Circuit de Filtrare a Semnalelor Digitale

Documentație proiect la disciplina

Structura Sistemelor de Calcul

Kőrössy Zsolt, Grupa 30325

Universitatea Tehnică din Cluj-Napoca

2024

Cuprins

Introducere

1.1 Domeniu și context

Scopul acestei lucrări este proiectarea și implementarea unui filtru digital de ordinul 2 pe un circuit hardware programabil (FPGA), care transformă o secvență de valori de intrare X(i) pe baza unei formule de genul:

$$Y(k) = X(k) \cdot a_1 + X(k\text{-}1) \cdot a_2 + X(k\text{-}2) \cdot a_3,$$

a_1, a_2 și a_3 fiind valori constante arbitrare și *Y* fiind valoarea de ieșire. Având în vedere funcția de filtrare aplicată secvenței valorilor de intrare, filtrul implementat în acest proiect este de tip FIR (**F**inite **I**mpulse **R**esponse) [1], ceea ce înseamnă că circuitul are un răspuns la impuls de durată finită pentru orice secvență de intrare de lungime finită.

Filtrele FIR digitale se bazează exclusiv pe convoluția dintre un nucleu și valorile de intrare. Acest lucru presupune un filtru nerecursiv (opus filtrelor IIR (**I**nfinite **I**mpulse **R**esponse), care sunt recursive), în sensul că valorile de ieșire sunt calculate folosind numai intrarea curentă și cele anterioare. Circuitul nu utilizează valorile anterioare ale ieșirii, deci nu există feedback în structura filtrului.

1.2 Motivație și obiective

Circuitul va fi implementat în limbajul de descriere hardware VHDL, cu ajutorul aplicației Xilinx Vivado. De asemenea, va fi furnizat un testbench pentru simulare și pentru verificarea corectitudinii.

Motivația principală a acestui proiect este folosirea noțiunilor și cunoștințelor atât din descriere hardware, cât și din teoria sistemelor, pentru implementarea circuitului aferent. În același timp, este și exploatarea avantajelor implementării filtrelor digitale direct pe circuitele FPGA, lucru care oferă viteză mare de procesare, flexibilitate și consum redus de resurse în comparație cu soluțiile software tradiționale, care pot fi limitate de viteza procesorului sau de latențele introduse de sistemul de operare.

Obiectivele specifice acestui proiect includ:

- Proiectarea și implementarea unui filtru digital FIR care transformă o secvență de valori de intrare pe baza formulei matematice date
- Verificarea funcționării corecte a filtrului prin simulări și teste, pentru a asigura conformitatea cu specificațiile date
- Optimizarea circuitului pentru a utiliza eficient resursele FPGA și pentru a obține un timp de răspuns cât mai rapid

Studiu bibliografic

2.1 Ce este un filtru FIR digital?

Un filtru FIR digital este un tip de filtru utilizat pentru prelucrarea semnalelor digitale al cărui răspuns la impuls este finit, deci se încheie după un număr finit de eșantioane (sample-uri). Un filtru FIR digital este practic o operație de convoluție dintre un nucleu (un set de coeficienți de filtru, numiți și ponderile asociate fiecărei eșantion, în cazul nostru a_1, a_2 și a_3) și valorile de intrare, ceea ce înseamnă că elementele respective a două șiruri sunt înmulțite și toate produsele sunt însumate pentru a produce valoarea de ieșire. Datorită faptului că filtrul nu are elemente de feedback, stabilitatea este întotdeauna garantată.

Deoarece filtrul este de ordinul 2 (N-1, unde N=3 este numărul de coeficienți de filtru), răspunsul la impuls durează 3 (N=3) eșantioane sau sample-uri (de la primul element diferit de zero până la ultimul element diferit de zero), înainte ca acesta să se stabilească apoi la zero.

2.2 Transformata Z

În matematică și în procesarea semnalelor, transformata Z convertește un semnal în timp discret, care este o secvență de numere reale sau complexe, într-o reprezentare complexă în domeniul frecvenței discrete (domeniul Z). Analiza în domeniul Z ne ajută și în ajustarea coeficienților (a_1, a_2, a_3) pentru a obține caracteristicile dorite ale filtrului, dacă este cazul.

Transformata Z este un instrument esențial în proiectarea filtrelor digitale, deoarece oferă o bază solidă pentru analiza și optimizarea filtrului, facilitând trecerea de la specificațiile inițiale la implementarea hardware.

Având în vedere formula matematică pe care se bazează filtrul nostru,

$$Y(k) = X(k) \cdot a_1 + X(k-1) \cdot a_2 + X(k-2) \cdot a_3,$$

și aplicând transformata Z pe fiecare termen (ținând cont că transformata Z a unui semnal $X(k-n)$ este $Z[X(k-n)] = X(z)*z^{-n}$, unde $X(z)$ este transformata Z a semnalului original $X(k)$; z-urile sunt numite și elemente de întârziere) obținem următoarea formulă în domeniul frecvenței:

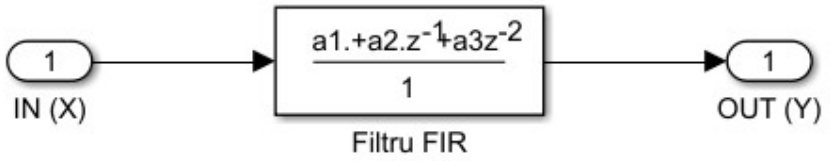
$$Y(z) = a_1 \cdot X(z) + a_2 \cdot X(z) \cdot z^{-1} + a_3 \cdot X(z) \cdot z^{-2}$$

2.3 Funcția de transfer

Având calculată formula filtrului în domeniul Z și factorizând $X(z)$, obținem formula de transfer $H(z)$, care este raportul dintre ieșirea $Y(z)$ și intrarea $X(z)$:

$$H(z) = \frac{Y(z)}{X(z)} = a_1 + a_2 \cdot z^{-1} + a_3 \cdot z^{-2}$$

Folosind funcția de transfer, putem determina răspunsul în frecvență al filtrului prin evaluarea *H(z)* pe cercul unitate din planul complex ($z = e^{j\omega}$), unde ω este frecvența unghiulară. Răspunsul ne arată cum sunt amplificate sau atenuate diferitele componente de frecvență ale semnalului de intrare. Transformata Z și funcția de transfer ajută la simularea comportamentului filtrului înainte de implementarea sa efectivă în VHDL.



Analiză și design

3.1 Blocurile de construcție ale filtrului

Înainte de a discuta avantajele și dezavantajele implementărilor în serie și paralele, și de a analiza schemele bloc, vom specifica blocurile de construcție din care sunt alcătuite. [2]

3.1.1 Elemente de memorie

În implementarea filtrului avem nevoie de **unități de stocare a memoriei** care dețin z-cantitate de eșantioane de date consecutive (pătratele din schema bloc). De exemplu, un element de întârziere z^{-1} deține o valoare, în timp ce z^{-n} deține n valori (în cazul nostru maximul este n=2). Când există mai multe elemente de întârziere într-un rând, datele sunt transferate (shiftate) de la un element la următorul o dată pe perioadă de eșantionare.

3.1.2 Operatori aritmetici

Operațiile aritmetice implicate în filtrul nostru FIR digital sunt **adunarea** (cercurile din schema bloc) și **înmulțirea** (triunghiurile). Acestea sunt unele dintre cele mai comune operații aritmetice și sunt ușor de implementate eficient pe plăcile FPGA.

3.1.3 Rata de eșantionare

Aceasta este rata la care datele sunt procesate de filtru, cunoscută și ca frecvență de eșantionare sau ceas/perioadă de eșantionare. În această perioadă toate operațiile (de memorie și aritmetice) sunt executate o singură dată.

3.2 Implementare paralelă

Spre deosebire de software, unde filtrul trebuie descris într-o manieră secvențială, în HDL avem libertatea de a-l implementa fie în paralel, fie în serial.

Ca modalitate de reprezentare a filtrului, schema bloc este o descriere directă a procesului de convoluție, specificând calea semnalului și operațiile efectuate pe acesta, dar nu și modul în care acestea sunt implementate. Cel mai frecvent, filtrele FIR sunt exprimate prin structura de filtru Direct Form. Schema bloc este formată din cele trei elemente menționate anterior, a căror implementare va fi discutată în continuare.

În acest capitol discutăm avantajele și dezavantajele ambelor implementări, nu implementarea efectivă a vreunuia dintre ele.

3.2.1 Implementare paralelă clasică

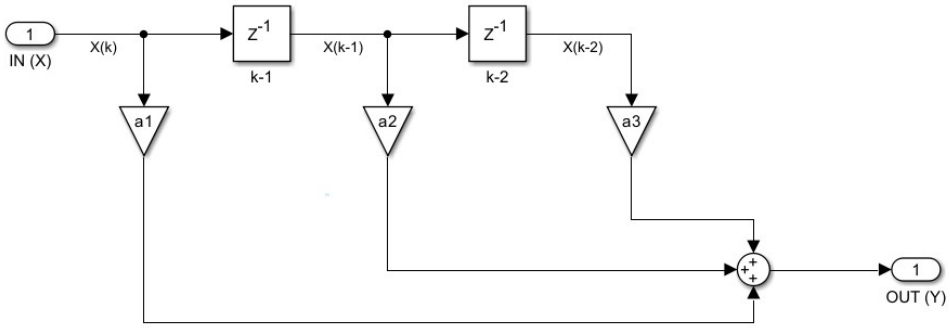


Figura 1. Schema bloc Direct Form al filtrului

Deși schema bloc nu spune cum ar trebui implementat filtrul, cu versiunea paralelă descriem practic calea semnalului din diagramă, putând folosi registre pentru elementele de memorie.

Capcana acestei implementări de filtru este adăugarea. De obicei, adăugarea a 2 sau mai multe semnale este descrisă cu o operație de însumare ca în *Figura 1* (sumătorul înainte de out). Deși filtrul nostru este foarte mic și acest lucru nu ar afecta foarte mult performanța și costurile resurselor (și este o soluție fezabilă în VHDL), nu este cea mai practică modalitate de a implementa filtrul. Calea logică dintre registrul de intrare și de ieșire poate deveni prea lungă și poate duce la probleme de sincronizare.

Problema poate fi rezolvată prin pipelining, ceea ce înseamnă procesul de scurtare a căii dintre registre prin rearanjarea designului sau inserarea registrelor între operații fără a modifica funcționalitatea circuitului.

3.2.2 Implementare paralelă – însumare și pipelining

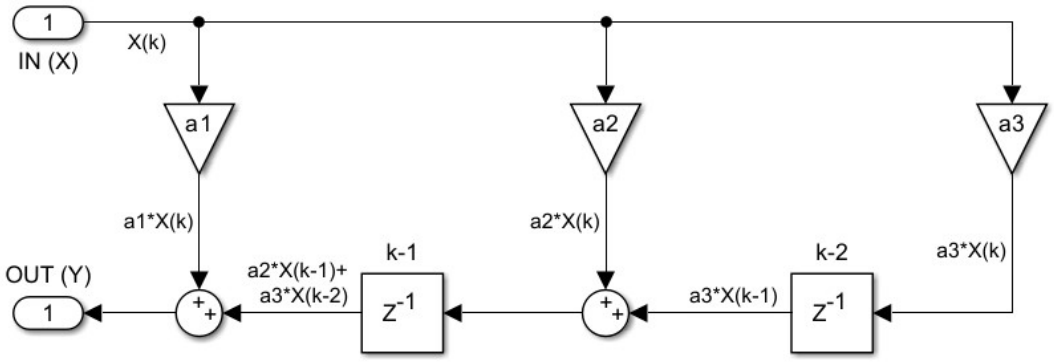


Figura 2. Schema bloc al filtrului transpus (transposed)

Această versiune a filtrului este mult mai favorabilă pentru implementarea HDL, deoarece rezolvă problema prin împărțirea adăugării în mai multe etape separate de un registru. De

asemenea, se simplifică și partea de , în sensul că acum doar valoarea de intrare este implicată în operația de înmulțire.

3.3 Implementare serială

Când ceasul sistemului rulează mult mai repede decât rata de eșantionare a datelor, este mult mai bine să descompunem procesul de convoluție în pași și să le executăm pe rând. Aceasta poate fi implementată ca o combinație între o mașină de stări finite (FSM), un numărător și instrucțiune If.

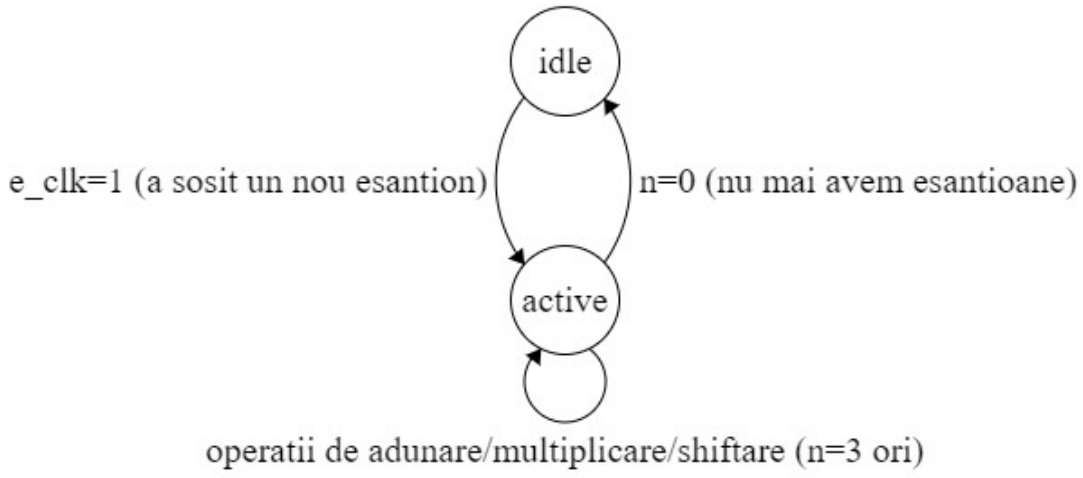


Figura 3. Mașina de stări finite pentru implementarea serială
(e_clk – rata de eșantionare, n – numărător)

De fiecare dată când sosește un eșantion (sample) nou, mașina de stări trece de la starea inactivă (idle) la cea activă (active), unde numărătorul trece de la N=3 la 0. Pornind contorul de la n=N, codul efectuează operația de înmulțire și de adunare asupra datelor stocate în elementul de memorie curent N=3 și transferă datele de la elementul N-1=2 la elementul N=3. Apoi contorul se deplasează la elementul N-1=2, iar operația se repetă până când contorul ajunge la n=0. La n=0 nu mai există memorie din care să citească, deci datele sunt alocate elementului de memorie al adresei 0, iar mașina de stări trece în starea inactivă.

3.4 Implementarea paralelă vs. serială

3.4.1 Rata de eșantionare

Implementarea paralelă poate funcționa la rate de eșantionare egale cu ceasul sistemului, și, prin urmare, poate fi utilă în implementări de mare viteză. Când nu este necesară viteză ridicată putem implementa serial circuitul, împărțind procesul de convoluție în pași pe care le putem executa pe rând.

3.4.2 Resurse aritmetice

În versiunea paralelă, fiecare operație de înmulțire și de adunare necesită hardware dedicat. Deși nu este o problemă în cazul filtrului nostru de dimensiuni reduse, proiectarea unor filtre FIR extinse poate consuma cu ușurință o mare parte din resursele DSP (Digital Signal Processing) [3] disponibile. Acest lucru, însă, nu este cazul la implementarea serială, în care filtrul alimentează intrarea și coeficienții la același multiplicator, reducând astfel necesarul de resurse aritmetice. Implementarea serială ne permite construirea filtrelor de ordin mult mai înalt.

Secțiunile DSP (DSP slice) sunt elemente logice de procesare a semnalelor digitale incluse pe anumite familii de dispozitive FPGA (ex. circuitul Nexys4 are 240 de secțiuni DSP, Basys3 90). Ele pot fi utilizate pentru a efectua eficient diferite tipuri de operații aritmetice, inclusiv înmulțirea.

3.4.3 Resurse de memorie

Elementele de întârziere din filtrul paralel sunt implementate printr-o serie de registre, un lucru necesar pentru a permite accesul simultan la linia de întârziere. În cazul filtrului FIR transpus, elementele de memorie necesare pentru linia de întârziere și pipelining sunt conținute în secțiunile DSP, reducând și mai mult cerințele de resurse. Totuși, în cazul implementării seriale, datele sunt alimentate secvențial, rezultând avantajul utilizării blocurilor RAM pentru a stoca datele, în loc de registre.

În ceea ce privește coeficienții, în implementarea paralelă, dacă coeficienții sunt fixe (constanți, ca în cazul nostru), sunt șanse ca sintetizatorul să nu ia niciun registru pentru aceștia, întrucât intrările respective pe multiplicatori ar fi legate la valori constante. Implementarea serială necesită o memorie dedicată pentru coeficienți, care poate fi fie într-o memorie RAM distribuită, fie intr-o BRAM.

3.4.4 Concluzii

În alegerea implementării nu există un răspuns greșit sau corect, ci adaptarea acestuia la nevoile noastre. Implementarea paralelă aduce o viteză foarte mare, cu prețul numărului mare de componente. Odată cu implementarea serială putem reutiliza anumite componente și folosim memorie RAM în loc de registre, dar avem o funcționare mult mai lentă.

Implementare

4.1 Considerații de implementare

Varianta filtrului FIR digital pe care o vom implementa este cea paralelă, îmbunătățită cu însumare și pipelining. Deși avem un filtru foarte mic și nu influențează foarte mult această alegere performanța, implementarea paralelă este cea care exploatează cel mai mult avantajele oferite de FPGA, având posibilitatea unor soluții concurente, nu doar seriale. Cu această versiune de implementare, avem opțiunea de a avea o frecvență de eșantionare egală cu frecvența plăcii FPGA. În același timp, este probabil și varianta cea mai ușor de implementat, deoarece putem urmări diagrama bloc foarte îndeaproape.

În descrierea componentelor (circuitelor) necesare în VHDL, m-am concentrat pe utilizarea cât mai multor atribute generice posibil pentru a realiza un șablon de filtru FIR digital, mai degrabă decât un filtru concret cu trei coeficienți.

4.2 Componentele filtrului

4.2.1 Divizorul de frecvență

Divizorul de frecvență ne ajută atunci când avem nevoie de un ceas mai lent decât cel din placa FPGA. În cazul nostru, această componentă va fi folosită pentru a seta frecvența de eșantionare, dacă nu vrem să folosim ceasul intern.

```
36  entity divizor_clk is
37      Generic(
38          generated_freq : INTEGER := 1
39      );
40      Port ( clk : in STD_LOGIC;
41             rst : in STD_LOGIC;
42             new_clk : out STD_LOGIC
43      );
44  end divizor_clk;
```

În afară de ceasul de intrare original, noul ceas de ieșire cu frecvența dorită și un semnal de resetare asincronă, avem și o valoare generică *generated_freq*, care este practic frecvența dorită, dată în Hz.

```
46  architecture Behavioral of divizor_clk is
47
48  signal counter: integer := 0;
49  signal counter_max: integer := 1_000_000_000 / generated_freq / 20 - 1;
50  signal clk_aux: std_logic := '0';
51
52  begin
53
54  new_clk <= clk_aux;
55
56  process(clk, rst)
57  begin
58
59  if rst = '1' then
60      counter <= 0;
61      clk_aux <= '0';
62  elsif rising_edge(clk) then
63      if counter < counter_max then
64          counter <= counter + 1;
65      else
66          counter <= 0;
67          clk_aux <= not(clk_aux);
68      end if;
69  end if;
70
71  end process;
72
73  end Behavioral;
```

Divizorul de frecvență funcționează prin numărarea de la 0 la valoarea *counter_max*, incrementând contorul la fiecare front ascendent al ceasului de intrare. Când contorul atinge această valoarea, este resetat la 0 și ceasul de ieșire devine valoarea sa curentă negat.

Deoarece acest proiect este realizat având în vedere placa FPGA Basys3, se presupune că avem un ceas intern de 100MHz (=100.000.000Hz). Dacă nu folosim acesta (deși posibil în implementarea pipeline), valoarea maximă la care putem împărți frecvența noastră este 50Mhz (pentru o frecvență peste acea *counter_max* ar trebui să aibă o valoare sub 1, deoarece este de fapt jumătate din perioada de ceas), deci *generate_freq* trebuie să fie în intervalul [1, 50.000.000]. Valoarea lui *counter_max* este calculată automat în funcție de această frecvență dată.

4.2.2 Generatorul de monoimpuls

Generatorul de monoimpuls (debouncer) ne asigură că la apăsarea unui buton avem un semnal stabil de o anumită perioadă, nu unul oscilant. Această componentă este folosită pentru butoanele de resetare și de activare (enable), dacă încărcăm filtrul pe un FPGA.

Debouncerul funcționează cu ajutorul unui contor, care trebuie să atingă o anumită valoare, înainte ca datele de intrare obținute prin apăsarea butonului (*btn*) să fie transmise la semnalul stabil de ieșire *en*.

```vhdl
entity debouncer is
    Port ( clk : in STD_LOGIC;
           btn : in STD_LOGIC;
           en : out STD_LOGIC);
end debouncer;

architecture Behavioral of debouncer is

signal counter : std_logic_vector(15 downto 0) := (others => '0');
signal Q1, Q2, Q3 : std_logic := '0';

begin
    en <= Q2 and not(Q3);

    process(clk)
    begin
        if rising_edge(clk) then
            counter <= counter + 1;

            if counter = "1111111111111111" then
                Q1 <= btn;
            end if;

            Q2 <= Q1;
            Q3 <= Q2;
        end if;

    end process;
```

4.2.3 Filtrul FIR digital

Filtrul în sine este un circuit care primește un semnal de intrare digital (șir de biți) la fiecare tact de ceas, în funcție de frecvența de eșantionare, și dă un semnal de ieșire, în funcție de intrări și de coeficienții de filtru. Când primele valori de intrare intră în filtru, avem câteva perioade de ceas (în cazul nostru 2, numărul de elemente de întârziere) când nu există semnal de ieșire, deoarece intrările trebuie să treacă prin toate elementele de întârziere înainte de a începe să producă rezultate (filtrul trebuie să devină saturat). După aceste perioade inițiale, filtrul va produce un semnal de ieșire la fiecare ciclu de ceas.

```vhdl
entity filter_FIR is
    Generic( input_width : INTEGER range 4 to 32 := 8;
             coeff_width : INTEGER range 4 to 32 := 8;
             nr_of_coeffs : INTEGER range 1 to 60 := 3;
             sampling_rate : INTEGER range 50 to 50_000_000 := 50_000_000); --dat in Hz
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           en : in STD_LOGIC;
           input_data : in STD_LOGIC_VECTOR (input_width - 1 downto 0);
           output_data : out STD_LOGIC_VECTOR (input_width + coeff_width - 1 downto 0));
end filter_FIR;
```

Semnalele de intrare ale filtrului sunt ceasul (*clk*), datele de intrare (*input_data*), un semnal de resetare asincronă (*rst*) și un semnal de activare (*en*). Semnalul de ieșire (*output_data*) are lățimea sumei lățimilor semnalului de intrare și a coeficienților.

Genericele *input_width* și *coeff_width* ne ajută să specificăm dimensiunea șirurilor de biți pentru intrări și coeficienți în momentul instanțierii filtrului, în loc să hardcodăm aceste informații. În mod similar, putem specifica numărul de coeficienți (totuși, cu modificarea acestuia trebuie să și setăm valorile fiecărui coeficient unul câte unul) și frecvența de eșantionare (*sampling_rate*). Intervalul de constrângere pentru cea din urmă este specificat aici.

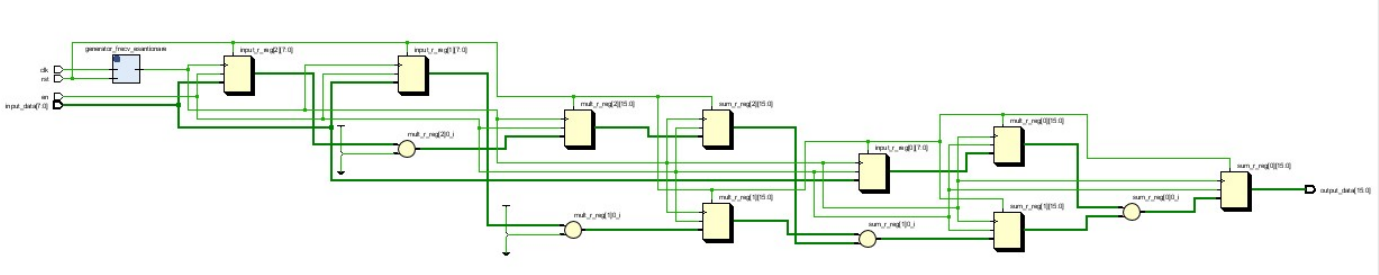
Pe lângă includerea componentei divizor de frecvență în partea de arhitectură, definim și tipurile necesare pentru stocarea și procesarea semnalelor intermediare, respectiv a celui final. Urmând schema bloc din proiectare, avem nevoie de registre pentru fiecare semnal de intrare, pentru stocarea valorilor acestora înmulțite cu coeficienții de filtru, pentru rezultatele obținute după însumări și pentru coeficienții înșiși. Aici precizăm și valorile coeficienților, respectiv aici declarăm semnalul ceasului de eșantionare (*clk_fs*).

```
52  component divizor_clk is
53  generic( generated_freq : INTEGER --Basys3 clock speed 100MHz
54  );
55  port ( clk : in STD_LOGIC;
56        rst : in STD_LOGIC;
57        new_clk : out STD_LOGIC
58  );
59  end component;
60
61  type input_regs is array(0 to nr_of_coeffs - 1) of unsigned(input_width - 1 downto 0);
62  type mult_regs is array(0 to nr_of_coeffs - 1) of unsigned(input_width + coeff_width - 1 downto 0);
63  type dsp_regs is array(0 to nr_of_coeffs - 1) of unsigned(input_width + coeff_width - 1 downto 0);
64  type coeffs is array(0 to nr_of_coeffs - 1) of unsigned(coeff_width - 1 downto 0);
65
66  signal input_r: input_regs := (others => (others => '0')); -- registre pentru semnalul de intrare
67  signal mult_r: mult_regs := (others => (others => '0')); -- registre de multiplicare
68  signal sum_r: dsp_regs := (others => (others => '0')); -- registre de insumare
69  signal coeffs_r: coeffs := (
70  0=>x"0001", -- a1
71  1=>x"0002", -- a2
72  2=>x"0003"  -- a3
73  ); -- coeficientii filtrului
74
75  signal clk_fs: std_logic := '0'; --frecventa de esantionare
```

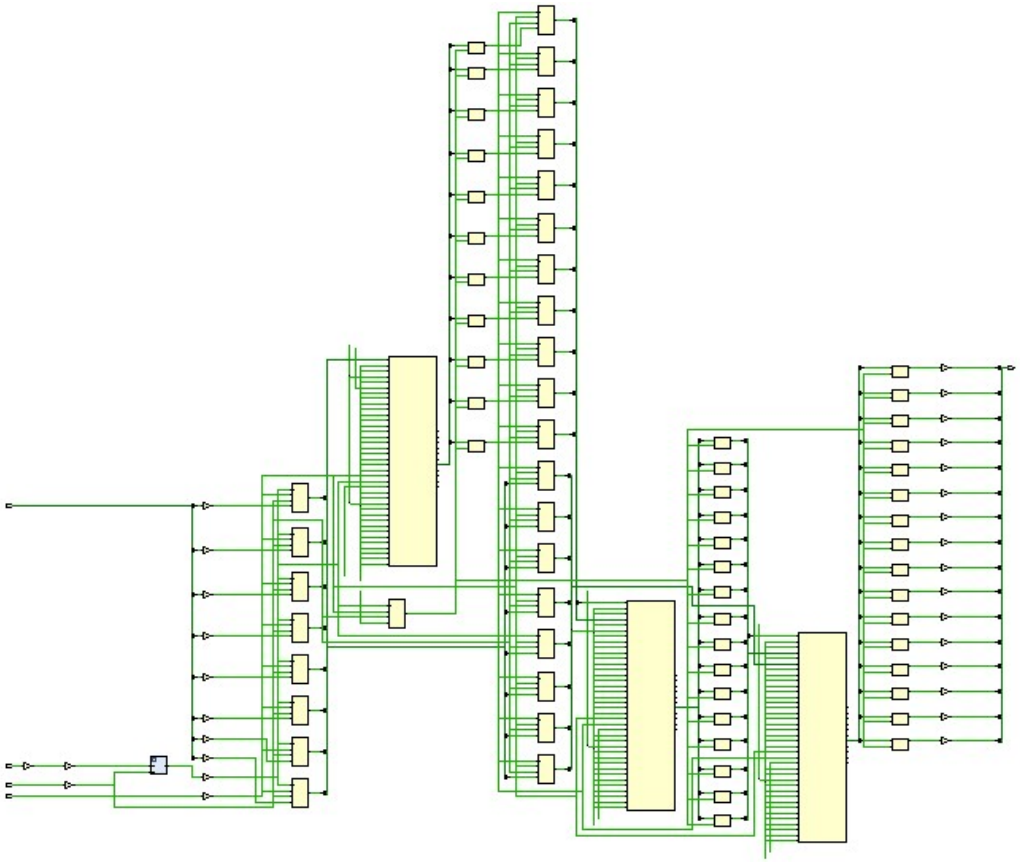
```
77  begin
78
79  output_data <= std_logic_vector(sum_r(0));
80  clk_fs <= clk; --clk, daca folosim clock-ul circuitului FPGA (frecventa nedivizata = 100MHz)
81
82  filter: process(rst, clk_fs)
83  begin
84  if rst = '1' then
85      for i in 0 to nr_of_coeffs-1 loop
86          input_r(i) <= (others => '0');
87          mult_r(i) <= (others => '0');
88          sum_r(i) <= (others => '0');
89      end loop;
90  elsif rising_edge(clk_fs) and en = '1' then
91      for i in 0 to nr_of_coeffs - 1 loop
92          input_r(i) <= unsigned(input_data);
93
94          mult_r(i) <= input_r(i) * coeffs_r(i);
95
96          if i < nr_of_coeffs - 1 then
97              sum_r(i) <= mult_r(i) + sum_r(i+1);
98          else
99              sum_r(i) <= mult_r(i);
100         end if;
101     end loop;
102 end if;
103
104 end process;
```

Procesul **filter** descrie funcționalitatea propriu-zisă a filtrului. Semnalul de resetare este asincron și setează valorile fiecărui registru la 0.

În funcție de frontul ascendent al ceasului de eșantionare și de semnalul de activare are loc o buclă for, în care valorile fiecărui etaj sunt calculate și transferate la etajul următor. Aceste calcule presupun înmulțiri, adunări și scriere în registre, fiecare întâmplându-se în fiecare tact de ceas. Rezultatul, semnalul de ieșire, este dat de primul registru de însumare, care conține suma valorii curente calculate în momentul k, cu valorile sumelor la momentul k+1 și k+2. Din acest motiv, nu avem semnal de ieșire în primele 2 perioade de ceas, deoarece acele registre cu valori viitoare încă nu au nimic calculate în ele, și prima valoare calculată trebuie să se propagă prin toate elementele de întârziere. După acest timp inițial, filtrul devine saturat și va da un semnal de ieșire în fiecare ciclu de ceas.



Pe diagramă putem observa designul pipeline al filtrului, foarte asemănător cu ce am avut ca design inițial. Secțiunile (slice) DSP menționate mai sus, care măresc mult performanța filtrului, fiind astfel principalul avantaj al implementării hardware, sunt deduse automat din cod în timpum generării implementării, unde avem operații de multiplicare. Cu toate acestea, dacă vrem să maximizăm utilizarea secțiunilor DSP în implementare, putem indica acest lucru setând atributul *use_dsp* la “yes”.



După cum putem observa, pentru operațiile de multiplicare tool-ul a folosit automat secțiuni DSP în implementarea circuitului (dreptunghurile mari), care permit procesarea semnalului la frecvența plăcii. Dacă vrem cu adevărat să rezolvăm tot cu secțiuni DSP, le putem configura și instanția manual și din catalogul IP. Pentru filtrul nostru de dimensiuni mici, acest lucru este inutil și mult mai complex decât implementarea actuală a procese.

4.2.4 Entitatea principală pentru test

Această entitate este menită să creeze o interfață mai ușoară pentru faza de testare waveform prin instanțierea unui filtru în ea, dând valori parametrilor generici. Este același test ca cel de pe filtru în sine, dar cu mai multe valori de intrare și șiruri de 16 biți lățime de data aceasta.

```
34  entity main_entity is
35      Port ( clk : in STD_LOGIC;
36             rst_btn : in STD_LOGIC;
37             en_btn : in STD_LOGIC;
38             input_data : in STD_LOGIC_VECTOR (7 downto 0);
39             output_data : out STD_LOGIC_VECTOR (15 downto 0));
40  end main_entity;
```

```
begin

filter: filter_FIR
    generic map(
        input_width => 16,
        coeff_width => 16,
        nr_of_coeffs => 3,
        sampling_rate => 50 --nu conteaza valoarea, pentru testbench folosim clk-ul intern fara divizare
    )
    port map(
        clk => clk,
        rst => rst_btn,
        en => en_btn,
        input_data => input_data,
        output_data => output_data
    );
```

4.2.5 Entitatea principală pentru FPGA (Basys3)

Scopul acestui modul este de a putea încărca filtrul pe un FPGA (Basys3 în cazul nostru) astfel încât să putem verifica funcționalitatea acestuia pe hardware, nu doar în simularea software.

```
34  entity Basys3_module is
35  Port (  clk: in STD_LOGIC;
36          sw: in STD_LOGIC_VECTOR(7 downto 0);
37          btn: in STD_LOGIC_VECTOR(4 downto 0);
38          led : out STD_LOGIC_VECTOR (15 downto 0);
39          an : out STD_LOGIC_VECTOR (3 downto 0);
40          cat : out STD_LOGIC_VECTOR (6 downto 0)
41          --outt: out STD_LOGIC_VECTOR(15 downto 0)
42      );
43  end Basys3_module;
```

Pe lângă maparea porturilor componentelor utilizate și în simularea software (filtrul, care include și divizorul de ceas), generatorii de monoimpuls își găsesc și ei utilizarea în acest modul, precum și afișorul cu 7 segmente al plăcii. Butoanele de activare și de resetare au ambele alocate unul, așa că prin apăsarea unuia nu va rezulta un semnal oscilant. Acest lucru este important deoarece implementarea hardware a filtrului seamănă mai mult cu una asincronă, spre deosebire de simularea software (unde viteza filtrului este dată de ceas), deoarece introducem semnalele de intrare setând comutatoarele (switch-urile) FPGA-ului pe poziția corectă și introducând aceste valori prin apăsarea butonului de activare.

Rezultatele pot fi văzute pe afișorul cu 7 segmente (în acest fel, semnalele de ieșire obținute vor fi sub 16 biți).

```
80  begin
81      input_data <= sw(7 downto 0);
82      --outt <= output_data;
83
84      en: debouncer port map (
85          clk => clk,
86          btn => btn(0),
87          en => en_btn
88      );
89
90      rst: debouncer port map (
91          clk => clk,
92          btn => btn(1),
93          en => rst_btn
94      );
95
96      ssd: display_7seg port map (
97          digit0=>output_data(3 downto 0),
98          digit1=>output_data(7 downto 4),
99          digit2=>output_data(11 downto 8),
100         digit3=>output_data(15 downto 12),
101         clk=>clk,
102         cat=>cat,
103         an=>an
104     );
```

```
106     filter: filter_FIR
107     generic map(
108         input_width => 8,
109         coeff_width => 8,
110         nr_of_coeffs => 3,
111         sampling_rate => 50 --nu conteaza valoarea, deocamdata folosim ceasul intern
112     )
113     port map(
114         clk => clk,
115         rst => rst_btn,
116         en => en_btn,
117         input_data => input_data,
118         output_data => output_data
119     );
```

Testare și validare

5.1 Simulare waveform

Etapa de testare software implică rularea unor simulări pe modulul top-level, precum și pe fiecare dintre componentele individuale. În proiect sunt incluse testbench-uri separate pentru testarea filtrului și a divizorului de ceas, oferind simulări mai concise pentru aceste componente, totuși, ne vom concentra pe testarea modulului principal, care practic include testarea tuturor componentelor în același timp.

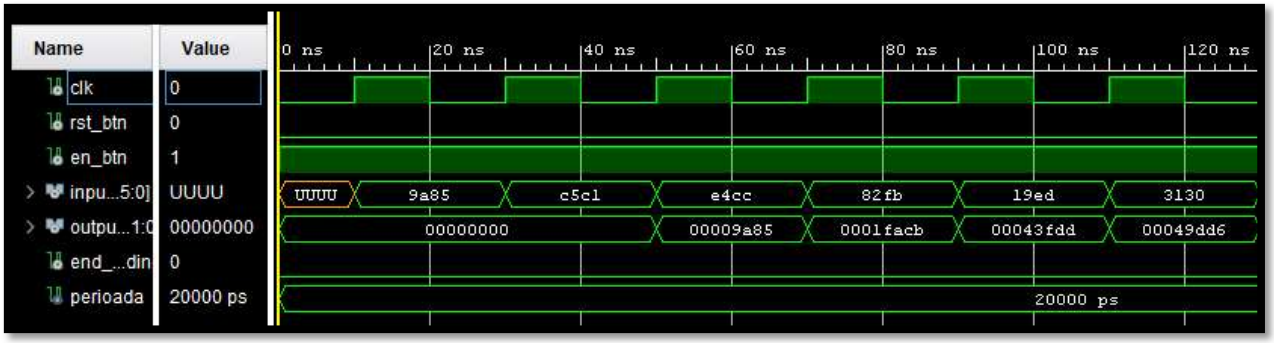
```
64  begin
65
66      main_entity_TB: main_entity port map( --coeficientii sunt a1=1, a2=2, a3=3
67          clk => clk,
68          rst_btn => rst_btn,
69          en_btn => en_btn,
70          input_data => input_data,
71          output_data => output_data
72      );
73
74      clk <= not clk after perioada / 2;
75
76      process (clk)
77          file inputs : text open read_mode is "input_data.txt";
78          variable in_line : line;
79
80          variable read_data : std_logic_vector(15 downto 0);
81          variable space : character;
82      begin
83          if rising_edge(clk) then
84              if end_of_reading = '0' then
85                  if not endfile(inputs) then
86                      readline(inputs, in_line);
87                      read(in_line, read_data);
88
89                      input_data <= read_data;
90                  else
91                      file_close(inputs);
92                      end_of_reading <= '1';
93                  end if;
94              end if;
95          end if;
96      end process;
97
98  end Behavioral;
```

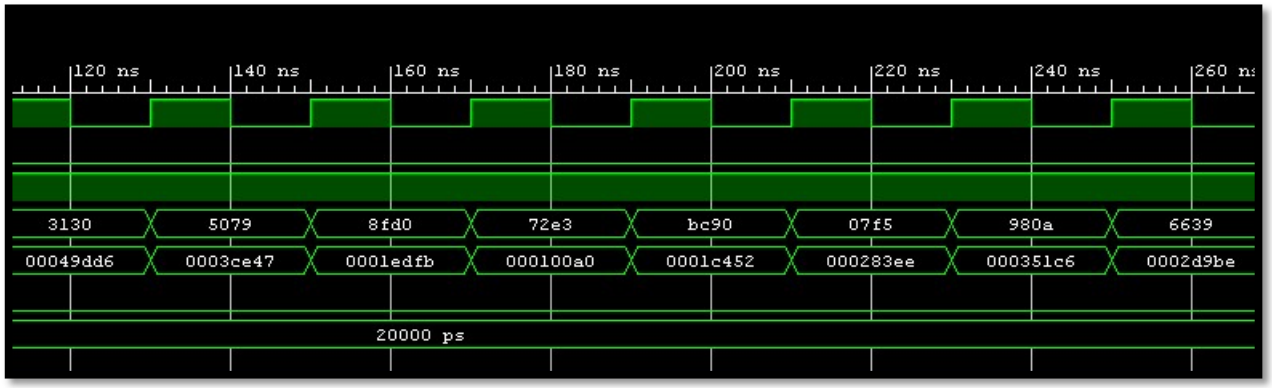
Pentru a testa modulul principal, folosim un fișier text de intrare care conține date sub formă de șiruri de biți (un șir pe fiecare linie), simulând astfel semnale de intrare care pot fi

procesate de filtru. În același timp, se include și valoarea în hexazecimal pentru fiecare șir de biți, respectiv operația efectuată de filtru asupra datelor (pentru primele 10 valori citite) care sunt procesate în momentul citirii noii valori din fișier. Toate aceste operațiuni se fac într-un proces, pe fiecare front ascendent al ceasului de intrare (semnalul de activare are valoarea 1 logic pe toată durata simulării).

Pentru a testa filtrul folosim ceasul intern. Dacă vrem o altă frecvență de eșantionare, trebuie modificat codul care setează filtrul în interiorul descrierii filtrului (în acest caz trebuie să avem grijă la rata cu care introducem datele în filtru). Pentru simplitatea înțelegerii rezultatelor, valorile utilizate pentru coeficienții de filtru au fost a_1=1, a_2=2 și a_3=3.

```
1001101010000101, 9A85
1100010111000001, C5C1
1110010011001100, E4CC - 9A85*1 + 0000*2 + 0000*3 = 9A85
1000001011111011, 82FB - C5C1*1 + 9A85*2 + 0000*3 = C5C1 + 1350A = 1FACA
0001100111101101, 19ED - E4CC*1 + C5C1*2 + 9A85*3 = E4CC + 18B82 + 1CF8F = 43FFD
0011000100110000, 3130 - 82FB*1 + E4CC*2 + C5C1*3 = 82FB + 1C998 + 25143 = 49DD6
0101000001111001, 5079 - 19ED*1 + 82FB*2 + E4CC*3 = 19ED + 105F6 + 2AE64 = 3CE47
1000111111010000, 8FD0 - 3130*1 + 19ED*2 + 82FB*3 = 3130 + 33DA + 188F1 = 1EDFB
0111001011100011, 72E3 - 5079*1 + 3130*2 + 19ED*3 = 5079 + 6260 + 4DC7 = 100A0
1011110010010000, BC90 - 8FD0*1 + 5079*2 + 3130*3 = 8FD0 + A0F2 + 9390 = 1C452
0000011111110101, 07F5 - 72E3*1 + 8FD0*2 + 5079*3 = 72E3 + 11FA0 + F16B = 283EE
1001100000001010, 980A - BC90*1 + 72E3*2 + 8FD0*3 = BC90 + E5C6 + 1AF70 = 351C6
```

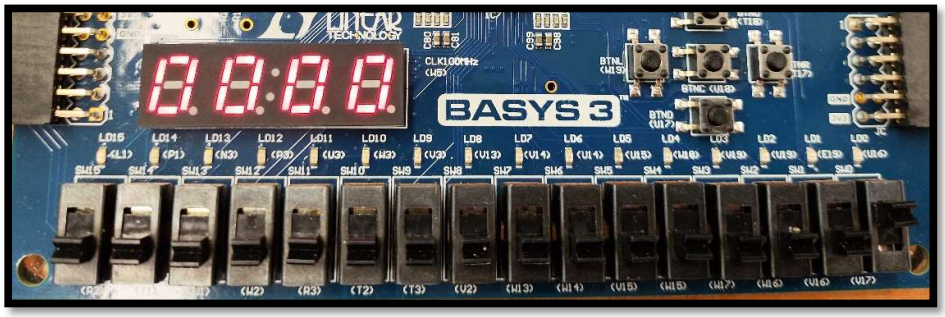


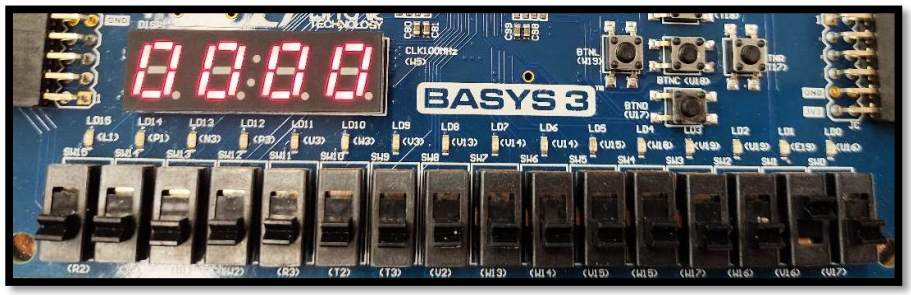


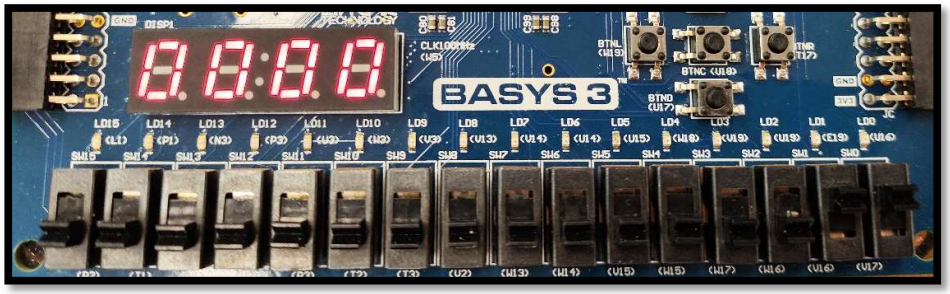
5.2 Testare pe FPGA (Basys3)

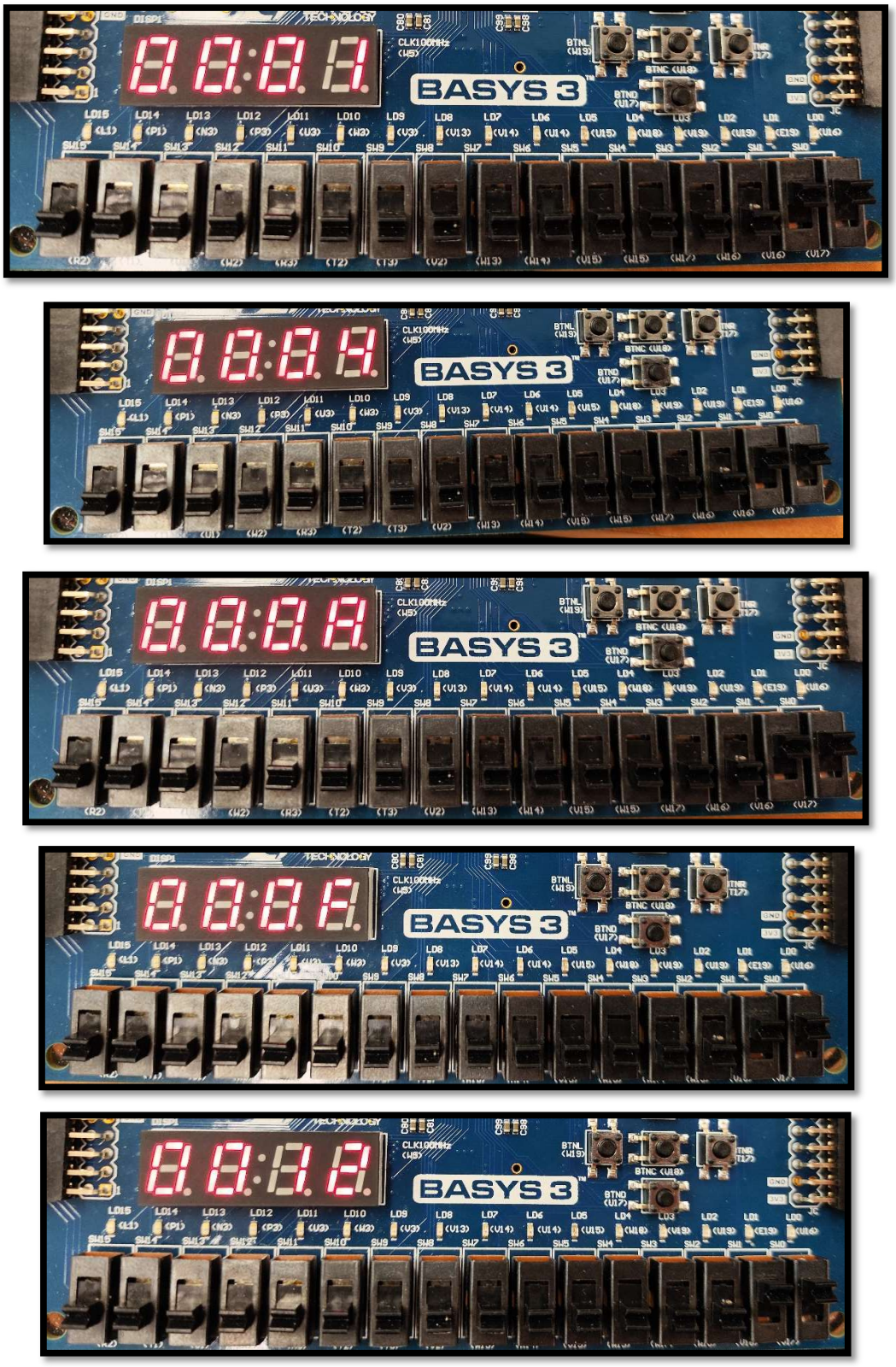
Pentru simplitate, valorile de intrare introduse în filtru prin comutatoare au fost 1, 2 și 3, în această ordine, și lăsând filtrul să se stabilizeze după ce nu schimbăm valoarea 3 timp de 3 cicluri. Putem observa și primele două perioade când nu avem semnal de ieșire, după care filtrul este saturat. Trasarea funcționării filtrului poate fi văzută mai jos, împreună cu rezultatele obținute pe FPGA.

```
1  00000001, 01
2  00000010, 02
3  00000011, 03 - 01*1 + 00*2 + 00*3 = 0001
4  00000011, 03 - 02*1 + 01*2 + 00*3 = 02 + 02 = 0004
5  00000011, 03 - 03*1 + 02*2 + 01*3 = 03 + 04 + 03 = 000A
6  00000011, 03 - 03*1 + 03*2 + 02*3 = 03 + 06 + 06 = 000F
7  00000011, 03 - 03*1 + 03*2 + 03*3 = 03 + 06 + 09 = 0012
8  00000011, 03 - 03*1 + 03*2 + 03*3 = 03 + 06 + 09 = 0012
9  ...
```









(Fiecare fotograf a fost făcut înainte ca butonul de activare să fie apăsat, astfel rezultatul pe SSD este cu 1 ciclu în urmă pentru fiecare în comparație cu trasarea.)

Concluzii

Procesarea semnalelor este piatra de temelie a tehnologiilor moderne, deoarece majoritatea instrumentelor și implementărilor noastre digitale folosesc semnale într-o anumită formă. Și odată cu creșterea performanței acestor sisteme, este mai important ca niciodată să avem o procesare eficientă a semnalelor. După cum am văzut, acest lucru poate fi implementat utilizând componente hardware specifice (în cazul nostru FPGA), cum ar fi secțiunile DSP, și exploatând posibilitatea de a proiecta circuite cu operații paralele, îmbunătățind astfel mult performanța operațiunilor în comparație cu soluțiile software.

Cu cunoștințe combinate din descriere hardware și teoria sistemelor, filtrul rezultat a fost, de asemenea, testat pe un flux de date simulate, iar rezultatele au fost comparate cu rezultatele calculate manual, asigurând funcționalitatea corespunzătoare a circuitului. Pe lângă îndeplinirea cerințelor inițiale, acest proiect s-a concentrat și pe asigurarea unui design cât mai generic, astfel încât cu modificări simple și puține să putem realiza filtre digitale FIR de ordine diferită, cu intrări și coeficienți de lățime diferitășau cu o altă frecvență de eșantionare.

Bibliografie

[1] „Know all About FIR Filters in Digital Signal Processing,” ElProCus, [Interactiv]. Available: https://www.elprocus.com/fir-filter-for-digital-signal-processing/.

[2] D. Marinov, „Part 2: Finite Impulse Response (FIR) filters,” 11 August 2023. [Interactiv]. Available: https://vhdlwhiz.com/part-2-finite-impulse-response-fir-filters/.

[3] „7 Series DSP48E1 Slice User Guide (UG479),” AMD, 27 March 2018. [Interactiv]. Available: https://docs.amd.com/v/u/en-US/ug479_7Series_DSP48E1.