjShasmfCexVD9XPkv2fep6SiT2Bnp6P7Xbpf_k

[2] Baruch, Z. F., *Arithmetic Logic Unit – Shift-and-Add Multiplication*, [Online], Disponibilă: https://users.utcluj.ro/~baruch/book_ssce/SSCE-Shift-Mult.pdf

[3] Baruch, Z. F., *Arithmetic Logic Unit – Restoring Division*, [Online], Disponibilă: https://users.utcluj.ro/~baruch/book_ssce/SSCE-Basic-Division.pdf

[4] *All About FPGA – ALU Structural Modelling FPGA Implementation*, [Online], Disponibilă: https://allaboutfpga.com/tutorial-3-alu-structural-modelling-fpga-implementation/?fbclid=IwAR3MIWvq0300E0AUT-n-UU32LtPjHAPHhIWyq3rOGwUAkX1UUB2bN9u1yxY

[5] *Arithmetic Logic Unit (ALU)*, [Online], Disponibilă: https://getmyuni.azureedge.net/assets/main/study-material/notes/electronics-communication_engineering_computer-architecture_arithmetic-and-logic-unit_notes.pdf

[6] *What is an arithmetic-logic unit (ALU) and how does it work?*, [Online], Disponibilă: https://whatis.techtarget.com/definition/arithmetic-logic-unit-ALU

[7] *Complement față de 2 – Wikipedia*, [Online], Disponibilă: https://ro.wikipedia.org/wiki/Complement_fa%C8%9B%C4%83_de_doi

[8] Babic, G., *Arithmetic / Logic Unit – ALU Design*, [Online], Disponibilă: https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/CSE675_05_ALUDesign.pdf

[9] *Control Unit, ALU, and Memory*, [Online], Disponibilă: https://www.robots.ox.ac.uk/~dwm/Courses/2CO_2014/2CO-N3.pdf?fbclid=IwAR3z00dhFQEXv0h1IpGOoMbrPfxP8FbHk0toogUHiC8vLJatYH34nrphy4A

[10] *Concurrent Statements - GENERATE*, [Online], Disponibilă: http://web.engr.oregonstate.edu/~traylor/ece474/vhdl_lectures/essential_vhdl_pdfs/essential_

vhdl61-76.pdf?fbclid=IwAR38ISUTMm3SKNDru4aRpqJ96yBS3xLE1AqEsztkcjA1PJiGCH7hgX0PHRE

[11] *Basys 3 Reference Manual*, [Online], Disponibil: https://digilent.com/reference/programmable-logic/basys-3/reference-manual