MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Structura sistemelor de calcul

Proiect : Proiectarea unei unități aritmetico - logice (UAL)

Student : Sescu Diana

Grupa : 30234

Profesor îndrumător : Bozdog Raluca

An universitar : 2025 – 2026

Cuprins :

I. Propunere de proiect

Se cere proiectarea unei unități aritmetico - logice a cărei operanzi să fie întregi pe 32 de biți în complement față de 2 (C2). UAL proiectată trebuie să permită efectuarea următoarelor operații aritmetice : adunare și scădere in complement față de 2, incrementare, decrementare, negare și a operațiilor logice : ȘI – logic pe biți , SAU – logic pe biți, NOT – logic pe biți, rotație logică spre stânga și rotație logică spre dreapta. Unitatea aritmetico - logică necesită folosirea unui acumulator care prelucrează unul dintre operanzii de intrare, rezultatul operațiilor aritmetice sau logice fiind stocat în acumulator. În plus, proiectul va conține și un circuit separat pentru înmulțirea si împărțirea numerelor pe 32 de biți.

Pentru instrucțiunile pe care unitatea aritmetico - logică le va executa se va proiecta o memorie, trecerea de la o instrucțiune la alta efectuându-se după apăsarea unui buton (enable). Pentru decodificarea instrucțiunilor din memorie va fi proiectată si o unitate centrală (UC) responsabilă de obținerea operanzilor, semnalelor de control și gestionarea excepțiilor.

Implementarea UAL se va realiza structural utilizând limbajul de descriere hardware VHDL.

II. Studiu bibliografic

Introducere generală:

Unitatea aritmetico – logică reprezintă unul dintre cele mai importante blocuri funcționale din arhitectura unui sistem de calcul digital. Ea este responsabilă de efectuarea operațiilor aritmetice și logice asupra datelor binare, construind elementul central al unității centrale de prelucrare (CPU).

UAL este un circuit combinațional complex, capabil să realizeze, în funcție de semnalele de control primite, o gamă variată de operații, de la adunare și scădere, până la operații logice elementare pe biți. În general, UAL utilizează un registru acumulator, care reține unul dintre operanzi și în care este stocat rezultatul fiecărei operații.

Reprezentarea datelor – Aritmetică în complement față de 2

Pentru reprezentarea numerelor întregi cu semn se utilizează sistemul complement față de 2, datorită proprietăților sale de a permite efectuarea operațiilor de adunare și scădere cu același circuit hardware.

În această reprezentare numerele pozitive se codifică la fel ca în binar iar numerele negative se obțin prin inversarea tuturor biților valorii absolute și adunarea lui 1 la rezultat.

Astfel, pentru o lungime de n biți, domeniul valorilor posibile este :

$$[-2^{n-1},\ 2^{n-1}-1]$$

iar valoarea zecimală a unui număr binar exprimat în complement față de 2 este dată de relația:

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

Un avantaj major al acestei reprezentări constă în faptul că poate fi implementată ca o simplă adunare:

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

prin urmare, aceeași unitate hardware de adunare poate fi folosită pentru ambele operații aritmetice fundamentale.

Strategia de proiectare:

Proiectul urmărește implementarea structurală a unei unități aritmetico – logice pe 32 de biți, realizată ierarhic, pornind de la un modul de 1 bit, apoi extins pe 8 biți și, în final, la 32 de biți. Această abordare modulară are rolul de a asigura claritate, posibilitatea testării pe nivele intermediare și reutilizarea componentelor elementare.

III. Planificare

- Săptămâna 4 (22 oct. 2025) – Propunere de proiect, studiu bibliografic și planificare
- Săptămâna 6 (5 noi. 2025) – Proiectare și implementare UAL pe 1 bit, UAL pe 8 biți prin interconectarea a 8 UAL pe 1 bit.
- Săptămâna 8 (19 noi. 2025) – Implementarea circuitelor pentru înmulțitor și împărțitor
- Săptămâna 10 (3 dec. 2025) – Sinteza și implementarea UC și memoria pentru UAL
- Săptămâna 12 (17 dec. 2025) – Interconectare componente, testare și rezultate experimentale

IV. Analiză

Etapa de analiză se bazează pe descrierea listei de funcționalități menționate în specificația proiectului și are ca 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 pe 32 de biți.

Operații aritmetice de bază

1. Adunarea

Operația de adunare reprezintă fundamentul operațiilor aritmetice din cadrul UAL, întrucât majoritatea celorlalte operații pot fi reduse la adunări : scăderea devine o adunare cu opusul scăzătorului, incrementarea este o adunare în care un operand este 1, decrementarea o scădere în care un operand este 1, înmulțirea o secvență de adunări repetate, iar împărțirea o succesiune de scăderi.

Pentru a putea realiza operația de adunare pe 32 de biți trebuie să pornim de la adunarea pe 1 bit. Ecuațiile fundamentale sunt

$$S = A\ xor\ B\ \ xor\ C_{in}$$
$$C_{out} = AB\ or\ (A\ and\ B)C_{in}$$

Extensia la 32 de biți se realizează prin algoritmul Ripple-Carry Adder care propagă transportul generat la un rang inferior drept transport de intrare pentru rangul imediat următor. Ecuațiile pentru biții de pe poziția i devin:

$$S_i = A_i\ xor\ B_i\ \ xor\ C_{ini}$$
$$C_{outi} = A_iB_i\ or\ (A_i\ and\ B_i)C_{ini}$$

Depășirea (overflow) are loc când rezultatul are semn opus de operanzii si se poate produce doar daca cei doi operanzi, A și B, au același semn. Detecția depășirii se poate face :

$$\text{Overflow } = A_{31}\,B_{31}\bar{S}_{31} + \bar{A}_{31}\bar{B}_{31}S_{31}$$

Alternativ, depășirea poate fi detectată prin XOR între transporturile de intrare și ieșire de la bitul cel mai semnificativ. Practic, depășirea apare atunci când transportul de intrare și cel de ieșire de la rangul cel mai semnificativ diferă:

$$\text{Overflow } = C_{out31}\ xor\ C_{in31} = \ C_{out31}\ xor\ C_{out30}$$

2. Scăderea

Operația de scădere exploatează reprezentarea în complement față de 2,

Opusul unui număr se obține prin:

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

Astfel, scăderea devine :

$$A - B = A + (\bar{B} + 1)$$

Extensia operații de scădere de la 1 bit la 32 de biți se realizează ca la adunare deoarece scăderea reprezintă, în fapt, o adunare.

Depășirea domeniului de reprezentare diferă față de adunare, la scădere apărând atunci când operanzii au semn diferit iar semnul rezultatului este identic cu semnul scăzătorului. Astfel, ecuația de detecție a depășirii este :

$$\text{Overflow} = A_{31}\,\bar{B}_{31}\bar{S}_{31} + \bar{A}_{31}B_{31}S_{31}$$

3. Incrementarea

   Operația de incrementare reprezintă adunarea cu valoarea 1 :

   $$x = x + 1$$

   Se realizează prin setarea celui de-al doilea operand la valoarea 1,

folosind același circuit de adunare. Cazurile de depășire si algoritmul sunt identice cu cele prezentate la adunare

4. Decrementarea

   Decrementarea reprezintă scăderea cu valoarea 1 :

   $$x = x - 1$$

   Operația de decrementare este de fapt o scădere în care scăzătorul este 1. Cazurile de depășire și algoritmul sunt identice cu cele prezentate în secțiunea dedicată scăderii.

5. Negarea

   Operația de negare are ca scop obține opusul unui număr. Se bazează pe proprietățile reprezentării în complement față de 2. Negarea se obține prin complementarea tuturor biților din reprezentarea binară a numărului și incrementarea valorii obținute. Procesul folosit este :

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

Operații logice

1. ȘI – logic

   Operația ȘI-logic (AND) are ca rezultat 1 doar când amândoi operanzii sunt 1. Tabelul de adevăr :

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

Pentru a aplica ȘI-logic pe 32 de biți trebuie cascadate 32 de porți AND, câte o poarta pentru fiecare pereche de biți ai operanzilor. Astfel, pentru 32 de biți :

$$Y_i = A_i \; and \; B_i$$

2. SAU-logic

Operația SAU-logic (OR) are ca rezultat 1 atunci când cel puțin un operand are valoarea 1.

Tabelul de adevăr :

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

Pentru a aplica SAU-logic pe 32 de biți trebuie cascadate 32 de porți OR, câte o poarta pentru fiecare pereche de biți ai operanzilor. Astfel, pentru 32 de biți :

$$Y_i = A_i \; or \; B_i$$

3. NU-logic

Operația NU-logic (NOT) este o operație cu un singur operand, rezultatul fiind operandul negat.

Tabelul de adevăr :

A	not A
0	1
1	0

Negarea unui număr pe 32 de biți, bit cu bit, reprezintă complementul față de 1 al numărului respectiv.

4. Rotire logică spre stânga :

Operația realizează o rotire circulară spre stânga a tuturor biților operandului cu o poziție. Pe poziția 0, adică în locul bitului cel mai puțin semnificativ va ajunge bitul cel mai semnificativ, adică bitul 31.

Rezultatul operației este :

$$y = x_{30}x_{29} \ldots . x_0x_{31}$$

5. Rotire logică spre dreapta :

Operația realizează o rotire circulară spre dreapta a tuturor biților operandului cu o poziție. Reprezintă operația opusă rotirii logice spre stânga, bitul cel mai puțin semnificativ devenind bitul cel mai semnificativ.

Rezultatul operației este :

$$y = x_0x_{31} \ldots . x_2x_1$$

Operații aritmetice suplimentare

1. Înmulțirea :

Algoritmul folosit pentru înmulțire este Shift-and-Add Multiplication bazat pe înmulțirea binară și adunări.

Principiu :

a. Pentru fiecare bit al înmulțitorului B (de la stânga la dreapta):
   - Dacă bitul = 1: adună A shiftat cu i poziții la rezultatul parțial
   - Dacă bitul = 0 : nu adună nimic.
b. Acumulează toate produsele parțiale într-o matrice de 32×32
c. După calcularea produselor parțiale se shiftează fiecare linie i cu i poziții la stânga.
d. La final, se adună pe coloane cu un sumator în cascadă.

Pentru gestionarea semnului rezultatului :

a. Se calculează folosind poarta logică XOR semnul pe care trebuie să îl ia rezultatul pe baza semnelor operanzilor :

$$sign = A_{31}\ xor\ B_{31}$$

b. Se calculează valoarea în modul a operanzilor doar dacă au valoare negativă
c. Produsul se calculează folosind valorile absolute
d. Dacă rezultatul de la punctul a. indică semnul negativ pentru produs, negăm rezultatul obținut la punctul c.

Rezultatul final pe 64 de biți permite reprezentarea întregii game de valori posibile.

2. Împărțirea :

Operația de împărțire este de departe cea mai costisitoare operație efectuată la nivelul unității aritmetico-logice. Algoritmul ales pentru acest proiect este Restore Division (împărțire cu refacerea restului), care reprezintă o metodă secvențială iterativă pentru împărțirea numerelor binare întregi.

Principiu :

a. Inițializare :
   - Rest (R) = 0 extins la 33 biți
   - Cât (Q) = deîmpărțitul A
   - Count = 32 de iterații
b. Pentru fiecare iterație :
   - Shift left combinat : R = (R << 1) & MSB(Q), Q = Q << 1
   - Scădere : R = R – împărțitorul B
   - Dacă R < 0 (rezultat negativ) :
     - Restaurare : R = R + B
     - Q(0) = 0
   - Altfel :
     - Q(0) = 1
c. După 32 de iterații:
   - Cât = Q
   - Rest = R

Gestionarea semnului :

- Q are ca semn rezultatul aplicării porții logice XOR între semnul lui A si semnul lui B:

$$sign\_Q = A_{31}\ xor\ B_{31}$$

- R primește semnul lui A:

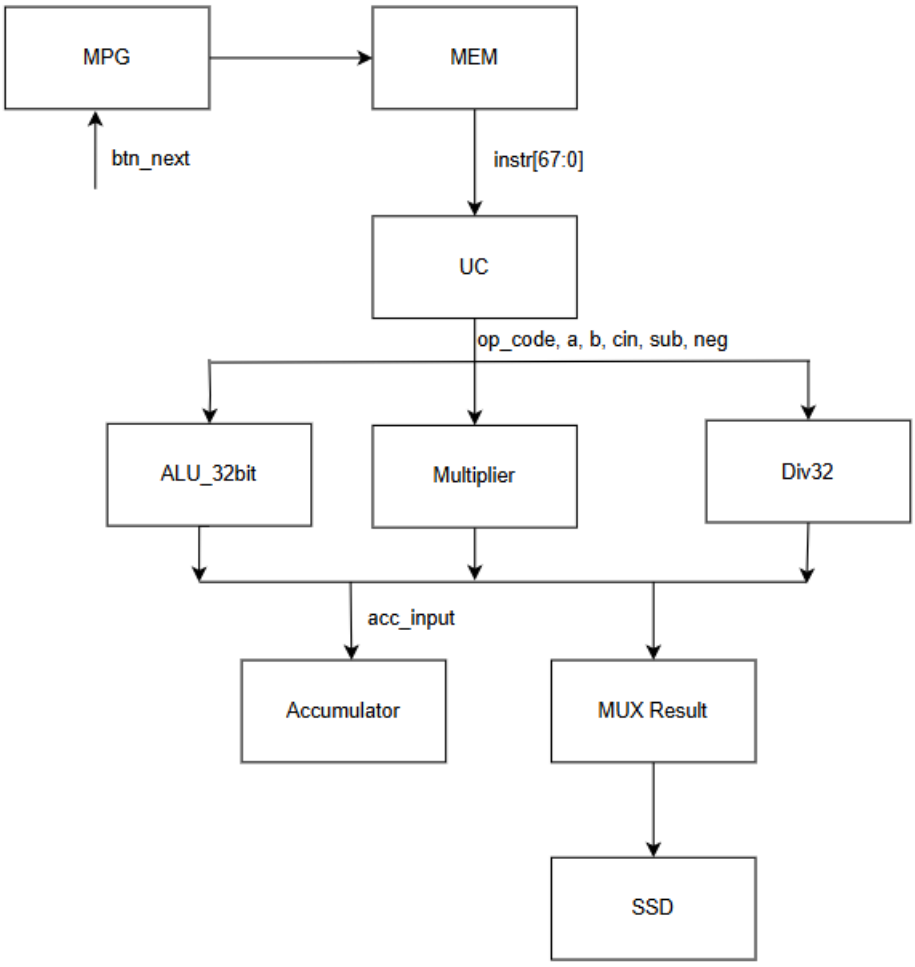
$$sign\_R = A_{31}$$

Deoarece latența algoritmului este de 33 – 34 de cicluri de ceas, detecția împărțirii la 0 se face înainte de pornirea algoritmului, verificând daca B este egal cu 0.

V. Design

Luând în considerare lista funcționalităților ce trebuie satisfăcute de către unitatea ce urmează a fi proiectată, se pot identifica macro-componentele circuitului.

Schema bloc generală a sistemului:



Comunicarea între componentele prezentate în cadrul circuitului se realizează prin intermediul magistralelor de date.

Componentele principale sunt :

a. Memoria de instrucțiuni (MEM.vhd)

Este o memorie de tip ROM, care conține instrucțiuni harcodate, destinate testării unității aritmetice și logice. Conține 16 instrucțiuni pre-programate pentru a testa fiecare operație și excepțiile posibile.

Formatul instrucțiunii (68 biți)

Op_code[67: 64] 4 biți	A[63:32] 32 biți	B[31:0] 32 biți

b. Unitatea de control (UC.vhd)

Unitatea de control primește ca input instrucțiunea extrasă din memorie și o decodifica în op_code, a, b. Pe baza codului de operație generează semnalele de control folosite de circuitul unității aritmetico-logice.

Op_code	Operație	cin	sub	neg	b_instr
0000	ADD	0	0	0	0
0001	SUB	1	1	0	0
0010	AND	0	0	0	0
0011	OR	0	0	0	0
0100	NOT	0	0	0	0
0101	Negare	0	0	1	1
0110	INC	0	0	0	1
0111	DEC	1	1	0	1
1000	ROL	0	0	0	0
1001	ROR	0	0	0	0

Când b_instr = ’1’, operandul B este înlocuit cu valoarea ”0X00000001”.

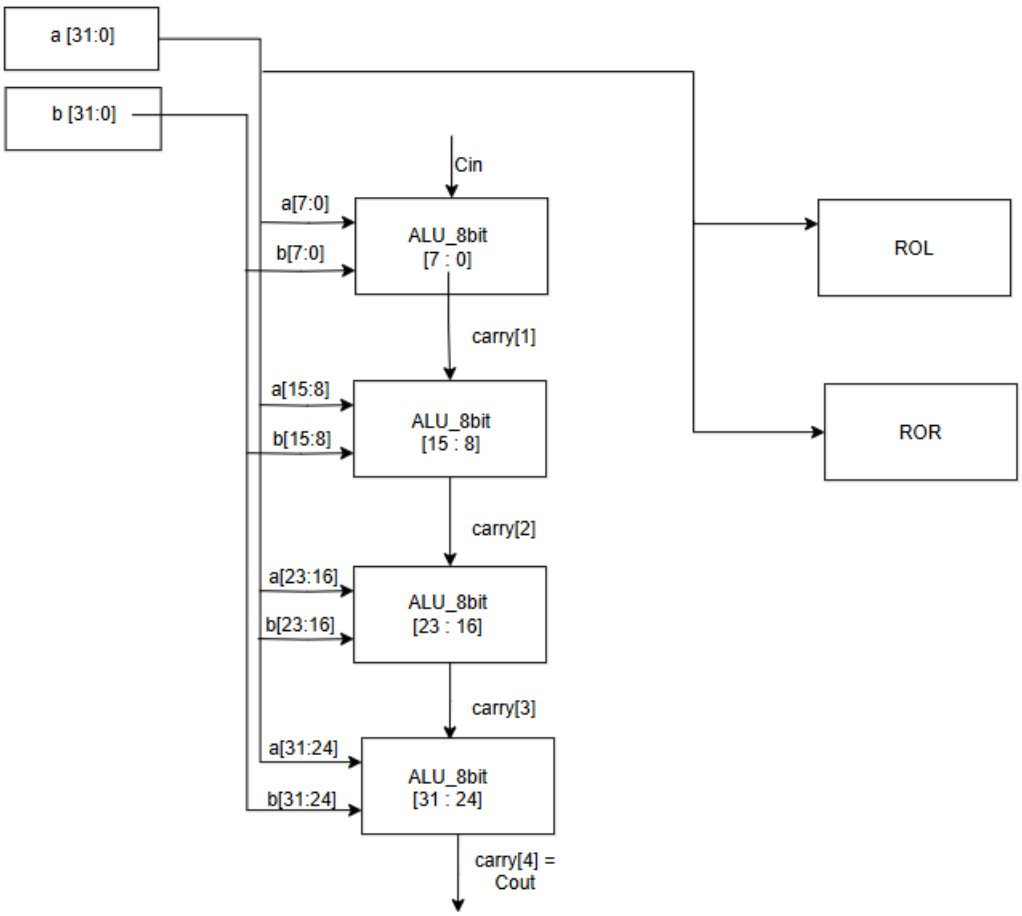
c. Registrul Acumulator (accumulator.vhd)

Este un registru pe 32 de biți cu încărcare controlată, care stochează rezultatele operanților.

d. Unitatea aritmetică și logică (ALU_32bit.vhd)

Unitatea aritmetico-logică este structurat ierarhic: 4 instanțe de Alu_8bit, fiecare conținând 8 instanțe Alu_1bit. În plus, sunt prezente și instanțele pentru ROL și ROR pe 32 de biți.

Schema bloc Alu 32:



Rezultatul este obținut folosind un MUX, care alege în funcție de operația efectuată de unde provine rezultatul: din alu_8bit, din entitatea ROR sau entitatea ROL.

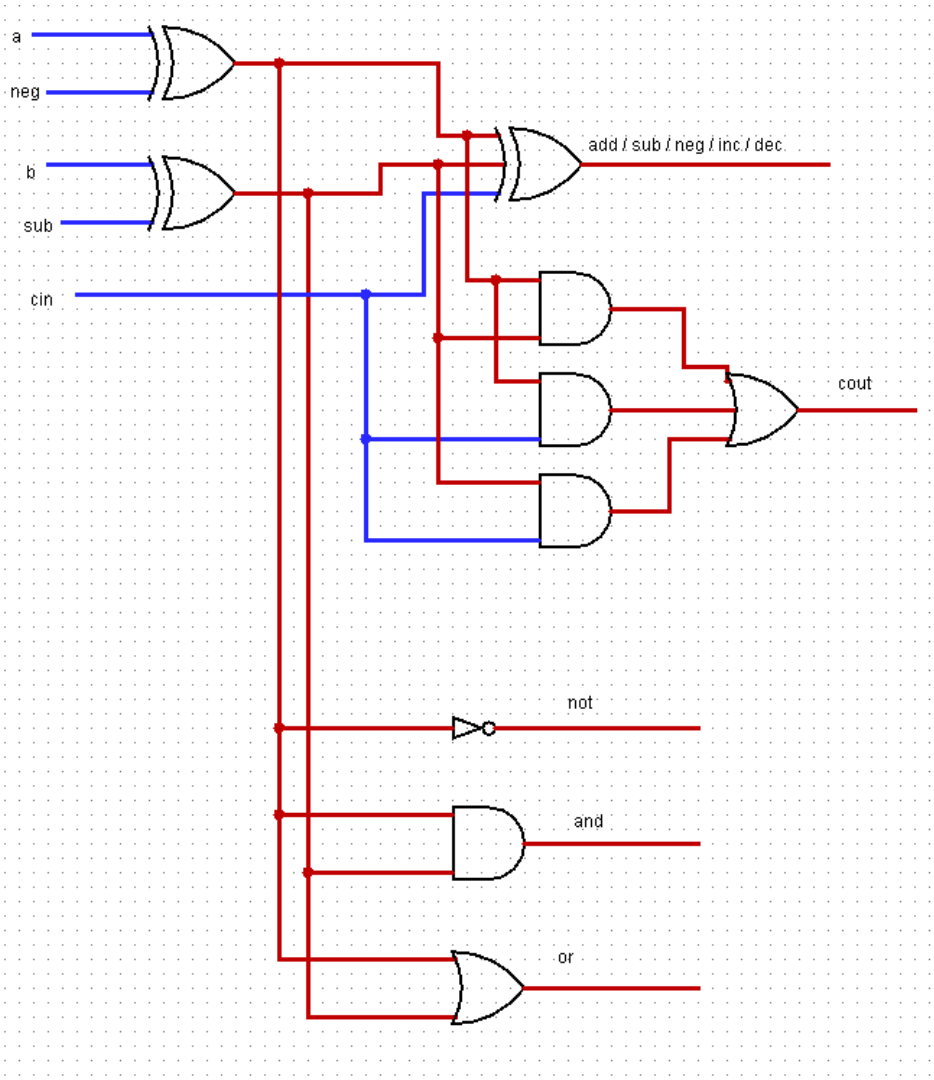
e. Unitatea aritmetico-logică pe 1 bit (ALU_1bit):

Folosind circuitul pentru adunarea a două numere de 1 bit se realizează : adunare, scădere, incrementare, decrementare și negare. Pentru a se realiza corect operațiile folosim 2 semnale de control : sub , care atunci când are valoarea 1 complementează toți biții lui B, și neg care complementează biții lui A la operația de negare.

Operațiile logice : AND, OR, NOT se realizează folosind porțile logice corespunzătoare operației.

Pe baza codului operației se alege de unde provine rezultatul: suma sau porțile logice.

Schema circuitului :



f. Rotirea logică la stânga (rotate_left_32.vhd) :

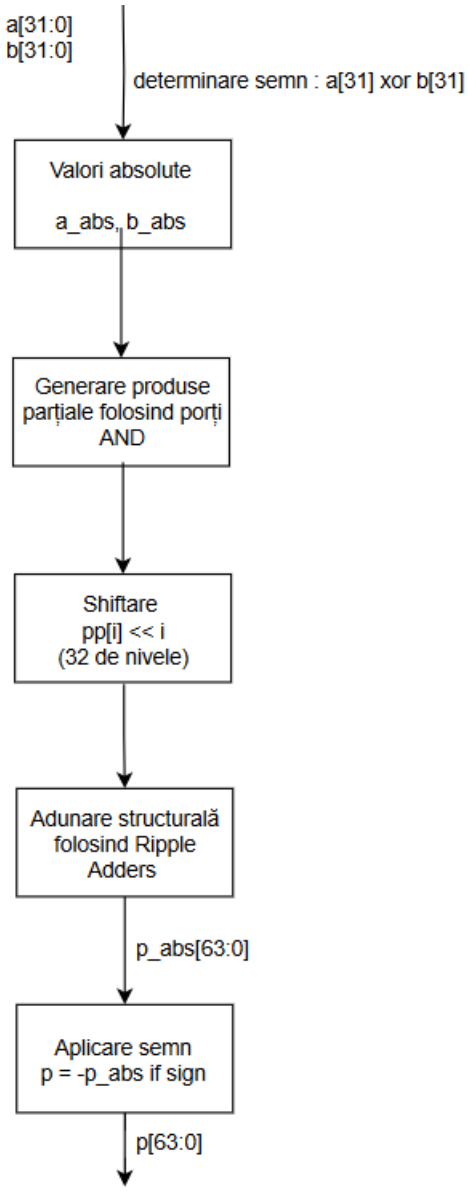
Pentru a efectua rotirea logică la stânga am ales o soluție iterativă, folosind o buclă for, care mută fiecare bit pe poziția imediat următoare în stânga. La finalul buclei, bitul cel mai semnificativ este pus pe poziția 0, a bitului cel mai puțin semnificativ.

g. Rotirea logică la dreapta (rotate_right_32.vhd):

Pentru a efectua rotirea logică la stânga am ales o soluție iterativă, folosind o buclă for, care mută fiecare bit pe poziția imediat următoare în stânga. La finalul buclei, bitul cel mai semnificativ este pus pe poziția 0, a bitului cel mai puțin semnificativ.

h. Înmulțirea (multiplier.vhd) :

Realizată folosind algoritmul Shift-and-Add Multiplication descris în partea de analiză.



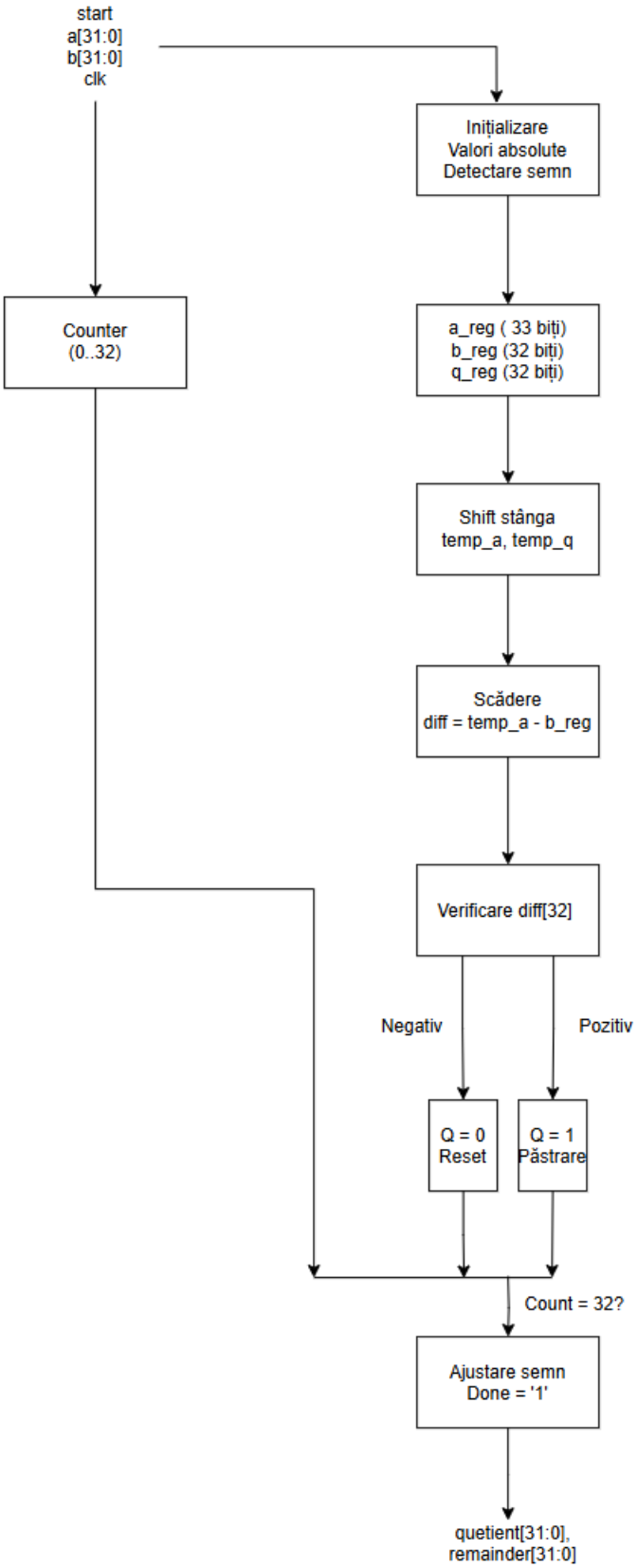
Înmulțitorul produce un rezultat pe 64 de biți. Switch-ul 3 de pe plăcuță ne permite să selectăm daca vrem sa vedem primii 32 de biți sau următorii 32, oferind astfel posibilitatea de verificare în detaliu a rezultatului.

Pentru : - sw(3) = ’0’ : biții inferiori [31:0]

- sw(3) = ’1’ : biții superiori [63:32]

i. Împărțirea (div32.vhd) :

Se realizează folosind algoritmul Restore Division descris în partea de analiză.


start
a[31:0]
b[31:0]
clk
Inițializare
Valori absolute
Detectare semn
Counter
(0..32)
a_reg (33 biți)
b_reg (32 biți)
q_reg (32 biți)
Shift stânga
temp_a, temp_q
Scădere
diff = temp_a - b_reg
Verificare diff[32]
Negativ
Pozitiv
Q = 0
Reset
Q = 1
Păstrare
Count = 32?
Ajustare semn
Done = '1'
quetient[31:0],
remainder[31:0]

Împărțitorul produce un cât și un rest, ambele pe 32 de biți.. Switch-ul 3 de pe plăcuță ne permite să selectăm daca vrem sa vedem câtul sau restul , oferind astfel posibilitatea de verificare în detaliu a rezultatului.

Pentru : - sw(3) = ’0’ : cât

- sw(3) = ’1’ : rest

j. Modului principal(top.vhd):

Modului principal reprezintă componentă de nivel superior care integrează toate sub modulele și implementează logica de control prin intermediul unui automat cu stări finite. Acesta coordonează execuția secvențială a instrucțiunilor, gestionează fluxul de date între componente și asigură interfața cu utilizatorului prin intermediul butoanelor, switch-urilor și a afișoarelor.

Mașina de stări finite este alcătuită din 4 stări responsabile de buna funcționare a circuitului. Stările sunt:

- IDLE : Stare de așteptare, așteaptă apăsarea butonului de schimbare a instrucțiunii
- FETCH : Încarcă instrucțiunea din memorie și resetează flag-urile
- EXEC : Execută operația și decide starea următoare
- LOAD_ACC : Încarcă valorile în acumulator, setând acc_load pe ’1’.
- SHOW_RES : Face ca datele să fie încărcate in acumulator după citirea rezultatului pentru a evita confuzia
- WAIT_DIV : Așteaptă finalizarea împărțirii(doar pentru instrucțiunea DIV)

VI. Implementare

Implementarea unității aritmetice și logice urmărește o abordare modulară, în care fiecare componentă își îndeplinește un rol bine definit în cadrul sistemului. Arhitectura adoptată combina descrieri structurale pentru componente complexe cu descrieri comportamentale pentru modulele de control și interfață, asigurând astfel un echilibru optim între claritatea codului și eficiența implementării pe dispozitivul FPGA.

Codul VHDL ce descrie componentele principale ale unității aritmetice și logice este prezentat în continuare, fiecare entitate fiind detaliată din punct de vedere al structurii porturilor și al funcționalității implementate.

Entitatea principală (top)

Reprezintă modulul de nivel superior al sistemului, integrând toate componentele unității aritmetice și logice și asigurând interfața cu utilizatorul prin intermediul perifericelor disponibile pe plăcuța Nexys A7.

```
entity top is
    Port ( clk : in STD_LOGIC;
           btn_next : in STD_LOGIC;
           btn_load : in STD_LOGIC;
           btn_rst : in STD_LOGIC;
           sw : in STD_LOGIC_VECTOR(3 downto 0);
           ledOver : out STD_LOGIC;
           ledIlligalOp : out STD_LOGIC;
           ledDivByZero : out STD_LOGIC;
           an : out STD_LOGIC_VECTOR (7 downto 0);
           cat : out STD_LOGIC_VECTOR (6 downto 0));
end top;
```

Semnificația porturilor de intrare/ieșire :

- clk – intrare pentru semnalul de ceas
- btn_next – intrare ce validează trecerea la instrucțiunea următoare
- btn_load – intrare care validează încărcarea rezultatului în acumulator;
- btn_rst – intrare care reinițializează întregul sistem
- sw - intrare ce permite schimbări în sistem: sw(0) și sw(1) determină ce se afișează pe SSD : 00 – primul operand, 01 – al doilea operand, 10 – codul operației, 11- rezultatul ; sw(2) determină daca pentru primul operand folosim valoarea din memorie (când se află pe 0 sau LOW) sau valoarea din acumulator (când se află pe 1 sau HIGH); sw(3) determină daca vedem low (0) sau high (1) la operația de înmulțire sau câtul (0) sau restul (1) la operația de împărțire
- ledOver – ieșire; LED dedicat care se aprinde pentru a semnaliza Overflow;
- ledIlligalOp - ieșire; LED dedicat care se aprinde pentru a semnaliză o operație ilegală, nedefinită
- ledDivByZero - ieșire; LED dedicat care se aprinde pentru a semnaliză încercarea efectuării unei împărțiri la 0
- an – ieșire; vector pe 8 biți ce controlează anozii afișorului pe 7 segmente
- cat – ieșire; vector pe 7 biți ce controlează catozii afișorului pe 7 segmente

Unitatea de control (UC)

Este componenta responsabilă de decodificarea instrucțiunilor primite din memoria ROM și de generare a semnalelor de control necesare pentru configurarea unității aritmetice și logice în vederea executării operației specificate.

```
entity UC is
    Port ( instr : in STD_LOGIC_VECTOR (67 downto 0);
           op_code : out STD_LOGIC_VECTOR (3 downto 0);
           a : out STD_LOGIC_VECTOR (31 downto 0);
           b : out STD_LOGIC_VECTOR (31 downto 0);
           cin : out STD_LOGIC;
           sub : out STD_LOGIC;
           neg : out STD_LOGIC);
end UC;
```

Semnificația porturilor de intrare/ieșire :

- instr – intrare reprezentând instrucțiunea curentă ce urmează să fie decodificată pe 68 de biți;
- op_code – ieșire de 4 biți reprezentând codul operației ce urmează să fie executată;
- a – ieșire de 32 de biți ce reprezintă primul operand
- b – ieșire de 32 de biți ce reprezintă al doilea operand
- cin – ieșire reprezentând bitul de transport de intrare
- sub – ieșire de control pentru operațiile ce necesită efectuarea unei scăderi
- neg – ieșire de control pentru operațiile ce necesită negarea primului operand

Unitatea aritmetică și logică pe 32 de biți (Alu_32bit)

Reprezintă nucleul sistemului, fiind responsabilă de executarea operațiilor aritmetice(adunare, scădere, incrementare, decrementare) și logice (ȘI, SAU, NU, rotiri) asupra operanzilor pe 32 de biți.

```
entity Alu_32bit is
    Port ( a : in STD_LOGIC_VECTOR (31 downto 0);
           b : in STD_LOGIC_VECTOR (31 downto 0);
           cin : in STD_LOGIC;
           sub : in STD_LOGIC;
           neg : in STD_LOGIC;
           op : in STD_LOGIC_VECTOR (3 downto 0);
           result : out STD_LOGIC_VECTOR (31 downto 0);
           overFlow : out STD_LOGIC);
end Alu_32bit;
```

Semnificația porturilor de intrare/ieșire :

- a – intrare pentru primul operand, reprezentat pe 32 de biți în complement față de 2
- b – intrare pentru al doilea operand, reprezentat pe 32 de biți în complement față de 2
- cin – intrare de transport pentru operațiile aritmetice
- sub – intrare de control ce indică efectuarea unei scăderi în locul adunării
- neg – intrare de control pentru negarea primului operand
- op – intrare; vector pe 4 biți reprezentând codul operației ce trebuie modificată
- result – ieșire de 32 de biți ce reprezintă rezultatul operației efectuate, reprezentat în complement față de 2
- overFlow – ieșire de semnalizare care se activează când o operație aritmetică produce un rezultat care depășește domeniul de reprezentare

Înmulțitorul pe 32 de biți (multiplier)

Circuitul de înmulțire implementează operația de înmulțire a două numere întregi cu semn, reprezentate pe 32 de biți fiecare, furnizând un produs reprezentat pe 64 de biți pentru a acomoda întreaga gamă de valori posibile.

```
entity multiplier is
    Port ( a : in signed (31 downto 0);
           b : in signed (31 downto 0);
           p : out signed (63 downto 0));
end multiplier;
```

Semnificația porturilor de intrare/ieșire :

- a – intrare pentru primul operand al înmulțirii, de tip signed pe 32 de biți
- b – intrare pentru al doilea operand al înmulțirii, de tip signed pe 32 de biți
- p – ieșire reprezentând rezultatul înmulțirii, de tip signed pe 64 de biți, format necesar pentru a preveni pierderea de informație în cazul valorilor maxime

Împărțitorul pe 32 de biți (div32)

Circuitul de împărțire binară implementează operația de împărțire cu rest a două numere pe 32 de biți, furnizând atât câtul cât și restul împărțirii. Datorită complexității algoritmului, această componentă necesită cele mai multe cicluri de ceas pentru finalizarea operației.

```
entity div32 is
    Port ( clk : in STD_LOGIC;
           start : in STD_LOGIC;
           a : in STD_LOGIC_VECTOR (31 downto 0);
           b : in STD_LOGIC_VECTOR (31 downto 0);
           quotient : out STD_LOGIC_VECTOR (31 downto 0);
           remainder : out STD_LOGIC_VECTOR (31 downto 0);
           done : out STD_LOGIC );
end div32;
```

Semnificația porturilor de intrare/ieșire :

- clk – intrare pentru semnalul de ceas, necesar orchestrării pașilor succesivi ai algoritmului de împărțire
- start – intrare de control care inițializează operația de împărțire; la activarea acestui semnal circuitul începe procesul de calcul
- a – intrare de 32 de biți reprezentând deîmpărțitul
- b – intrare de 32 de biți reprezentând împărțitorul
- quotient – ieșire de 32 de biți ce conține câtul împărțirii
- remainder – ieșire pe 32 de biți conținând restul împărțirii
- done – ieșire; semnal de finalizare care se activează pe nivel logic 1 când operația de împărțire s-a încheiat și rezultatele sunt disponibile pe ieșiri

Registrul Acumulator (accumulator)

Este o componentă de stocare pe 32 de biți care păstrează rezultatul ultimei operații efectuate, permițând utilizarea acestuia ca operand în operații ulterioare, similar funcționării calculatoarelor clasice cu arhitectură bazată pe acumulator.

```
entity accumulator is
    Port ( clk : in STD_LOGIC;
           data : in STD_LOGIC_VECTOR (31 downto 0);
           a_out : out STD_LOGIC_VECTOR (31 downto 0);
           load : in STD_LOGIC);
end accumulator;
```

Semnificația porturilor de intrare/ieșire :

- clk – intrare pentru semnalul de ceas, necesar sincronizării operației de încărcare a datelor în registru
- load – intrare de control care validează încărcarea unei noi valori în acumulator
- data – intrare de 32 de biți reprezentând valoarea ce urmează a fi încărcată în registruș acumulator
- a_out – ieșire de 32 de biți care oferă acces continuu la valoarea curentă stocată în acumulator, putând fi folosită ca operand pentru operațiile viitoare

Memoria de instrucțiuni (MEM)

Este o memorie de tip ROM care stochează un set predefinit de instrucțiuni destinate testării și validării funcționalităților implementate de unitatea aritmetică și logică. Permite execuția secvențială a operațiilor de test fără necesitatea unei intervenții externe continue.

```
entity MEM is
    Port ( clk : in STD_LOGIC;
           addr : in STD_LOGIC_VECTOR (3 downto 0);
           instr : out STD_LOGIC_VECTOR (67 downto 0));
end MEM;
```

Semnificația porturilor de intrare/ieșire :

- clk – intrare pentru semnalul de ceas, necesar sincronizării operației de citire a instrucțiunilor din memorie
- addr – intrare de adresă reprezentată pe 4 biți, indicând indexul instrucțiunii curente din memoria ROM
- instr – ieșire de date reprezentând instrucțiunea curentă citită din memorie, codificată pe 68 de biți conform formatului definit în partea de Design

Conținutul memoriei de instrucțiuni :

```
signal rom : rom_type := (
    0  => "0000" & X"00000005" & X"00000003", -- add ; 00000005 + 00000003 = 00000008
    1  => "0001" & X"00000009" & X"00000002", -- sub ; 00000009 - 00000002 = 00000007
    2  => "0010" & X"FFFFFFFF" & X"00000001", -- and ; FFFFFFFF && 00000001 = 00000001
    3  => "0011" & X"0000000F" & X"00000001", -- or ; 0000000F || 00000001 = 0000000f
    4  => "0100" & X"0000000F" & X"00000000", -- not ; ~0000000F = FFFFFFF0
    5  => "0101" & X"FFFFFFF0" & X"00000000", -- neg ; FFFFFFF0 => 00000010
    6  => "0110" & X"0000000A" & X"00000000", -- inc ; 0000000A++ = 0000000B
    7  => "0111" & X"0000000A" & X"00000000", -- dec ; 0000000A-- = 00000009
    8  => "1000" & X"00000001" & X"00000000", -- rol ; rol 00000001 = 00000002
    9  => "1001" & X"00000001" & X"00000000", -- ror ; ror 00000001 = 80000000
    10 => "1010" & X"00000006" & X"00000007", -- mul ; 00000006 * 00000007 = 0000002A
    11 => "1011" & X"00000021" & X"FFFFFFFE", -- div ; 00000021 / FFFFFFFE = FFFFFFF0, 00000001
    12 => "1010" & X"00000006" & X"FFFFFFFE", -- mul cu operand negativ ; 00000006 * FFFFFFFE = FFFFFFFF FFFFFFF4
    13 => "0000" & X"7FFFFFFF" & X"00000001", -- overflow
    14 => "1011" & X"23000000" & X"00000000", --divByZero
    15 => "1111" & X"12345678" & X"89456123"  --illigalOp
    );
```

Generator de monoimpuls (MPG)

Generatorul de monoimpuls sincron este o componentă hardware auxiliară esențială pentru interfața utilizator, având rolul de a elimina fenomenul de ”bouncing” caracteristic butoanelor fizice și de a asigura o singură activare a semnalului de enable la fiecare apăsare a butonului.

```
entity MPG is
    Port ( enable : out STD_LOGIC;
           btn : in STD_LOGIC;
           clk : in STD_LOGIC);
end MPG;
```

Afișorul pe 7 segmente (SSD)

Afișorul pe 7 segmente este componenta de interfață responsabilă de vizualizarea datelor în format hexazecimal pe cele 8 cifre disponibile pe plăcuța Nexys A7, folosind tehnica de multiplexare temporală pentru controlul individual al fiecărei cifre.

```
entity SSD is
    Port ( clk : in STD_LOGIC;
           digits : in STD_LOGIC_VECTOR(31 downto 0);
           an : out STD_LOGIC_VECTOR(7 downto 0);
           cat : out STD_LOGIC_VECTOR(6 downto 0));
end SSD;
```

VII. Simulare și testare

Validarea corectitudinii implementării unității aritmetice și logice se realizează prin execuția unui set de cazuri de testare predefinite, stocate în memoria ROM de instrucțiuni.

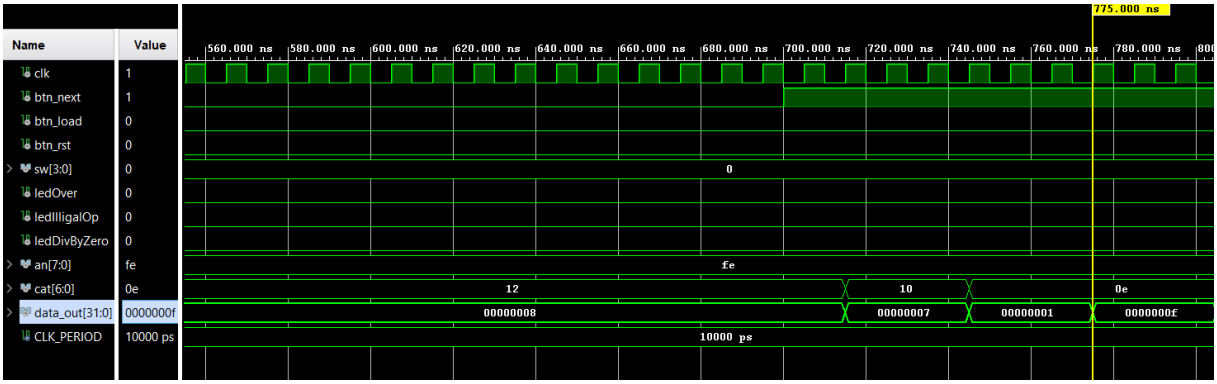
Aceste cazuri au fost concepute pentru a acoperi o gamă diversă de scenarii de funcționare, incluzând operații cu operanzi pozitivi, negativi, valori limită și situații excepționale.

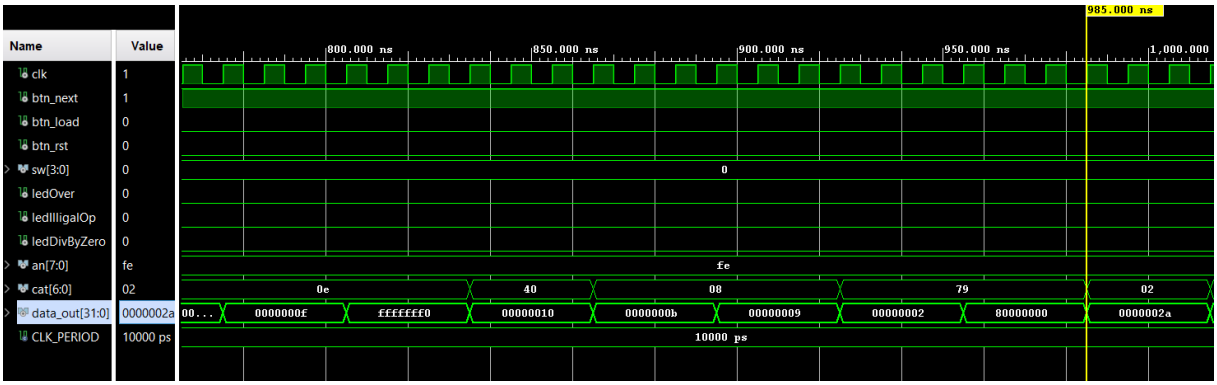
Primul operand		Al doilea operand		Operație	Rezultat așteptat		Rezultat obținut	
Baza 10	Baza 16	Baza 10	Baza 16		Baza 10	Baza 16	Baza 10	Baza 16
5	5	3	3	+	8	8	8	8
9	9	2	2	-	7	7	7	7
-1	FFFFFFFF	1	1	and	1	1	1	1
15	F	1	1	or	15	F	15	F
15	F	0	0	not	-16	FFFFFFF0	-16	FFFFFFF0
-16	FFFFFFF0	0	0	neg	16	10	16	10
10	A	0	0	inc	11	B	11	B
10	A	0	0	dec	9	9	9	9
1	1	0	0	rol	2	2	2	2
1	1	0	0	ror	2147483648	80000000	2147483648	80000000
6	6	7	7	*	42	2A	42	2A
33	21	-2	FFFFFFFE	/	Cât = -16 Rest = 1	FFFFFFF0 1	Cât = -16 Rest = 1	FFFFFFF0 1
6	6	-2	FFFFFFFE	*	-12	FFFFFFF4	-12	FFFFFFF4
2147483647	7FFFFFFF	1	1	+	Overflow	-	-	-

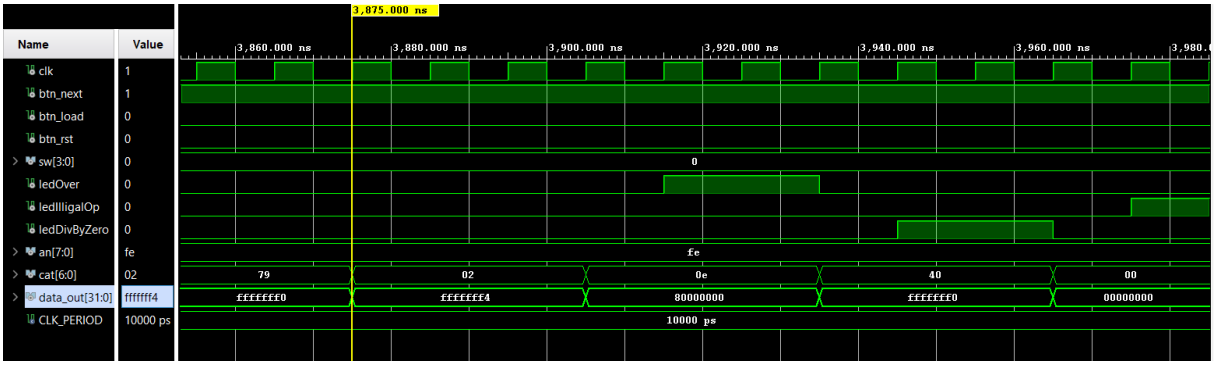
587202560	23000000	0	0	/	Division By zero	-	-	-
305419896	12345678	2557891634	98765432	Illigal Op	Illigal OP	-	-	-

Fiecare instrucțiune din memoria ROM corespunde unui caz de testare specific, alegerea operanzilor și a operațiilor fiind efectuată strategic pentru a verifica comportamentul corect al unității în diverse condiții de utilizare. Tabelul precedent prezintă cazurile de testare implementate, structurate în funcție de adresa instrucțiunii din memoria ROM:

Rezultatele simulării pot fi vizualizate în figurile de mai jos :







După analiza rezultatelor simulării, se observă că circuitul implementat răspunde corect tuturor cererilor, furnizând rezultatele așteptate pentru fiecare dintre operațiile aritmetico-logice prevăzute în proiect.

VIII. Concluzii

Proiectul prezent demonstrează modul în care pot fi soluționate operațiile pe numere întregi întru-un sistem de calcul, prin proiectarea și implementarea unei componente esențiale unei unități centrale de prelucrare : unitatea aritmetico-logică.

Implementarea circuitului a evidențiat faptul că operațiile uzuale asupra numerelor întregi reprezintă o provocare complexă pentru unitatea centrală, iar UAL trebuie să asigure atât flexibilitate cât și precizie și corectitudine în furnizarea rezultatelor necesare celorlalte componente ale sistemului.

Din perspectiva dezvoltatorului, realizarea unității aritmetico-logice a oferit posibilitatea de a înțelege mai bine manipularea numerelor întregi reprezentate în complement față de 2, consolidând totodată conceptele fundamentale ale proiectării și implementării componentelor hardware folosind VHDL. De asemenea, proiectul a oferit experiență practică în programarea dispozitivelor FPGA, utilizând în acest caz placa Nexys A7, bazată pe familia FPGA Artix.

IX. Bibliografie

[1] David Harris, Sarah Harris, Digital Design and Computer Architecture, 2nd Edition, Morgan Kaufmann , 2012, [Online] (accesat pe 18 Octombrie 2025)

[2] M. Morris Mano, Digital Design – With an Introduction to the Verilog HDL, 6th Edition, Pearson, 2013, [Online] (accesat pe 18 Octombrie 2025)

[3] Charles Roth, Fundamentals of Logic Design, 7th Edition, Cengage Learning, 2013, [Online] (accesat pe 17 Octombrie 2025)

[4] IEEE Std 1076 – 2008, VHDL Language Reference Manual, [Online] (accesat pe 18 Octombrie 2025)

[5] Baruch Z.F., Arithmetic Logic Unit – Restoring Division, [Online] (accesat pe 01 Noiembrie 2025)

[6] Baruch Z.F. , Arithmetic Logic Unit – Shift-and-Add Multiplication, [Online] (accesat pe 01 Noiembrie 2025)

[7] Wakerly J.F. , Digital Design : Principles and Practices, [Online] (accesat pe 31 Octombrie 2025)