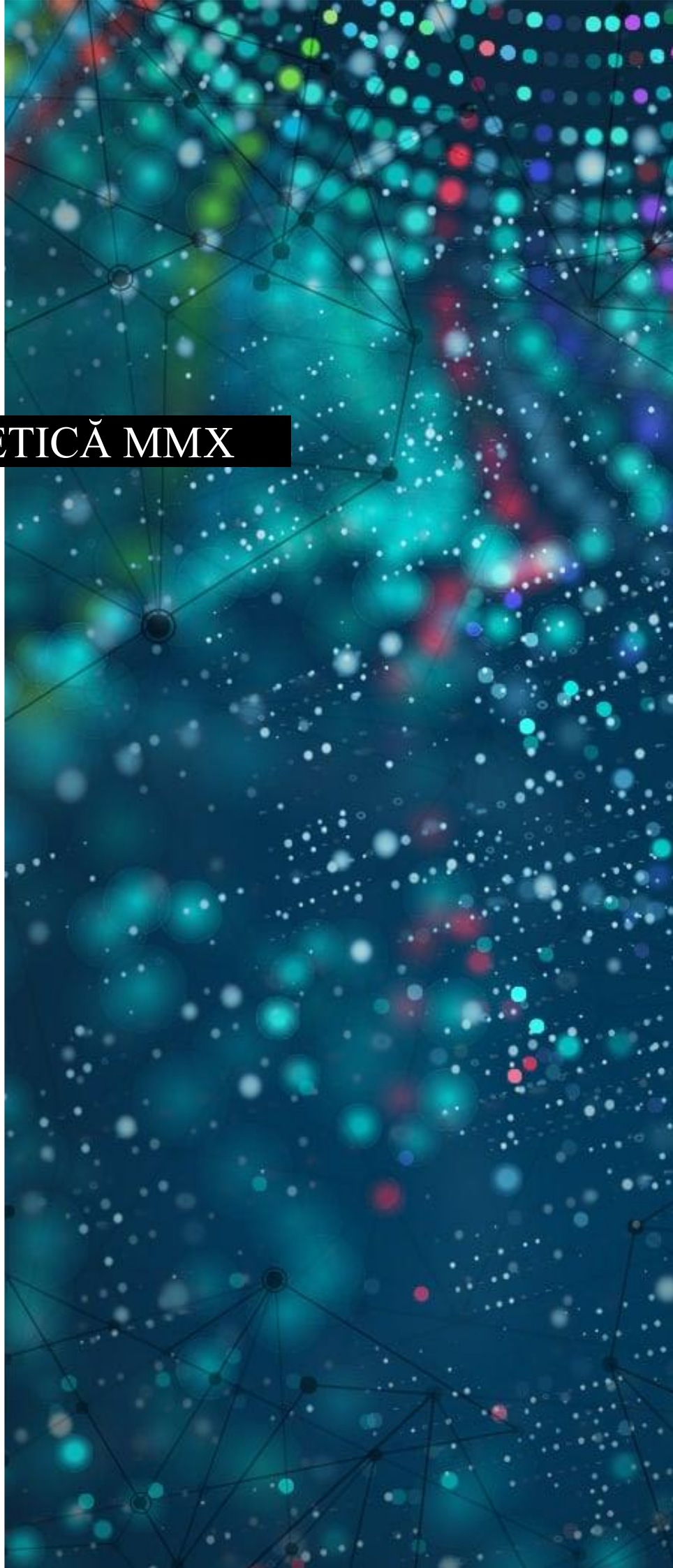
UNITATE ARITMETICĂ MMX



Remeș Daria-Maria

Universitatea Thenică din Cluj-Napoca

Ianuarie 2024

Unitatea aritmetică de tip MMX

Cuprins

1.Introducere

1.1 Context

Scopul acestui proiect este acela de a proiecta, implementa și testa o unitate aritmetică de tip MMX (MultiMedia eXtension). Vor fi implementate șase operații aritmetice din setul de instrucțiuni MMX al arhitecturii x86, divizie a categoriei SIMD (Single Instruction Multiple Data), utilizat pentru prelucrarea datelor multimedia, cum sunt imaginile și înregistrările video.

Componenta rezultată poate fi utilizată ca și un calculator, ce efectuează operații diverse cum sunt: adunarea, scăderea, înmulțirea, compararea a două numere, ce utilizează date pe 64 de biți. Ulterior poate fi integrată în cadrul altor sisteme mult mai complexe de prelucrare grafică.

1.2 Specificații

Sistemul va fi descris în limbajul VHDL, simulat în IDE-ul pus la dispoziție de Vivado, iar mai apoi simulat. Această unitate trebuie să fie eficientă în ceea ce privește resursele utilizate, să asigure o performanță optimă și să poată fi integrată cu ușurință în diverse aplicații multimedia.

1.3 Obiective

Obiectivul principal îl reprezintă dezvoltarea unei unități aritmetice MMX corect funcțională din punct de vedere logic, care să respecte setul de instrucțiuni și să ofere rezultate precise pentru operațiile aritmetice. Introducerea instrucțiunii de către utilizator, prelucrarea acesteia prin extragerea operanzilor, selectarea operației de efectuat și afișarea rezultatului corespunzător.

Realizarea unui proces complet de testare și validare pentru a asigura corectitudinea și fiabilitatea operațiilor aritmetice pentru diverse scenarii.

Întocmirea documentației clar și precis pentru a facilita înțelegerea și utilizarea unității aritmetice.

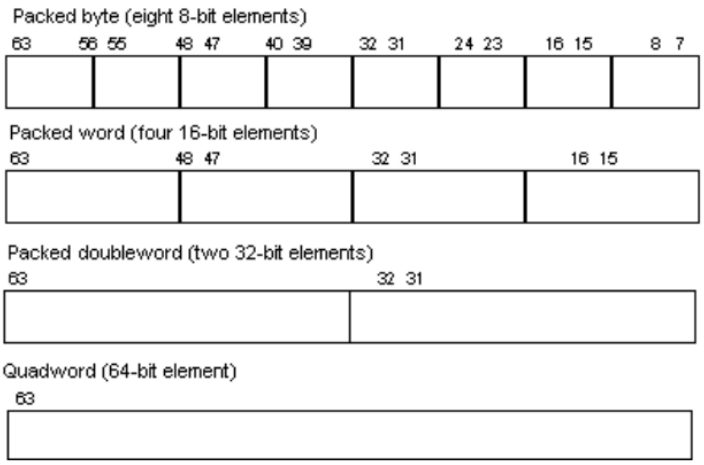
2. Studiu bibliografic

Principalul tip de date utilizat în setul de instrucțiuni de tip MMX este cel împachetat. Mai multe numere întregi sunt grupate într-o structură de dimensiunea a 64 de biți. Acești 64 de biți sunt apoi stocați în cadrul regiștrilor MMX de dimensiune identică. Tipurile de date suportate pentru reprezentare sunt sign și unsigned fixed-point integers, bytes, word, doublewords și quadwors.

Fiecare tip de date cu care pot opera instrucțiunile MMX are următoarea reprezentare:

- Packed byte – 8 bytes împachetați într-o structură de 64 de biți.
- Packed word – 4 words (16 biți) împachetate într-o structură de 64 de biți.
- Packed doubleword – 2 words (32 de biți) împachetate într-o strcutură de 64 de biți.

- Quadword – un element de 64 de biți.



În momentul în care o instrucțiune MMX se va executa, operația va fi aplicată pe valorile selectate, stocate în cadrul regiștrilor, iar rezultatul va fi stocat într-un alt registru MMX.

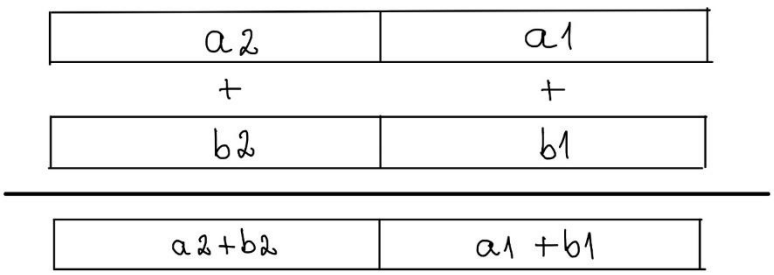
Selecția operației are loc în baza unui unui opcode unic, asociat fiecărei operații în parte.

3. Analiză

În această secțiune vor fi prezentate pe scurt instrucțiunile implementate.

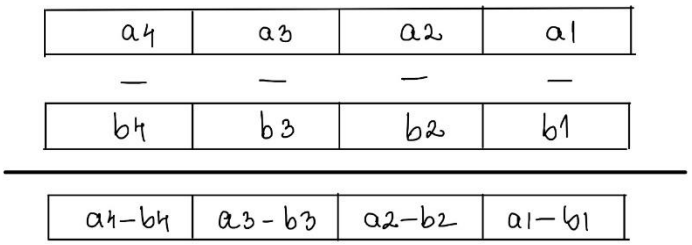
3.1 PADDD (Add with wrap-around on doubleword)

Efectuează adunarea împachetată a numerelor pe doubleword (32 de biți) utilizând wrap-around. Orice depășire care genereză transport este ignorată. Transportul nu se ia in considerare.



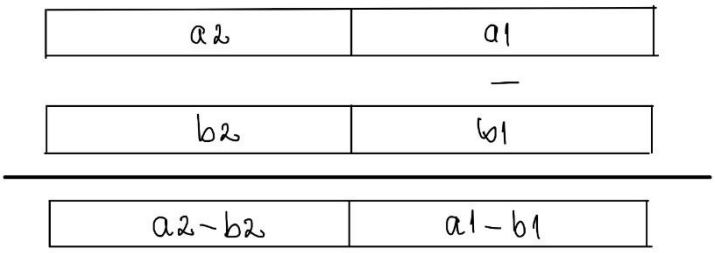
3.2 PSUBB (Subtraction with wrap-around on byte)

Efectuează scăderea împachetată a numerelor de bytes (8 biți) utilizând wrap-around. Ignoră transportul analog instrucțiunii PADDD.



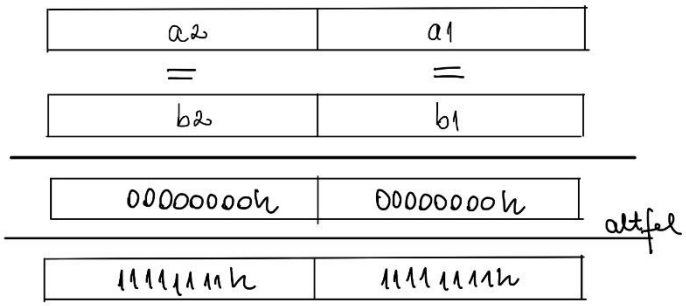
3.3 PSUBD (Subtraction with wrap-around on doubleword)

Efectuează scăderea analog PSUBB, dar pe doublewords.



3.4 PCMPEQD (Packed compare for equality on doubleword)

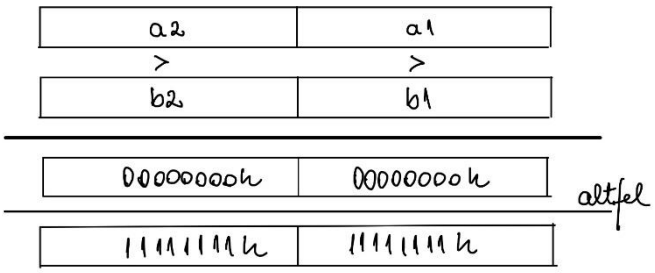
Efectuează compararea împachetată a operanzilor pentru a verifica egalitatea. Această instrucțiune compară două perechi de doublewords. Dacă perechile corespunzătoare se retureaza true, altfel false.



3.5 PCMPGTD (Packed compare greater than on doubleword)

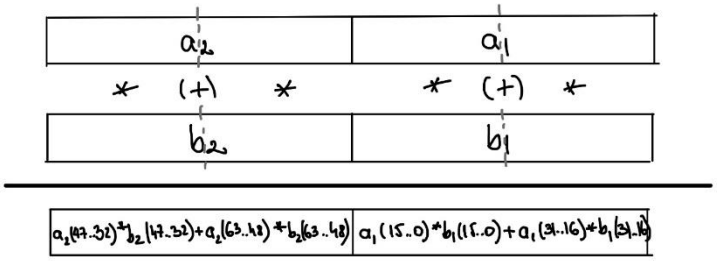
Efectuează compararea împachetată a operanzilor pentru a testa dacă este mai mare. Compară două perechi de doubleword. Dacă rezultatul este adevărat se setează true, altfel false.

Diferența față de operația obișnuită de comparare este că această instrucțiune MMX compară al doilea operand cu primul.



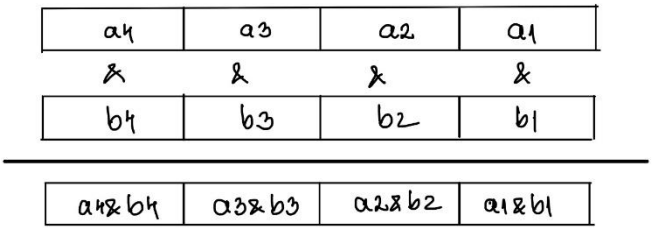
3.6 PMADDWD (Packed multiply on words and add resulting pairs)

Efectuează înmulțirea celor patru words din primul operand cu cele 4 words din cel de-al doilea pentru a produce patru produse doubleword pe care ulterior le adună două câte două (low order doubleword și high order doubleword).

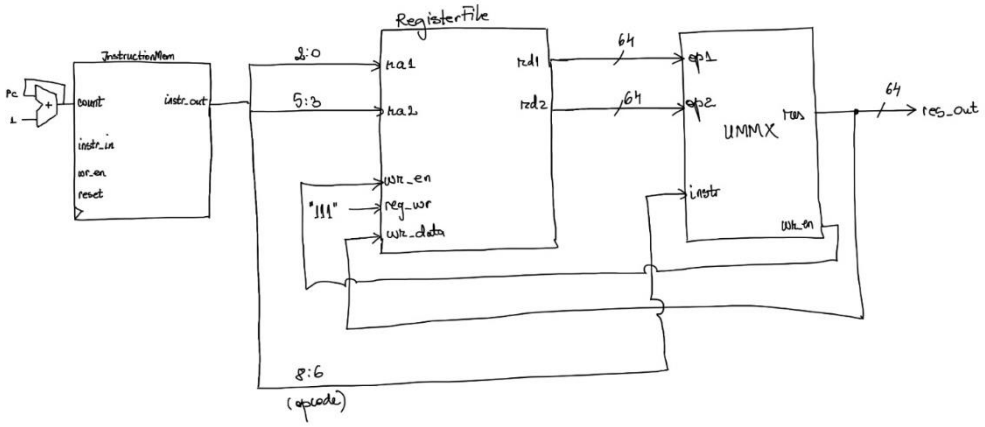


3.7 PANDB (Bitwise and)

Efectuează operația logică and pe biți.



4. Design



Principalele componente ale arhitecturii unității aritmetice sunt setul de instrucțiuni, registrul pentru stocarea datelor și unitatea aritmetică MMX.

Setul de instrucțiuni este stocat la nivelul unei memorii. Componenta se numește **InstructionMem** și prezintă un semnal de ceas, PC cu ajutorul căruia are loc stocarea instrucțiunii dorite în memorie și o ieșire inst_out reprezentând instrucțiunea care urmează să fie executată.

Instrucțiunile sunt stucturate într-un șir de 9 biți, primii 3 cei mai semnificativi reprezintă opcode-ul specific instrucțiunii, următorii 3 adresa primului operand, iar ultimii 3 adresa celui de-al doilea operand.

Datele cu care se operează sunt stocate la nivelului unui set de regiștrii, componentă numită **RegisterFile**. Acesta prezintă cei 8 regiștri a câte 64 de biți, specifici instrucțiunilor de tip MMX (MM0-MM7). De aici sunt citite datele cu care se operează la nivelul instrucțiunilor și este stocat rezultatul instrucțiunii curente (în cadrul registrului MM7).

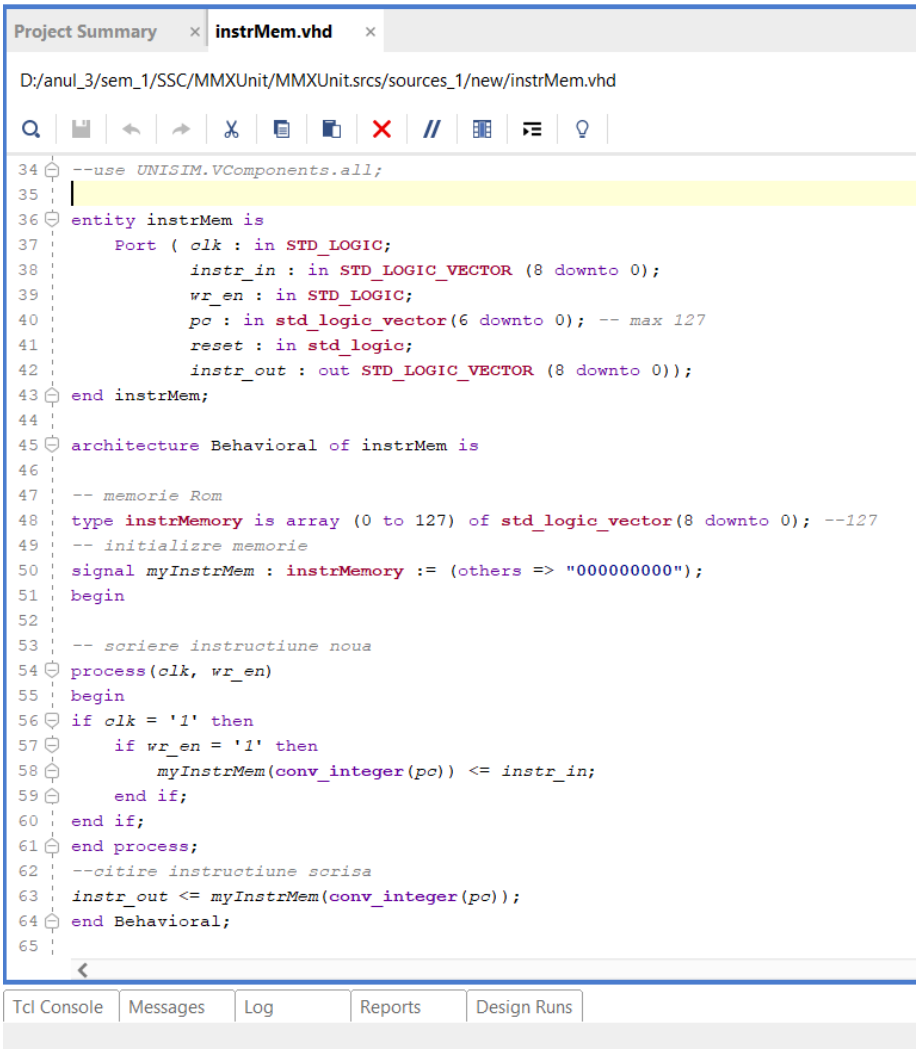
Cea mai importană componentă a arhitecturii este **Unitatea aritmetcă MMX** (UMMX). În cadrul acesteia sunt implementate istrucțiunile alese, care vor fi selectate pe baza opcode-ului unic. Operanzii cu care se lucrează sunt transmiși din RegiserFile, iar rezultatul obținut este afișat în simulare.

5. Implementare

5.1 Memoria de instrucțiuni

Odată transmise la intrare, instrucțiunile sunt salvate într-o memorie RAM care poate stoca până la 127 de instrucțiuni de lungime 9 biți (primii 3 biți repezintă opcode-ul operației,

următorii 3 adresa registrului primului operand, iar ultimii 3 adresa registrului celui de-al doilea operand). Poziția stocării este selectată cu ajutorul unui semnal dat de program counter. În urma stocării, instrucțiunea va fi ulterior citită și transmisă spre execuție.



5.2 Register file

Regiștrii cu care operează instrucțiunile MMX sunt stocați în cadrul unei alte memorii. Primii 7 regiștrii stochează valorile cu care vor lucra operațiile, iar ultimul registru este utlizat pentru memorarea rezultatului opearației curente.

Oată transmisă instrucțiuna selectată spre execuție, furnizează la intrarea în register file adresele regiștrilor cu care oprează ciclul curent. Adresa registrului în care se scrie rezultatul oprației este permanent setat pe “111”, iar semnalul de write enable se transmite din unitatea mmx, după finalizarea operației. La ieșirea din memoria de regiștrii sunt transmiși operanzii spre efectuarea operației.

Project Summary | regFile.vhd

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/regFile.vhd

```
35
36  entity regFile is
37    Port ( clk : in std_logic;
38           ra1 : in std_logic_vector(2 downto 0); -- adresa din instr
39           ra2 : in std_logic_vector(2 downto 0); -- adresa din instr
40           w_en : in std_logic; -- activ cand se genereaza rez op
41           reg_write : in std_logic_vector (2 downto 0); -- adresa pt stocarea rez
42           wd : in std_logic_vector(63 downto 0); -- rez final
43           rd1 : out std_logic_vector(63 downto 0); -- operand 1 din reg
44           rd2 : out std_logic_vector(63 downto 0) -- operand 2 din reg
45     );
46  end regFile;
```

Project Summary | regFile.vhd

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/regFile.vhd

```
50  -- memorie de registri MM0 -> MM7 (registrii MMX)
51  type regMemory is array (0 to 7) of std_logic_vector(63 downto 0);
52  signal myReg : regMemory := (x"0000000000000001",
53                               x"0000000000000002",
54                               x"0000000000000003",
55                               x"0000000000000004",
56                               x"0000000000000005",
57                               x"0000000000000006",
58                               x"0000000000000007",
59                               x"0000000000000000");
60
61  begin
62  -- citesc valorile pentru operanzi
63  process(ra1, ra2)
64  begin
65      rd1 <= myReg(conv_integer(ra1));
66      rd2 <= myReg(conv_integer(ra2));
67  end process;
68
69  -- scriu rezultatul final in ultimul reg
70  process(clk, w_en, wd)
71  begin
72  if falling_edge(clk) then
73      if w_en = '1' then
74          myReg(conv_integer(reg_write)) <= wd;
75      end if;
76  end if;
77  end process;
78
79
80  end Behavioral;
81
```

5.3 Unitatea MMX

Similar cu o unitate aritmetico-logică, în cadrul acestei componente sunt implementate operațiile ca și funcții.

Apelul fiecărei funcții se face în cadrul unei structuri case în baza opcode-ului extras din instrucțiunea de intrare. La finalizarea execuției este activat semnalul de write pentru scrierea rezultatului in register file.

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/UMMX.vhd

```
58  -- PSUBB
59  function psubb(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
60  variable result : std_logic_vector(63 downto 0);
61  begin
62
63      result(7 downto 0) := a(7 downto 0) - b(7 downto 0);
64      result(15 downto 8) := a(15 downto 8) - b(15 downto 8);
65      result(23 downto 16) := a(23 downto 16) - b(23 downto 16);
66      result(31 downto 24) := a(31 downto 24) - b(31 downto 24);
67      result(39 downto 32) := a(39 downto 32) - b(39 downto 32);
68      result(47 downto 40) := a(47 downto 40) - b(47 downto 40);
69      result(55 downto 48) := a(55 downto 48) - b(55 downto 48);
70      result(63 downto 56) := a(63 downto 56) - b(63 downto 56);
71
72  return result;
73
74  end function psubb;
75
76  -- PSUBD
77  function psubd(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
78  variable result : std_logic_vector(63 downto 0);
79  begin
80
81      result(31 downto 0)  := a(31 downto 0) - b(31 downto 0);
82      result(63 downto 32) := a(63 downto 32) - b(63 downto 32);
83
84  return result;
85
86  end function psubd;
```

Project Summary | UMMX.vhd

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/UMMX.vhd

```
34
35  entity UMMX is
36    Port (  op1 : in std_logic_vector(63 downto 0);
37            op2 : in std_logic_vector(63 downto 0);
38            instr : in std_logic_vector(8 downto 0);
39            res : out std_logic_vector(63 downto 0);
40            wr_en : out std_logic
41     );
42  end UMMX;
43
44  architecture Behavioral of UMMX is
45
46  -- PADDD
47  function paddd(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
48  variable result : std_logic_vector(63 downto 0);
49  begin
50
51      result(31 downto 0)  := a(31 downto 0) + b(31 downto 0);
52      result(63 downto 32) := a(63 downto 32) + b(63 downto 32);
53
54  return result;
55
56  end function paddd;
```

Project Summary | UMMX.vhd

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/UMMX.vhd

```
88  -- PCMPEQD
89  function pcmpeqd(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
90  variable result : std_logic_vector(63 downto 0);
91  begin
92
93      if a(31 downto 0) = b(31 downto 0) then result(31 downto 0) := x"11111111"; else result(31 downto 0) := x"00000000"; end if;
94      if a(63 downto 32) = b(63 downto 32) then result(63 downto 32) := x"11111111"; else result(63 downto 32) := x"00000000"; end if;
95
96  return result;
97
98  end function pcmpeqd;
99
100 -- PCMPGTD
101 function pcmpgtd(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
102 variable result : std_logic_vector(63 downto 0);
103 begin
104
105     if b(31 downto 0) >= a(31 downto 0) then result(31 downto 0) := x"11111111"; else result(31 downto 0) := x"00000000"; end if;
106     if b(63 downto 32) >= a(63 downto 32) then result(63 downto 32) := x"11111111"; else result(63 downto 32) := x"00000000"; end if;
107
108 return result;
109
110 end function pcmpgtd;
111
112 -- PAND
113 function pand(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
114 variable result : std_logic_vector(63 downto 0);
115 begin
116
117     result(63 downto 0) := a(63 downto 0) and b(63 downto 0);
118
119 return result;
```

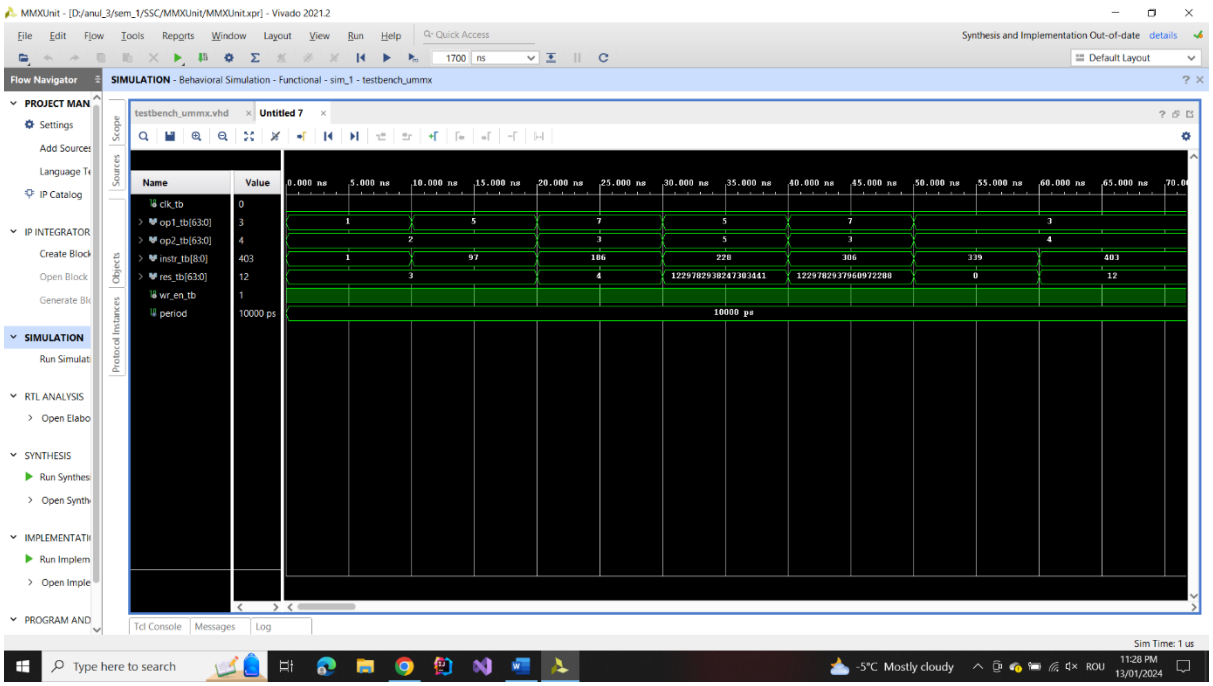
Project Summary | UMMX.vhd

D:/anul_3/sem_1/SSC/MMXUnit/MMXUnit.srcs/sources_1/new/UMMX.vhd

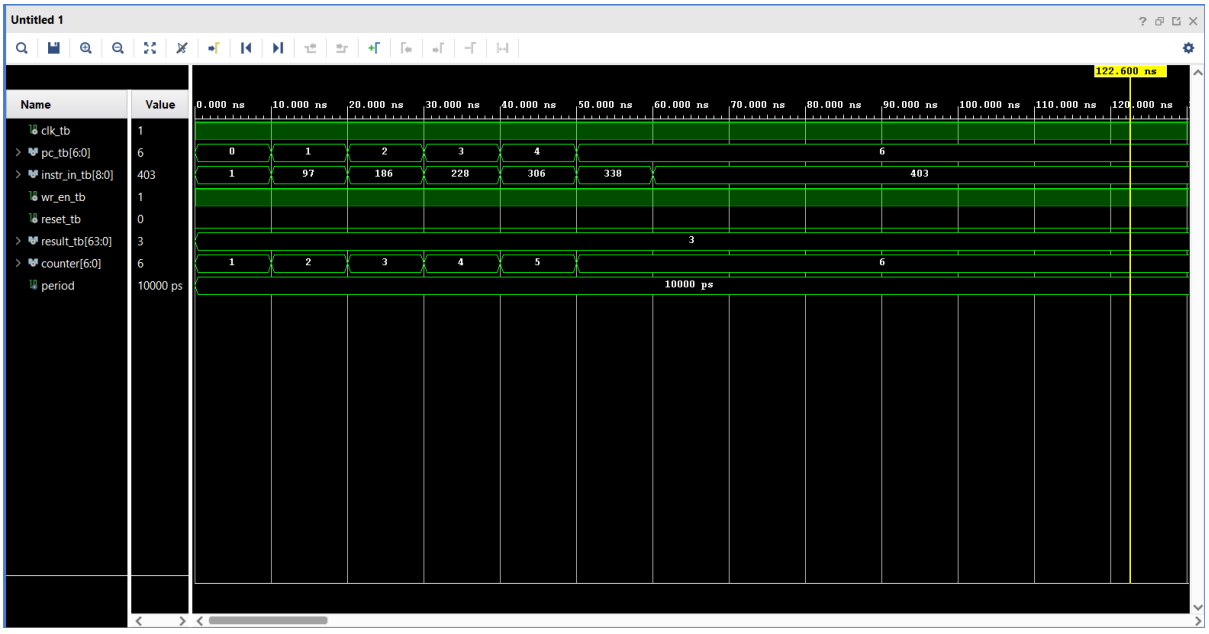
```
122
123 -- PMADDWD
124 function pmaddwd(a, b: std_logic_vector(63 downto 0)) return std_logic_vector is
125 variable result : std_logic_vector(63 downto 0);
126 begin
127
128     result(31 downto 0)  := (a(15 downto 0) * b(15 downto 0)) + (a(31 downto 16) * b(31 downto 16));
129     result(63 downto 32) := (a(47 downto 32) * b(47 downto 32)) + (a(63 downto 48) * b(63 downto 48));
130
131 return result;
132
133 end function pmaddwd;
134
135 signal result : std_logic_vector(63 downto 0) := x"0000000000000000";
136
137 begin
138 process(op1, op2, instr, result)
139 begin
140 case instr(8 downto 6) is
141     when "000" => result <= paddd(op1, op2); -- PADDD
142     when "001" => result <= psubb(op1, op2); -- PSUBB
143     when "010" => result <= psubd(op1, op2); -- PSUBD
144     when "011" => result <= pcmpeqd(op1, op2);-- PCMPEQD
145     when "100" => result <= pcmpgtd(op1, op2);-- PCMPGTD
146     when "101" => result <= pand(op1, op2);-- PAND
147     when "110" => result <= pmaddwd(op1, op2);-- PMADDWD
148     when others => result <= x"0000000000000000";
149 end case;
150 res <= result;
151 wr_en <= '1';
152 end process;
153
```

6. Testare și validare

Pentru testare am utilizat simularea din vivado. Am realizat un testbench pentru testarea operațiilor și unul pentru validarea întregii unități.



Pezultatele foarte mari pentru a 4-a și a 5-a operație sunt conversiile zecimale ale numrelor de 64 de biți activași, reprezentând rezultatul adevărat al comparațiilor.



*În simularea finală, valoarea rezultatului fiecărei operații nu se updatează, însă nu am identificat cauza. Se poate observa în schimb parcurgerea celorlalte semnale.

7. Concuzii

Implemetarea unității MMX a fost principalul scop al acestui proiect. Instrucțiunile alese au fost implemetate în cadrul unor funcții apelate corespunzător, iar comunicarea între componente a fost realizată prin intermediul semnalelor de legătura corespunzătoare. Finalitatea poate fi testată prin intermediul simulatii create în testbench-ul final.

8. Bibliografie

1. https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html
2. https://softpixel.com/~cwright/programming/simd/mmx.php
3. https://www.ecb.torontomu.ca/~courses/ele818/mmx.pdf
4. https://www.csie.ntu.edu.tw/~cyy/courses/assembly/docs/ch11_MMX.pdf