Proiectarea unei unități aritmetice de tip MMX

Buruian Cătălina

Gr 30231

Cuprins

1. Introducere

1.1. Context

Având în vedere marele avantaj pe care îl are arhitectura MMX în cadrul aplicațiilor de multimedia și comunicare(procesare de imagini, grafică 2D/3D, sinteză audio, speech compression and synthesis etc.), cunoașterea în profunzime a acesteia reprezintă un imperativ în rândul pasionaților de hardware. Așadar, scopul acestui proiect este de a realiza o unitate aritmetica de tip MMX capabilă să efectueze 6 operații din cele regăsite în setul de instrucțiuni MMX x86.
Instrucțiunile MMX permit procesoarelor x86 să efectueze operații de tipul single-instruction, multiple-data(SIMD) pe operanzi întregi de tipul byte, word, doubleword, sau quadword conținuti în memorie, în registrele MMX sau în registrele de uz general.

1.2. Specificații

Funcționalitatea unității aritmetice de tip MMX va fi vizibilă cu ajutorul mediului de simulare oferit de Vivado, iar mai apoi întreg proiectul va fi testat pe plăcuța Basys 3. Utilizatorul va putea alege atât operația dorită, cât și operanzii pe care se vor efectua diferite operații cu ajutorul switch-urilor de pe plăcuță, iar rezultatul va putea fi vizibil pe afișorul cu 7 segmente. Din cauza posibilelor rezultate pe 32 sau 64 biți care nu vor putea fi afișate în întregime pe cele 4 afișoare ale plăcuței, se propune afișarea rezultatului câte patru cifre odată, prin apăsarea unui buton de către utilizator.

1.3. Obiective

Principalele obiective sunt înțelegerea arhitecturii MMX, a instrucțiunilor și tipurilor de date nou introduse, a modului de stocare a acestora și a modului în care este exploatat și introdus paralelismul, pentru ca, în final, să poată fi proiectată o astfel de unitate aritmetică de tip MMX înzestrată cu 6 operații posibile care vor fi discutate mai târziu.

2. Studiu bibliografic

2.1. Setul de regiștrii al tehnologiei MMX

MMX definește atât cei opt regiștri de procesor, denumiți MM0 până la MM7, cât și operațiile care se pot efectua cu aceștia. Fiecare registru are o capacitate de 64 de biți și poate fi folosit pentru a păstra fie numere întregi de 64 de biți, fie mai multe numere întregi mai mici într-un format „împachetat”: o instrucțiune poate fi apoi aplicată la două numere întregi de 32 de biți, patru numere întregi de 16 biți sau opt numere întregi de 8 biți simultan. MMX utilizează 64 biți mantisă din cei 80 de biți ai regiștrilor FPU prin tehnica de aliasare. Spre deosebire de registrele x87, care se comportă ca o stivă, pentru registrele MMX se folosește adresarea directă (random access).

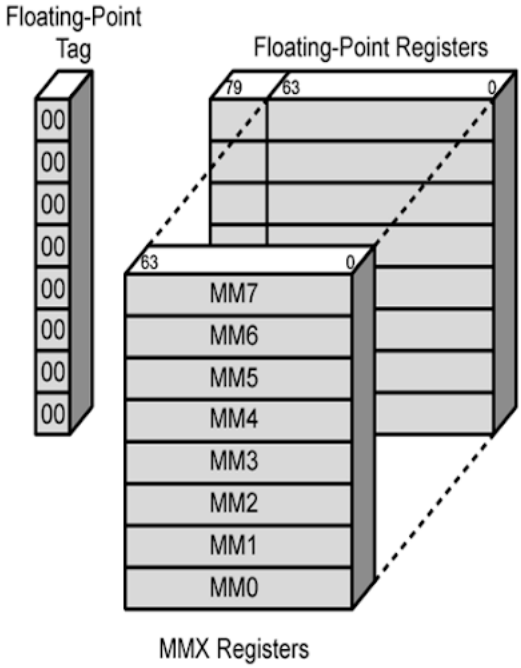

Fig.1. Setul de regiștri MMX

2.2. Tipuri de date

Arhitectura MMX introduce noi tipuri de date împachetate care pot fi grupate în regiștrii de 64 biți astfel:

- 8x8-bit packed bytes **B**
- 4x16-bit packed words **W**
- 2x32-bit packed doublewords **D**
- 1x64-bit quadword **Q**

Instrucțiunile MMX se aplică în paralel fiecărui bloc de date grupat în cadrul registrului de 64 biți.

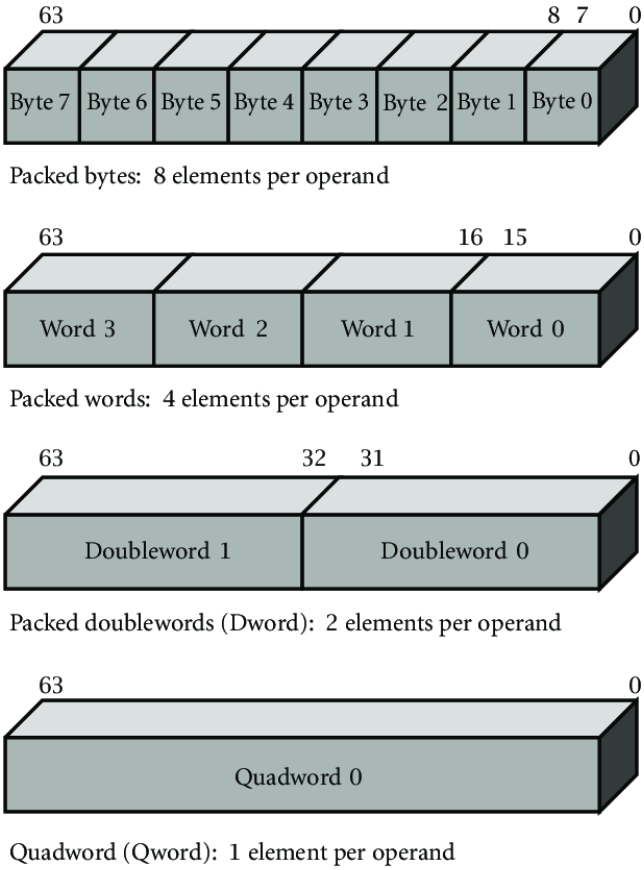

Fig.2. Tipuri de date introduse de arhitectura MMX

2.3. Instrucțiuni MMX (alese pentru proiect)

Majoritatea instrucțiunilor se efectuează pe doi operanzi, primul operand jucând rolul operandului sursă, iar cel de-al doilea fiind destinația. Cu toate acestea, există câteva instrucțiuni care utilizează trei operanzi, cel de-al treilea fiind un imediat (o valoarea constantă). Operandul destinație este aproape mereu un registru MMX, singura excepție fiind acele operații care salvează un registru MMX în memorie. Operandul sursă poate fi un registru MMX sau o locație de memorie. Instrucțiunile MMX accesează memoria folosind aceleași moduri de adresare ca instrucțiunile standard cu întregi, motiv pentru care orice mod de adresare 80x86 poate fi utilizat într-o instrucțiune MMX.

În cadrul acestui proiect am ales următoarele 6 instrucțiuni:

Aritmetice

PADD(B/W/D/Q) mm1, mm2/m64

Instrucțiunea PADD(B/W/D/Q) adună datele de tip B/W/D/Q utilizând adunare "wrap-around" (nesaturată). Orice transport al sumei este pierdut. Este responsabilitatea programatorului să se asigure că depășirea nu apare. Acestă

instrucțiune produce rezultat corect pentru operanzi de tipul signed și unsigned (presupunând că depășirea nu apare).

PSUB(B/W/D/Q) mm1, mm2/m64

Instrucțiunea PSUB(B/W/D) funcționează asemănător cu cea prezentată mai sus, făcând, în schimb scădere pe datele de tip B/W/D/Q. De asemenea, instrucțiunea poate funcționa fie pe numere întregi împachetate fără semn, fie pe numere cu semn (în complement față de 2); cu toate acestea, nu setează biții în registrul EFLAGS pentru a indica depășire și/sau transport. Pentru a preveni condițiile de depășire nedetectate, software-ul trebuie să controleze intervalele de valori pe care operează.

Logice

PXOR mm1, mm2/m64

Instrucțiunea efectuează o operație logică de SAU exclusiv pe biții operandului sursă (al doilea operand) și operandului destinație (primul operand) și stochează rezultatul în operandul destinație. Fiecare bit al rezultatului este 1 dacă biții corespunzători celor doi operanzi sunt diferiți; fiecare bit este 0 dacă biții corespunzători operanzilor sunt identici.

De shiftare

PSLL(B/W/D/Q) mm1, mm2/m64

Instrucțiunea mută biții din elementele de date individuale (cuvinte, cuvinte duble sau patru cuvinte) din operandul destinație (al doilea operand) la stânga cu numărul de biți specificat în operandul de numărare (primul operand). Pe măsură ce biții din elementele de date sunt deplasați la stânga, biții de ordine inferioară sunt șterși (setați la 0). Dacă valoarea specificată de operandul de numărare este mai mare decât 15 (pentru cuvinte), 31 (pentru cuvinte duble) sau 63 (pentru un cuvânt patru), atunci operandul destinație va avea toți biții setați la 0.

De comparare

PCMPGTB mm1, mm2/m64

Instrucțiunea PCMPGTB efectuează o comparație cu semn pentru valoarea mai mare a întregului octet, cuvânt sau cuvânt dublu împachetat în operandul destinație (al doilea operand) și operandul sursă (primul operand). Dacă un element de date din operandul de destinație este mai mare decât elementul de dată corespunzător din operandul sursă, elementul de date corespunzător din operandul de destinație va avea toți biții setați pe 1; în caz contrar, toți biții vor fi setați pe 0.

De conversie

PACKSSWB mm1, mm2/m64

Instrucțiunea PACKSSWB convertește numerele întregi de cuvinte împachetate cu semn din cei doi operanzi în numere întregi de octeți împachetate cu semn folosind saturația cu semn pentru a gestiona condițiile de depășire dincolo de intervalul de numere întregi de octeți cu semn. Dacă valoarea cuvântului cu semn este dincolo de intervalul unei valori de octet cu semn (adică, mai mare de 7FH sau mai mică de 80H), valoarea întregului octet cu semn saturat de 7FH sau, respectiv, 80H, este stocată în destinație.

3. Analiză

3.1. Propunere proiect

Proiectul are ca scop realizarea unei unități aritmetice de tip MMX pe care utilizatorul o poate folosi și efectua 6 dintre instrucțiunile MMX. Utilizatorul va putea alege una dintre operațiile: PADD(B/W/D/Q) , PSUB(B/W/D/Q), PSLL(B/W/D/Q), PXOR, PCMPGTB, PACKSSWB, cât și operanzii doriți, aceștia fiind deja hard-codați și stocați în memoria RAM. Atât tipurile de date cu care operează instrucțiunile, cât și efectuarea instrucțiunilor în sine reflectă modul unic în care arhitectura MMX utilizează paralelismul.

3.2. Plan de dezvoltare

Planul de dezvoltare al unității aritmetice de tip MMX se rezumă la implementarea unei memorii care să stocheze datele, a unei unități de control, a unei unități aritmetice capabilă să efectueze operațiile dorite, și, în final, a unui afișor pentru afișarea datelor de intrare, respectiv de ieșire. Pentru realizarea instrucțiunilor se vor implementa următoarele:

- Pentru operația de adunare, se vor implementa sumatoare pe opt biți și mux-uri a căror selecție va decide dacă operația se efectuează pe date de 8/16/32/64 biti.
- Pentru operația de scădere, se vor implementa scăzătoare pe opt biți și mux-uri a căror selecție va decide dacă operația se efectuează pe date de 8/16/32/64 biti.
- Pentru operația de XOR se va folosi drept componentă principală o poartă XOR cu două intrări pe 1 bit. Operația se va efectua pe date de dimensiune 64 biți, bit cu bit.
- Pentru operația de shiftare la stânga se va implementa un registru SIPO de shiftare la stânga pe 8 biți. Acesta se poate cascada pentru a obține rezultate pe 16,32, respectiv 64 biți.
- Pentru operația de comparație se va implementa un comparator pe 8 biți care va avea ca intrări 2 numere pe 8 biți și va avea 3 ieșiri corespunzătoare relației dintre

cele două numere: primul număr este mai mare decât cel de-al doilea, primul număr este mai mic decât cel de-al doilea sau numerele sunt egale. În cazul în care al doilea operand(destinația) este mai mare decât primul(sursa), biții acestuia vor fi setați pe '1' logic, altfel vor fi setați pe '0'.

- Operația de împachetare funcționează ca în figura [3]. Împachetarea se va face folosind saturația de semn pentru a gestiona condițiile de depășire a intervalului de numere întregi cu semn pe 8 biți, astfel încât nu există valori mai mici decât 80H sau mai mari decât 7FH.

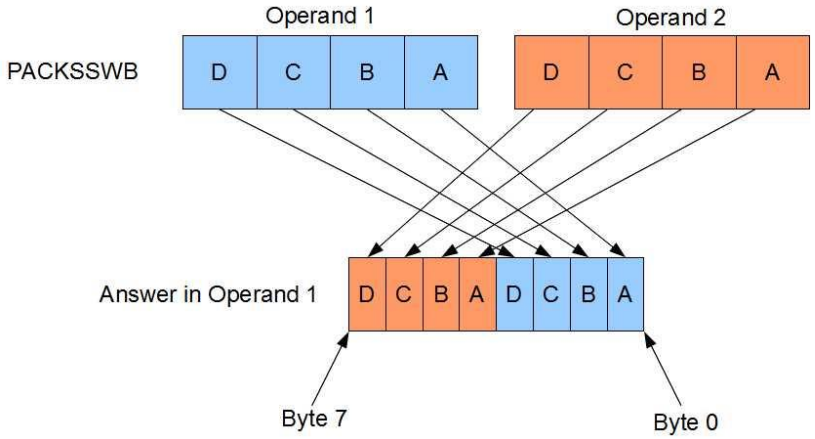

Fig.3. Instrucțiunea PACKSSWB

- Pentru stocarea datelor va fi implementată o memorie RAM care va putea stoca date de 64 biți. Aceasta va fi folosită atât pentru a citi operanzii sursă și destinație, cât și pentru a actualiza operandul destinație.
- Pentru afișarea datelor din memorie se va implementa afișorul cu 7 segmente și o unitate de control pentru decizii de afișare. Datele mai mari de 16 biți se vor afișa, pe rând, câte 16 biți odată prin apăsarea unui buton de pe plăcuță.

3.3. Descrierea funcționalității

Unitatea aritmetică de tip MMX va avea următoarele funcționalități:

1. Alegere și vizualizare operanzi (datele din memoria RAM)
2. Realizare diferite operații în funcție de combinația de switchuri aleasă
3. Vizualizare rezultat

Memoria RAM 8x64 va conține, în mod abstract, cei 8 regiștri MMX cu capacitatea de 64 biți, iar datele din regiștri vor fi hard-codate.

Ca prim pas în mecanismul de funcționare, se va realiza alegerea operanzilor de către utilizator, după cum urmează:

- “00000” —alegerea datelor din registrul mm1
- “00001” —alegerea datelor din registrul mm2
- “00010” —alegerea datelor din registrul mm3
- “00011” —alegerea datelor din registrul mm4
- “00100” —alegerea datelor din registrul mm5
- “00101” —alegerea datelor din registrul mm6
- “00110” —alegerea datelor din registrul mm7
- “00111” —alegerea datelor din registrul mm8

Al doilea pas constă în alegerea operației dorite, urmând următoarele codificări:

- “01000”—PADDB
- “01001”—PADDW
- “01010”—PADDD
- “01011”—PADDQ
- “01100”—PSUBB
- “01101”—PSUBW
- “01110”—PSUBD
- “01111”—PSUBQ
- “10000”—PXOR
- “10001”—PSLLB
- “10010”—PSLLW
- “10011”—PSLLD
- “10100” —PSLLQ
- “10101”—PCMPGTB
- “10110”—PACKSSWB

Pentru a înțelege pe deplin pașii parcurși de unitatea aritmetică MMX, organigrama de mai jos descrie funcționalitatea de bază a sistemului:

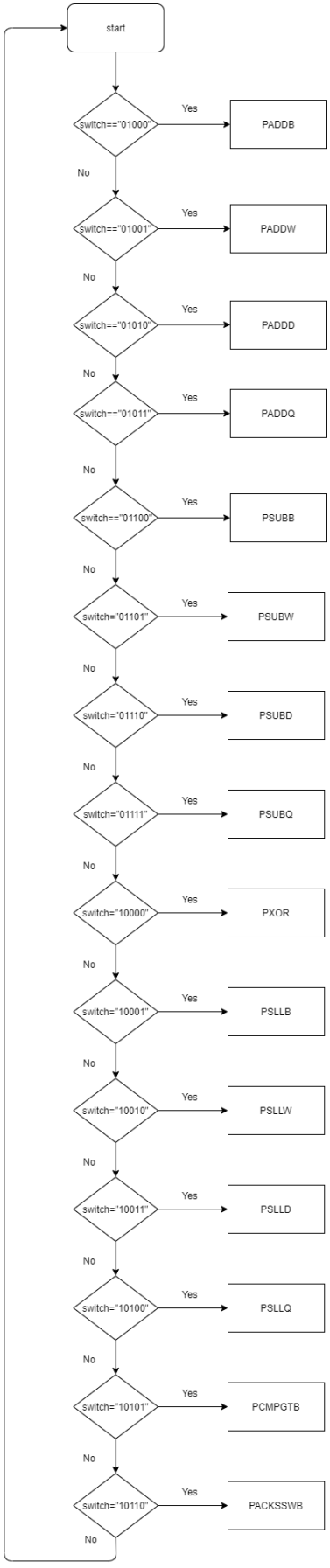

Fig.4.Organigrama care descrie operațiile

4. Design

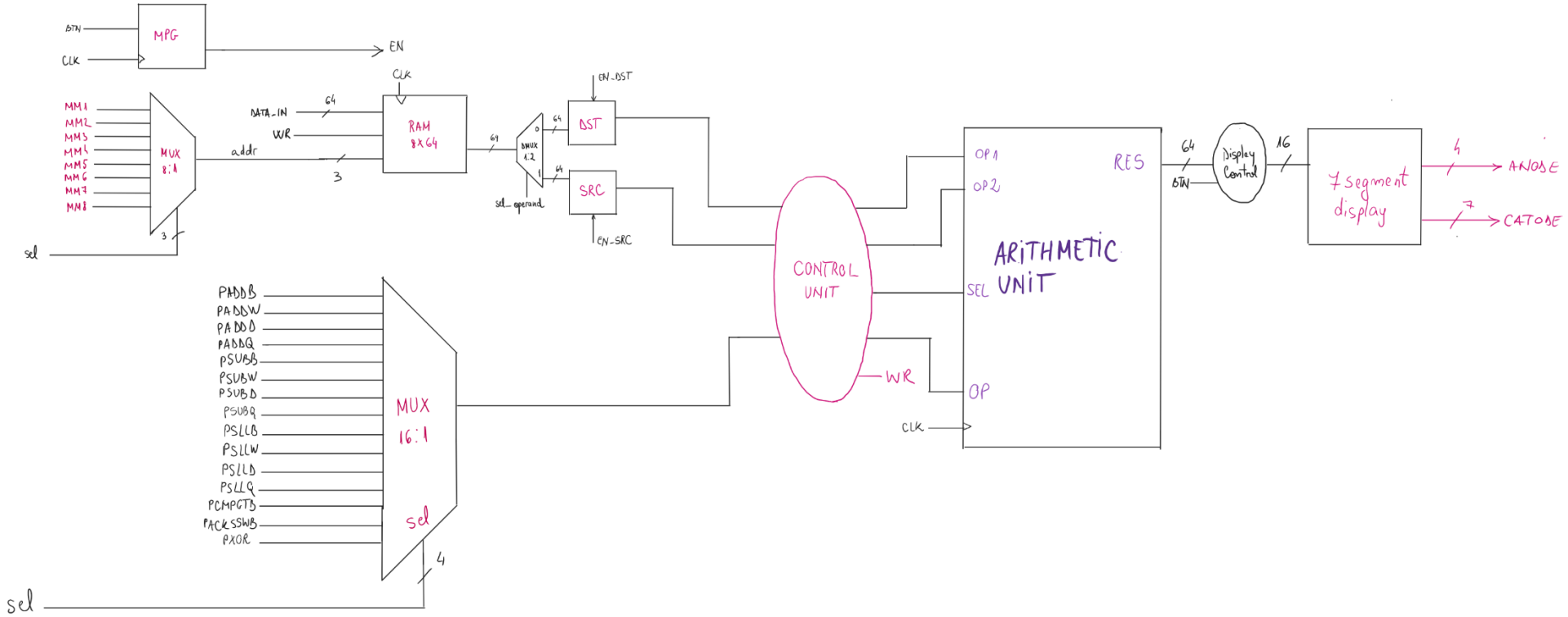

Fig.5. Schema de funcționare a unității aritmetice de tip MMX

Design-ul unității aritmetice de tip MMX este prezentat în figura [5], fiind alcătuit din câteva componente principale: un monopulse generator pentru activarea semnalelor de ENABLE o singură dată la apăsare unui buton, două multiplexoare pentru alegerea adresei operanzilor, respectiv alegerea operației, o memorie RAM care conține date hard-codate (operanzii), două registre în care sunt stocați operanzii sursă și destinație, o unitate de control care decide care sunt operanzii și operația, setând și semnalul de WR pentru scrierea rezultatului în RAM, o unitate aritmetică care este unitatea de bază, responsabilă cu efectuarea operațiilor (va fi discutată mai târziu), o unitate de control a afișării care decide care grup de 16 biți va fi afișat.

1. MonoPulse Generator (MPG) generează un semnal de enable pentru activarea/validarea frontului crescător al semnalului de ceas.

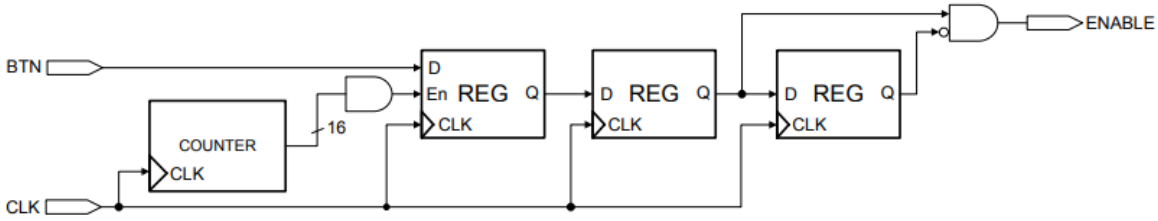

Fig.6. Generator de monoimpuls sincron

2. Multiplexorul 8:1 este folosit pentru generarea adresei la care se află datele MM1..MM8 în memorie. Selecția multiplexorului are ca input switchurile de la plăcuță, fiind controlată de utilizator.
3. Multiplexor 16:1 este folosit pentru alegerea operației dorite. De asemenea, selecția multiplexorului are ca input switchurile de la plăcuță, fiind controlată de utilizator.
4. Memoria RAM 8x64 conține cei opt regiștri MMX. De asemenea, în memorie se vor scrie rezultatele.
5. Control Unit preia datele de la multiplexoare și le interpretează. Principalul său rol este să preia operanzii din memorie (adresele lor) și să aleagă rezultatul conform operației alese (toate operațiile efectuându-se simultan). Totodată, la efectuarea unei instrucțiuni, vom avea nevoie să suprascriem registrul destinație cu rezultatul operației, de aceea semnalul WR va trebui să fie activ.
6. Display Control are rolul de a trimite la afișor câte 16 biți (maximul care poate fi afișat pe 7 segment display), afișajul fiind controlat de apăsarea unui buton.

7. Afișorul cu 7 segmente permite afișarea datelor pe plăcuță astfel încât să fie vizibile pentru utilizator. Circuitul SSD este descris în figura [7].

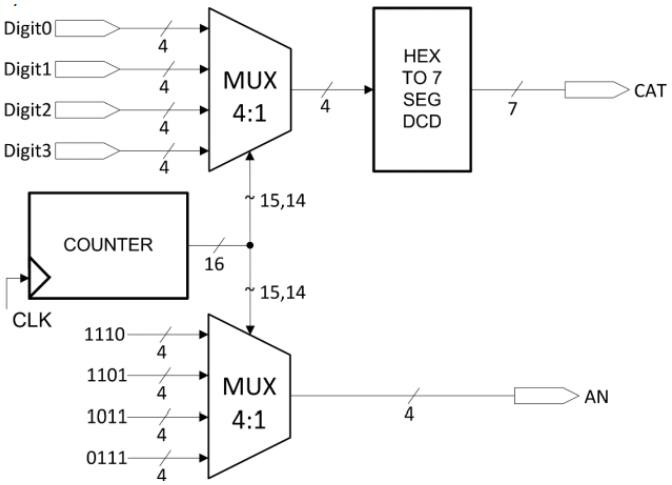

Fig.7. Schema circuitului de afișare SSD

8. Unitatea aritmetică este cea mai complexă parte care include circuite capabile să efectueze operațiile dorite. Aceasta are ca intrări operanzii, operația dorită, clock-ul plăcuței și o selecție prin care se alege pe câți biți se efectuează operația. Rezultatul obținut poate fi mai apoi afișat și salvat în memorie, registrul destinație fiind suprascris.

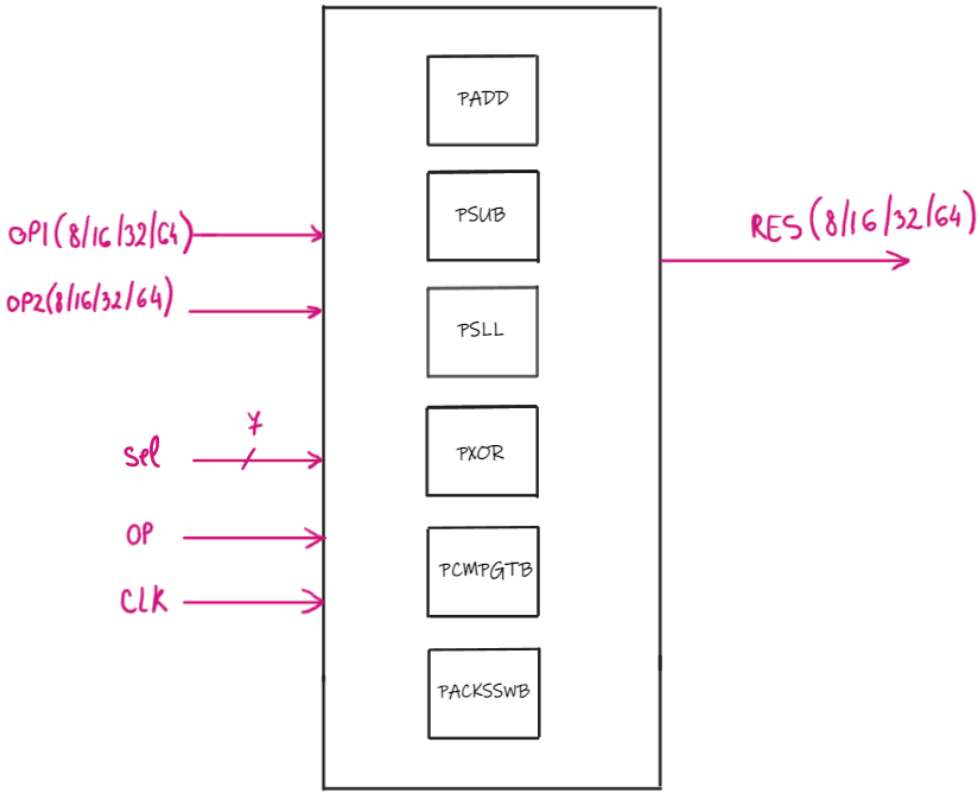

Fig.8. Top level design pentru unitatea aritmetică

5. Implementare

Implementarea unității aritmetice de tip MMX constă în principal în implementarea componentelor corespunzătoare pentru efectuarea diferitelor operații și a unei unități de control pentru realizarea deciziilor.

Astfel, pentru operația PADD am implementat sumatoare pe 8 biți cascadând sumatoare de 1 bit cu ajutorul semnalului de cout. Semnalul de cin este '0' logic.

```
entity full_adder is
Port ( x,y:in std_logic;
       cin:in std_logic;
       s,cout:out std_logic);
end full_adder;

architecture Behavioral of full_adder is
begin
     s<=x xor y xor cin;
     cout<=(x and y) or (x and cin) or (y and cin);
end Behavioral;
```

Fig.9. Sumator pe 1 bit

Pentru cascadarea sumatoarelor este necesar ca semnalul de carry out al unui sumator să fie legat la semnalul de carry in al următorului sumator.

```
signal carry : std_logic_vector(6 downto 0);
begin
     a0:full_adder port map (x=>x(0),y=>y(0),cin=>cin,s=>s(0),cout=>carry(0));
     a1:full_adder port map (x=>x(1),y=>y(1),cin=>carry(0),s=>s(1),cout=>carry(1));
     a2:full_adder port map (x=>x(2),y=>y(2),cin=>carry(1),s=>s(2),cout=>carry(2));
     a3:full_adder port map (x=>x(3),y=>y(3),cin=>carry(2),s=>s(3),cout=>carry(3));
     a4:full_adder port map (x=>x(4),y=>y(4),cin=>carry(3),s=>s(4),cout=>carry(4));
     a5:full_adder port map (x=>x(5),y=>y(5),cin=>carry(4),s=>s(5),cout=>carry(5));
     a6:full_adder port map (x=>x(6),y=>y(6),cin=>carry(5),s=>s(6),cout=>carry(6));
     a7:full_adder port map (x=>x(7),y=>y(7),cin=>carry(6),s=>s(7),cout=>cout);


end Behavioral;
```

Fig.10. Cascadare sumatoare pe 1 bit

Pentru a putea realiza operații în paralel pe diferite tipuri de date B/W/D/Q se vor folosi multiplexoare între perechi de sumatoare pe 8 biți care trimit pe intrarea sumatoarelor fie '0' logic, fie carry out al sumatorului anterior. În cazul în care operația este pe date de tip byte sumatoarele pe 8 biți vor funcționa independent, astfel selecțiile multiplexoarelor vor da mai departe pe carry in '0' logic. Pentru tipul de date word sumatoarele for funcționa grupate câte două, selecția fiind "1010101", pentru tipul de date double word sumatoarele vor funcționa grupate câte 4

(“1110111” pe selecție), iar pentru quad words sumatoarele vor funcționa toate cele 8 împreună (“1111111” pe selecție).

```
add8bit0:adder_8bit port map(x=>x(7 downto 0),y=>y(7 downto 0),cin=>'0',s=>res(7 downto 0),cout=>cout(0));
mux0:mux2_1 port map(i1=>'0',i2=>cout(0),s=>sel(0),i3=>mux_out(0));
add8bit1:adder_8bit port map(x=>x(15 downto 8),y=>y(15 downto 8),cin=>mux_out(0),s=>res(15 downto 8),cout=>cout(1));
mux1:mux2_1 port map(i1=>'0',i2=>cout(1),s=>sel(1),i3=>mux_out(1));
add8bit2:adder_8bit port map(x=>x(23 downto 16),y=>y(23 downto 16),cin=>mux_out(1),s=>res(23 downto 16),cout=>cout(2));
mux2:mux2_1 port map(i1=>'0',i2=>cout(2),s=>sel(2),i3=>mux_out(2));
add8bit3:adder_8bit port map(x=>x(31 downto 24),y=>y(31 downto 24),cin=>mux_out(2),s=>res(31 downto 24),cout=>cout(3));
mux3:mux2_1 port map(i1=>'0',i2=>cout(3),s=>sel(3),i3=>mux_out(3));
add8bit4:adder_8bit port map(x=>x(39 downto 32),y=>y(39 downto 32),cin=>mux_out(3),s=>res(39 downto 32),cout=>cout(4));
mux4:mux2_1 port map(i1=>'0',i2=>cout(4),s=>sel(4),i3=>mux_out(4));
add8bit5:adder_8bit port map(x=>x(47 downto 40),y=>y(47 downto 40),cin=>mux_out(4),s=>res(47 downto 40),cout=>cout(5));
mux5:mux2_1 port map(i1=>'0',i2=>cout(5),s=>sel(5),i3=>mux_out(5));
add8bit6:adder_8bit port map(x=>x(55 downto 48),y=>y(55 downto 48),cin=>mux_out(5),s=>res(55 downto 48),cout=>cout(6));
mux6:mux2_1 port map(i1=>'0',i2=>cout(6),s=>sel(6),i3=>mux_out(6));
add8bit7:adder_8bit port map(x=>x(63 downto 56),y=>y(63 downto 56),cin=>mux_out(6),s=>res(63 downto 56),cout=>cout(7));
```

Fig.11. Descrierea structurală a sumatorului pe 64 biți

Operația PSUB se realizează similar, diferența făcând-o componenta de bază și anume scăzătorul pe 1 bit.

```
entity full_subtractor is
Port ( A,B,C : in std_logic;
diff, borrow : out std_logic);
end full_subtractor;

architecture Behavioral of full_subtractor is
begin
diff <= (A xor B) xor C;
borrow <= ((not A) and (B or C)) or (B and C);
end Behavioral;
```

Fig.12. Scăzător pe 1 bit

De asemenea, pentru realizarea operației pe diferite tipuri de date se folosește același mecanism de grupare a scăzătoarelor pe 8 biți, alegându-se dacă se trimite mai departe transportul sau ‘0’ logic.

```
subtractor0:subtractor_8bit port map(a=>x(7 downto 0),b=>y(7 downto 0),c=>'0',diff=>res(7 downto 0),borrow=>br(0));
mux0:mux2_1 port map(i1=>'0',i2=>br(0),s=>sel(0),i3=>mux_out(0));
subtractor1:subtractor_8bit port map(a=>x(15 downto 8),b=>y(15 downto 8),c=>mux_out(0),diff=>res(15 downto 8),borrow=>br(1));
mux1:mux2_1 port map(i1=>'0',i2=>br(1),s=>sel(1),i3=>mux_out(1));
subtractor2:subtractor_8bit port map(a=>x(23 downto 16),b=>y(23 downto 16),c=>mux_out(1),diff=>res(23 downto 16),borrow=>br(2));
mux2:mux2_1 port map(i1=>'0',i2=>br(2),s=>sel(2),i3=>mux_out(2));
subtractor3:subtractor_8bit port map(a=>x(31 downto 24),b=>y(31 downto 24),c=>mux_out(2),diff=>res(31 downto 24),borrow=>br(3));
mux3:mux2_1 port map(i1=>'0',i2=>br(3),s=>sel(3),i3=>mux_out(3));
subtractor4:subtractor_8bit port map(a=>x(39 downto 32),b=>y(39 downto 32),c=>mux_out(3),diff=>res(39 downto 32),borrow=>br(4));
mux4:mux2_1 port map(i1=>'0',i2=>br(4),s=>sel(4),i3=>mux_out(4));
subtractor5:subtractor_8bit port map(a=>x(47 downto 40),b=>y(47 downto 40),c=>mux_out(4),diff=>res(47 downto 40),borrow=>br(5));
mux5:mux2_1 port map(i1=>'0',i2=>br(5),s=>sel(5),i3=>mux_out(5));
subtractor6:subtractor_8bit port map(a=>x(55 downto 48),b=>y(55 downto 48),c=>mux_out(5),diff=>res(55 downto 48),borrow=>br(6));
mux6:mux2_1 port map(i1=>'0',i2=>br(6),s=>sel(6),i3=>mux_out(6));
subtractor7:subtractor_8bit port map(a=>x(63 downto 56),b=>y(63 downto 56),c=>mux_out(6),diff=>res(63 downto 56),borrow=>br(7));
```

Fig.13.Descrierea structurală a scăzătorului pe 64 biți

Operația PXOR folosește o poartă XOR pe 1 bit, iar pentru realizarea operației pe 64 biți se vor folosi 64 de astfel de porți.

```
entity PXOR is
Port ( x,y:in std_logic_vector(63 downto 0);
       res:out std_logic_vector(63 downto 0));
end PXOR;

architecture Behavioral of PXOR is
component xor_gate is
Port ( x,y:in std_logic;
       z:out std_logic);
end component;
begin
mapping:for i in 0 to 63 generate
xor_instantiate:xor_gate port map(x=>x(i),y=>y(i),z=>res(i));
end generate;
end Behavioral;
```

Fig.14.Descrierea structurală a operației XOR pe 64 biți

Operația PSLL folosește registre de deplasare la stânga SIPO pe 8 biți și multiplexoare pentru efectuarea operației pe date de tipul B/W/D/Q. Implementarea registrelor pe 8 biți s-a realizat cu ajutorul a opt bistabile D Flip Flop cascadate.

```
architecture behave of d_ff is
begin
process(clk,reset,en,din)
begin
  if (reset='1') then
    dout <= '0';
  elsif clk'event and clk='1' then
    if en='1' then
      dout <= din;
    end if;
  end if ;
end process ;
end behave;
```

Fig.14.Descrierea comportamentală a bistabilului D flip flop

```
architecture Behavioral of shiftleft_8bit is
component d_ff is
port(
clk : in STD_LOGIC;
din : in STD_LOGIC;
reset : in STD_LOGIC;
en : in STD_LOGIC;
dout : out STD_LOGIC
);
end component;
signal s:std_logic_vector(7 downto 0);
begin
dout<=s;
u0 : d_ff port map (clk => clk, din => sin, reset => reset,en=>en, dout => s(0));
u1 : d_ff port map (clk => clk, din => s(0), reset => reset,en=>en, dout => s(1));
u2 : d_ff port map (clk => clk, din => s(1), reset => reset,en=>en, dout => s(2));
u3 : d_ff port map (clk => clk, din => s(2), reset => reset,en=>en, dout => s(3));
u4 : d_ff port map (clk => clk, din => s(3), reset => reset,en=>en, dout => s(4));
u5 : d_ff port map (clk => clk, din => s(4), reset => reset,en=>en, dout => s(5));
u6 : d_ff port map (clk => clk, din => s(5), reset => reset,en=>en, dout => s(6));
u7 : d_ff port map (clk => clk, din => s(6), reset => reset,en=>en, dout => s(7));
```

Fig.15.Descrierea structurală a registrului de deplasare la stânga pe 8 biți

Deoarece operația de shiftare folosește regiștrii de tip SIPO, descrierea componentei va fi de tip mixt, înglobând atât o parte comportamentală, cât și una strict structurală. Astfel, pentru a descrie comportamentul componentei se va folosi un proces în care se vor trimite pe intrarea serială atât biții numărului care trebuie shiftat, cât și biții de '0' care arată cu cât este shiftat numărul. Pentru a trimite numărul bit cu bit am folosit un semnal i care contorizează fiecare cifră, iar în momentul în care acesta ajunge la 64 adică numărul a fost trimis în întregime, este momentul ca acesta să fie shiftat cu x poziții (x fiind operandul sursă). Pentru a contoriza shiftarea am folosit un semnal j, iar atâta timp cât j nu a atins valoarea lui x se va trimite pe intrarea serială '0' logic.
În momentul în care semnalele i și j ating valorile maxime, semnalul de enable pentru toate registrele devine '0', pentru a opri shiftarea. De asemenea, componenta este controlată și printr-un semnal de enable care este '1' atunci când utilizatorul alege operația de shiftare, '0' în caz contrar.

```
process(clk)
begin
  if clk'event and clk='1' then
   if en='1' then
     if i<64 then
       sin<=y(i);
       i<=i+1;
     elsif j<to_integer(unsigned(x)) then
       sin<='0';
       j<=j+1;
     else
       enable<="00000000";
     end if;
     else
       i<=0;
       j<=0;
       enable<="11111111";
    end if;
  end if;
end process;
```

Fig.16. Proces care descrie trimiterea datelor pe intrarea serială a registrelor

La fel ca la operația de adunare și scădere, operația de shiftare folosește atât registrele pe 8 biți cât și multiplexoare pentru a alege în ce format se efectuează operația. Ieșirea multiplexoarelor va fi fie '0', fie cel mai semnificativ bit al rezultatului shiftării anterioare, această ieșire fiind ulterior folosită ca intrare serială la următorul shifter.

```
sll0:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(0),sin=>sin,dout=>res(7 downto 0));
mux0:mux2_1 port map(i1=>'0',i2=>res(7),s=>sell(0),i3=>mux_out(0));
sll1:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(1),sin=>mux_out(0),dout=>res(15 downto 8));
mux1:mux2_1 port map(i1=>'0',i2=>res(15),s=>sell(1),i3=>mux_out(1));
sll2:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(2),sin=>mux_out(1),dout=>res(23 downto 16));
mux2:mux2_1 port map(i1=>'0',i2=>res(23),s=>sell(2),i3=>mux_out(2));
sll3:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(3),sin=>mux_out(2),dout=>res(31 downto 24));
mux3:mux2_1 port map(i1=>'0',i2=>res(31),s=>sell(3),i3=>mux_out(3));
sll4:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(4),sin=>mux_out(3),dout=>res(39 downto 32));
mux4:mux2_1 port map(i1=>'0',i2=>res(39),s=>sell(4),i3=>mux_out(4));
sll5:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(5),sin=>mux_out(4),dout=>res(47 downto 40));
mux5:mux2_1 port map(i1=>'0',i2=>res(47),s=>sell(5),i3=>mux_out(5));
sll6:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(6),sin=>mux_out(5),dout=>res(55 downto 48));
mux6:mux2_1 port map(i1=>'0',i2=>res(55),s=>sell(6),i3=>mux_out(6));
sll7:shiftleft_8bit port map(clk=>clk,reset=>reset,en=>enable(7),sin=>mux_out(6),dout=>res(63 downto 56));
```

Fig.17. Descrierea structurală a operației PSLL

Operația PCMPGTB folosește comparatoare pe 8 biți, care sunt realizate din comparatoare pe 1 bit cascadate. Dintre cele trei semnale greater, smaller și equal ne interesează semnalul greater care este setat dacă operandul destinație este mai mare decât operandul sursă, adică al doilea operand este mai mare decât primul. Marea diferență a comparatorului implementat în acest caz este dată de ordinea operanzilor (primul operand este sursa și al doilea destinația) care este fix invers față de un comparator obișnuit.

```
entity comparator_1bit is
Port (x,y,gin,lin:in std_logic;
      eout,lout,gout:out std_logic );
end comparator_1bit;

--y>x greater

architecture Behavioral of comparator_1bit is
begin
lout<=(x and not y) or (x and gin) or (not y and gin);
eout<=(not x and not y and not gin and not lin) or (x and y and not gin and not lin);
gout<=(not x and y) or (not x and lin) or (y and lin);
end Behavioral;
```

Fig.18. Comparator pe 1 bit

Operația se efectuează pe octeți, astfel încât se va compara câte un octet de la operandul sursă cu câte un octet de la operandul destinație pentru obținerea rezultatului pe 64 biți. Dacă un octet din operandul destinație este mai mare decât octetul corespunzător operandului sursă, atunci în operandul destinație se va scrie x“FF”, altfel se va scrie x”00”.

Operația PACKSSWB convertește o secvență mai largă de valori într-o secvență mai restrânsă de valori prin saturație de semn, mai precis, cuvintele în octeți. Astfel că pentru a acoperi întreg numărul se vor folosi 8 componente care transformă word în byte. Convenția este relativ simplă: valorile pozitive nu pot depăși x”7F”, iar valorile negative nu pot depăși x”80”. În caz de depășire se va scrie în operandul destinație valoarea limită pentru numere pozitive, respectiv pentru cele negative. Pentru

alegerea operandului care va fi încadrat în valorile limită se va folosi un proces, iar pentru a realiza saturația de semn se va folosi, de asemenea, tot un proces. Pentru primii 4 octeți se va folosi operandul destinație, iar pentru ultimii patru operandul sursă.

```
process(sel,x,y)
begin
case sel is
  when '0'=>s<=x;
  when '1'=>s<=y;
  when others=>s<="XXXXXXXXXXXXXXXX";
end case;
end process;
process(s)
begin
if (signed(s(15 downto 0))>=signed(max_range_positive) and s(15)='0') then
res(7 downto 0)<=max_range_positive;
elsif (signed(s(15 downto 0))<signed(max_range_positive) and s(15)='0') then
res(7 downto 0)<=s(7 downto 0);
end if;
if (signed(s(15 downto 0))>=signed(max_range_negative) and s(15)='1') then
res(7 downto 0)<=max_range_negative;
elsif (signed(s(15 downto 0))<signed(max_range_negative) and s(15)='1') then
res(7 downto 0)<=s(7 downto 0);
end if;
end process;
```

Fig.19. Conversia unui word în byte

6. Testare și validare

Exemple folosite pentru a testa operația PADDQ:

- X=x"FFFFFFFFFFFFFFFF"
- Y= x" FFFFFFFFFFFFFFFF"
- Sel=x"7F"
- RES=x"FFFFFFFFFFFFFFFE"

Exemple folosite pentru a testa operația PADDD:

- X=x"FFFFFFFFFFFFFFFF"
- Y= x" FFFFFFFFFFFFFFFF"
- Sel=x"77"
- RES=x"FFFFFFFEFFFFFFFE"

Exemple folosite pentru a testa operația PADDW:

- X=x"FFFFFFFFFFFFFFFF"
- Y= x" FFFFFFFFFFFFFFFF"
- Sel=x"55"
- RES=x"FFFEFFFEFFFEFFFE"

Exemple folosite pentru a testa operația PADDB:

- X=x"FFFFFFFFFFFFFFFF"
- Y= x" FFFFFFFFFFFFFFFF"
- Sel=x"00"

- RES=x"FEFEFEFEFEFEFEFE"

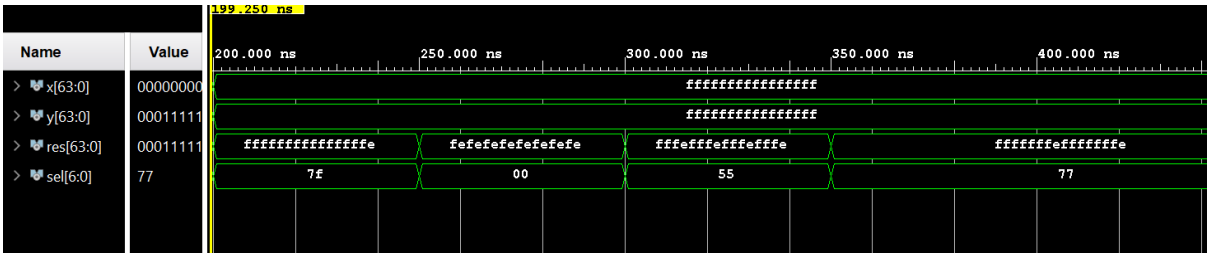

Fig.20. Testbench pentru operația PADD

Exemple folosite pentru a testa operația PSUBQ:

- X=x"2222222222222222"
- Y= x"1111111111111111"
- Sel=x"7F"
- RES=x"EEEEEEEEEEEEEEEF"

Exemple folosite pentru a testa operația PSUBD:

- X=x"2222222222222222"
- Y= x"1111111111111111"
- Sel=x"77"
- RES=x"EEEEEEEFEEEEEEEF"

Exemple folosite pentru a testa operația PSUBW:

- X=x"2222222222222222"
- Y= x"1111111111111111"
- Sel=x"55"
- RES=x"EEEFEEEFEEEFEEEF"

Exemple folosite pentru a testa operația PSUBB:

- X=x"2222222222222222"
- Y= x"1111111111111111"
- Sel=x"00"
- RES=x"EFEFEFEFEFEFEFEF"

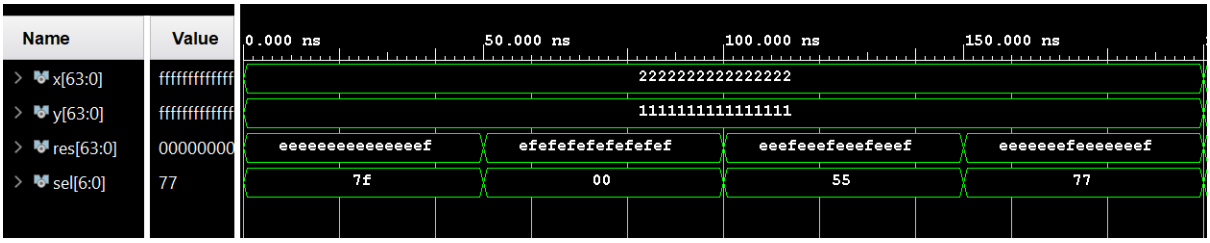

Fig.21. Testbench pentru operația PSUB

Exemple folosite pentru a testa operația PXOR:

- X=x"FFFF1111FFF88900"
- Y= x"FFFFFFFFFFFFFFFF"
- RES=x"0000EEEE000776FF"

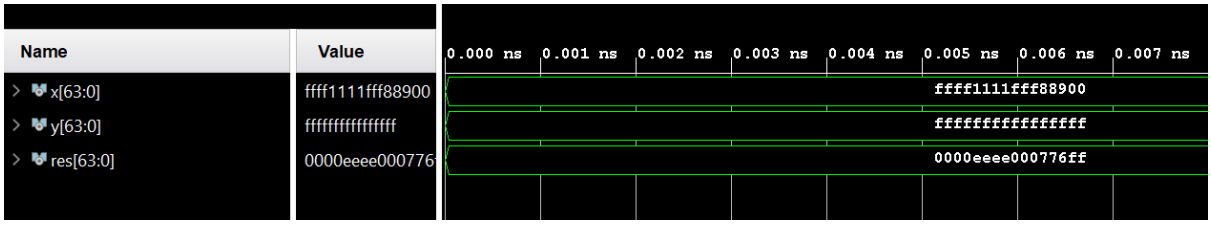

Fig.22. Testbench pentru operația PXOR

Exemple folosite pentru a testa operația PCMPGTB:

- X=x"1111111111111111"
- Y= x"0000001111111111"
- RES=x"0000000000000000"

Exemple folosite pentru a testa operația PCMPGTB:

- X=x"1111111111111111"
- Y= x"1111111111111111"
- RES=x"0000000000000000"

Exemple folosite pentru a testa operația PCMPGTB:

- X=x"0011111111111111"
- Y= x"1111111111111111"
- RES=x"0000000000000000"

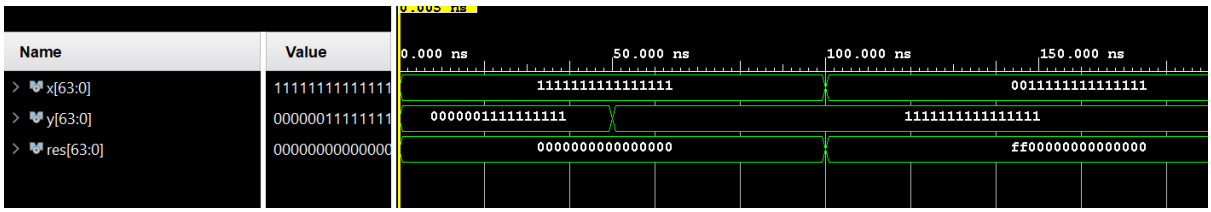

Fig.23. Testbench pentru operația PCMPGTB

Exemple folosite pentru a testa operația PSLLD:

- X=x"0000000000000007"
- Y= x"FFFFFFFFFFFFFFFF"
- Sel=x"77"
- RES=x"FFFFFFF80FFFFFF80"

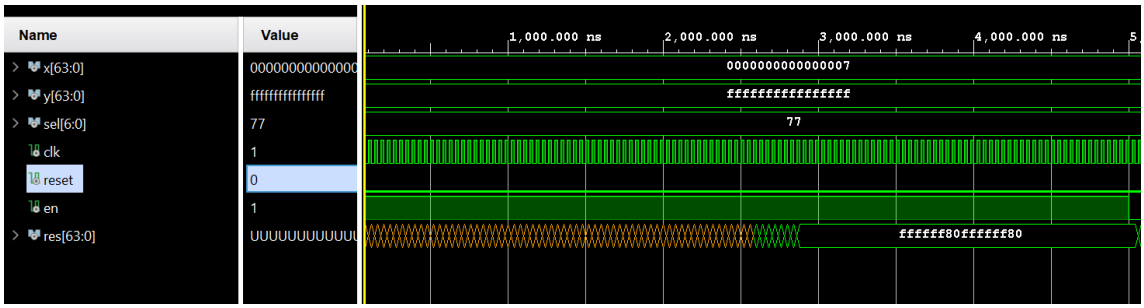

Fig.24. Testbench pentru operația PSLLD

Exemple folosite pentru a testa operația PSLLQ:

- X=x"0000000000000007"
- Y= x"FFFFFFFFFFFFFFFF"

- Sel=x”7F”
- RES=x”FFFFFFFFFFFFFF80”

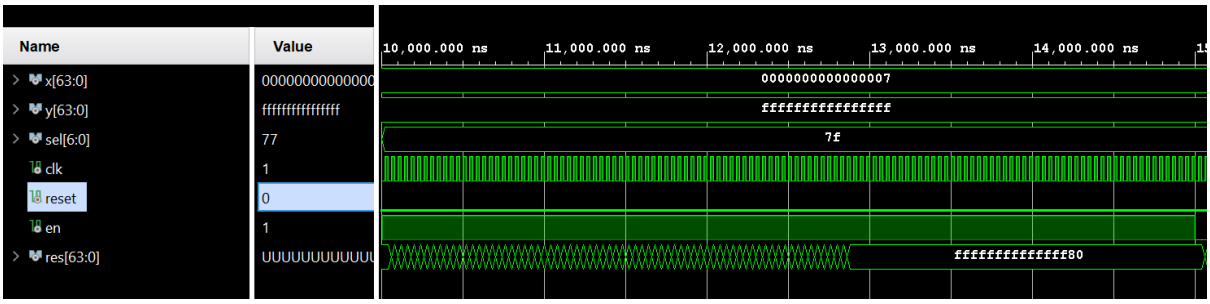

Fig.24. Testbench pentru operația PSLLQ

Exemple folosite pentru a testa operația PSLLW:

- X=x”0000000000000007”
- Y= x”FFFFFFFFFFFFFFFF”
- Sel=x”55”
- RES=x”FF80FF80FF80FF80”

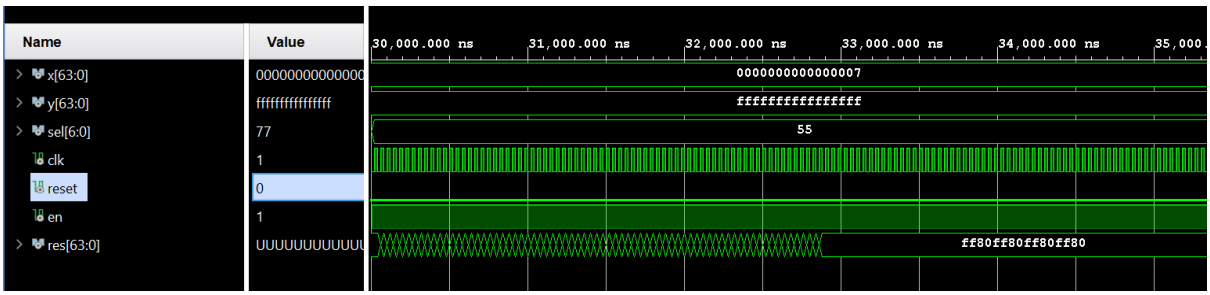

Fig.24. Testbench pentru operația PSLLW

Exemple folosite pentru a testa operația PSLLB:

- X=x”0000000000000007”
- Y= x”FFFFFFFFFFFFFFFF”
- Sel=x”00”
- RES=x”8080808080808080”

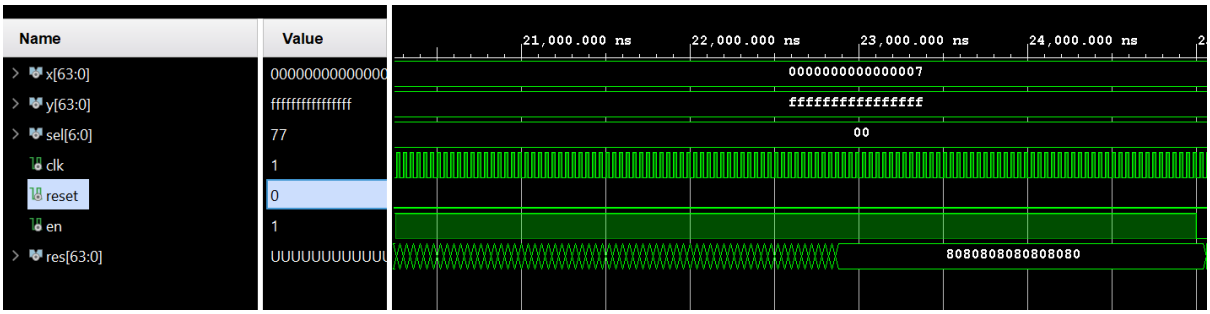

Fig.24. Testbench pentru operația PSLLB

Exemple folosite pentru a testa operația PACKSSWB:

- X=x”FFFFFFFFFFFFFFFF”
- Y= x”0000000000000000”
- RES=x”8080808000000000”

Exemple folosite pentru a testa operația PACKSSWB:

- X=x"FFFFFFFFFFFFFFFF"
- Y= x"0888888888888888"
- RES=x"80808087F888888"

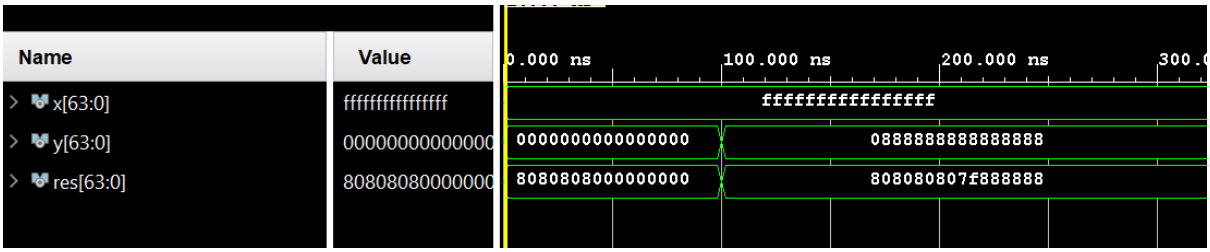

Fig.24. Testbench pentru operația PACKSSWB

6. Concluzii

În cadrul acestui proiect am reușit înțeleg modul de funcționare al arhitecturii MMX, a instrucțiunilor și tipurilor de date nou introduse, a modului de stocare a acestora și a modului în care este exploatat și introdus paralelismul.
Ceea ce mi-a plăcut în mod special a fost implementarea operațiilor și faptul că am avut libertatea de a proiecta unitatea aritmetică după bunul plac.

8.Bibliografie

[1] Oracle, "x86 Assembly Language Reference Manual" [Online]. Disponibil: https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html

[2] Randall Hyde, "The Art of Assembly Language 32-bit Edition" [Online]. Disponibil: www.lamed-oti.com/school/oe/assembly1/The Art of Assembly Language 32-bit Edition.pdf

[3] Felix Cloutier, "x86 and amd64 instruction reference"[Online]. Disponibil: https://www.felixcloutier.com/x86/index.html

[4] Randall Hyde, "IA-32 Intel® Architecture Software Developer's Manual"[Online]. Disponibil: https://www.csie.ntu.edu.tw/~cyy/courses/assembly/docs/ch11_MMX.pdf