PROIECTAREA UNEI UNITĂȚI ARITMETICE DE TIP MMX

CRISTE CASIAN

Universitatea Tehnică din Cluj-Napoca 2024

PROIECTAREA UNEI UNITĂȚI ARITMETICE DE TIP MMX

1. Introducere

1.1 Context

Acest proiect are ca scop proiectarea și implementarea unei unități aritmetice de tip MMX. Instrucțiunile MMX permit procesorului x86 să efectueze operații SIMD (Single Instruction, Multiple Data) printr-o singură instrucțiune pe date întregi împachetate de tip byte, word, doubleword sau quadword, stocate în memorie, în registrele MMX sau în registrele generale. Unitatea aritmetică MMX este concepută pentru a efectua operații aritmetice și logice rapide și eficiente asupra datelor multi-media, conform specificațiilor standardului IEEE pe 64 de biți.

1.2 Motivație

Unitățile aritmetice de tip MMX cu setul de instrucțiuni menționat sunt esențiale pentru optimizarea operațiilor pe date multi-media într-un mediu de procesare paralelă. Acestea sunt utilizate pentru a accelera operațiile complexe precum adunarea, scăderea, comparația și alte manipulări logice asupra datelor, fiind utile în diverse aplicații precum grafică, multi-media, și procesare a datelor.

1.3 Obiective

Obiectivul acestui proiect este de a proiecta și implementa o unitate aritmetică de tip MMX, integrând instrucțiunile paddb, psubb, pxor, psllq, psrlw și pcmpeqd. Această unitate va avea capacitatea de a efectua operații aritmetice și logice pe date multi-media conform standardului IEEE pe 64 de biți.

2. Studiu Bibliografic

Instrucțiunile MMX operează cu registre de 64 de biți, în special înregistrări MMX, și efectuează operații paralele pe datele stocate în aceste registre. Ele sunt utilizate pentru sarcini precum procesarea imaginilor, compresia și decomprimarea video, efectele grafice etc.

Iată câteva dintre instrucțiunile MMX pe 64 de biți pe care le vom implementa în proiect și vom vedea cum sunt folosite pentru manipularea datelor:

1. **paddb (Addition of Packed Bytes)**:
    - **paddb** realizează o adunare element-cu-element a 8 seturi de octeți (8 biți fiecare) din două registre MMX, rezultând un alt set de 8 octeți (64 biți).
    - De exemplu, **paddb** poate aduna fiecare octet dintr-o înregistrare cu octetul corespondent dintr-o altă înregistrare, efectuând adunări pe 8 biți simultan.

2. **psubb (Subtraction of Packed Bytes)**:
    - Similar cu **paddb**, dar efectuează scăderi pe 8 biți între octeții din două registre MMX.

3. **pxor (Logical Exclusive OR)**:
    - **pxor** efectuează o operație de XOR logic între fiecare pereche de octeți din două registre MMX.

4. **psllq (Logical Shift Left Quadword)**:
    - Realizează o deplasare logică la stânga a datelor înregistrate într-o înregistrare MMX pe 64 de biți.

5. **psrlw (Logical Shift Right Word)**:
    - Efectuează o deplasare logică la dreapta a datelor lucrând cu pachete pe 16 biți înregistrate într-o înregistrare MMX.

6. **pcmpeqd (Compare Packed Doubleword Data for Equality)**:
    - Compară două registre MMX pe 64 de biți și în caz de egalitate efectuează un salt la adresa precizată.

Aceste instrucțiuni permit prelucrarea simultană și paralelă a datelor, sporind eficiența operațiilor pentru anumite tipuri de aplicații, cum ar fi procesarea video, grafică sau alte sarcini care necesită manipularea rapidă a datelor multimedia.

3. Analiză și Design

3.1 Analiză

În cadrul proiectului de față este necesară analiza detaliată a fiecărei instrucțiuni în parte pentru stabilirea codului mașină respectiv și a semnalelor care fac posibilă calcularea datelor.

Pentru instrucțiunile de tip R vom avea avea biții împărțiți in următorul mod:

Opcode	Registrul s	Registrul t	Registrul d	Shift amount	Func
6	5	5	5	5	6

Opcode – codul specific instrucțiunilor de tip R
Registrul s – primul registru folosit pentru calculul datelor
Registrul t – al doilea registru
Registrul d – registrul unde se salvează calculul
Shift amount – cantitatea cu care se shiftează (la instr. de tip shift)
Func – un cod specific pentru a diferenția instrucțiunile între ele

Ex. 000000_00001_00010_00011_00000_000000

Pentru singura instrucțiune de tip I vom folosi următoarea structură:

Opcode	Registrul s	Registrul t	Adresa de salt
6	5	5	16

Opcode – codul specific instrucțiunilor de tip I
Registrul s – primul registru
Registrul t – al doilea registru
Adresa de salt – adresa unde se va efectua saltul

Ex. 000001_00010_00011_0000000000000001

În continuare vom analiza fiecare instrucțiune pe rând:

3.1.1. **PADDB**
Descriere: Adună două registre și memorează rezultatul în al treilea
Tip: R
Operație: $d <- $s + $t; PC <- PC + 4;
Sintaxă: paddb $d, $s, $t
Format: 000000 sssss ttttt ddddd 00000 000000

3.1.2. **PSUBB**
Descriere: Scade două registre și memorează rezultatul în al treilea
Tip: R

Operație: $d <- $s - $t; PC <- PC + 4;
Sintaxă: psubb $d, $s, $t
Format: 000000 sssss ttttt ddddd 00000 000001

3.1.3. **PXOR**
Descriere: SAU exclusiv logic între două registre, memorează rezultatul în alt registru
Tip: R
Operație: $d <- $s ^ $t; PC <- PC + 4;
Sintaxă: pxor $d, $s, $t
Format: 000000 sssss ttttt ddddd 00000 000010

3.1.4. **PSLLQ**
Descriere: Deplasare logică la stânga pentru un registru, rezultatul este memorat în altul, se introduc zerouri
Tip: R
Operație: $d <- $t << h; PC <- PC + 4;
Sintaxă: psllq $d, $t, h
Format: 000000 00000 ttttt ddddd hhhhh 000011

3.1.5. **PSRLW**
Descriere: Deplasare logică la dreapta pentru un registru, rezultatul este memorat în altul, se introduc zerouri
Tip: R
Operație: $d <- $t >> h; PC <- PC + 4;
Sintaxă: psrlw $d, $t, h
Format: 000000 00000 ttttt ddddd hhhhh 000100

3.1.6. **PCMPEQD**
Descriere: Salt condiționat dacă este egalitate între două registre
Tip: I
Operație: if $s == $t then PC <- PC + 4 + (offset << 2); else PC <- PC + 4;
Sintaxă: pcmpeqd $s, $t, offset
Format: 000001 sssss ttttt iiiiiiiiiiiiiiii

3.2. Design

În cadrul etapei de design vom stabili prima dată semnalele de control de care ne vom folosi pentru a putea implementa instrucțiunile.

Instrucțiune	**Opcode** *Instr(31-26)*	**RegDst**	**ExtOp**	**ALUSrc**	**Branch**	**RegWrite**	**ALUOp**	**func** *Instr(5-0)*	**ALUCtrl (2:0)**
PADDB	000000	1	0	0	0	1	0	000000	000 (+)
PSUBB	000000	1	0	0	0	1	0	000001	001 (-)
PXOR	000000	1	0	0	0	1	0	000010	010 (^)
PSLLQ	000000	1	0	0	0	1	0	000011	011 (<<)
PSRLW	000000	1	0	0	0	1	0	000100	100 (>>)
PCMPEQD	000001	0	1	1	1	0	1 (-)	XXXXXX	101 (-)

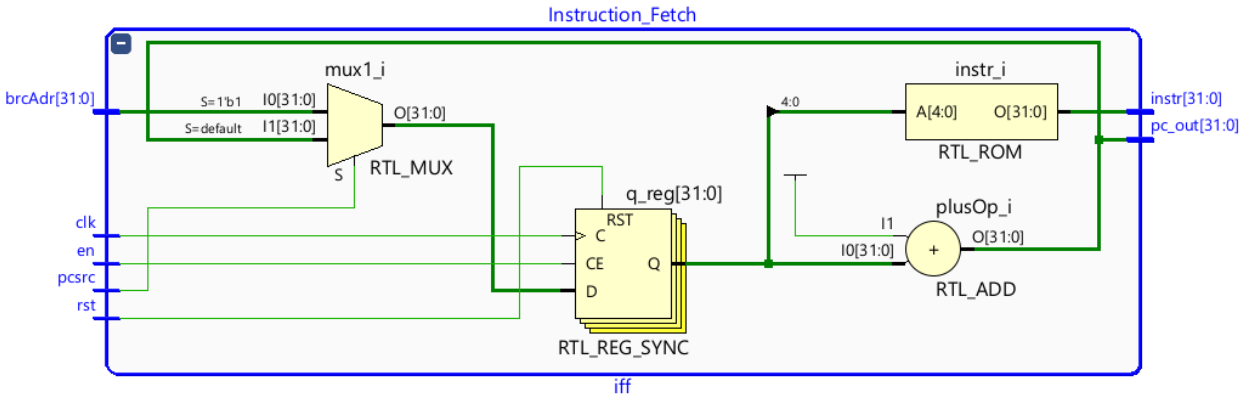
Opcode – codul folosit pentru a diferentia instructiunile după tipul lor
RegDst – dacă se folosește registru destinație
ExtOp – folosit la extinderea biților
ALUSrc – folosit doar la instrucțiunea de salt pentru stabilirea extinsului
Branch – folosit doar la instructiunea de salt
RegWrite – folosit atunci când se scrie într-un registru
ALUOp – necesar la diferențierea dintre instrucțiuni după tip
Func – necesar la diferențierea instrucțiunilor de tip R
ALUCtrl – operațiile folosite pentru fiecare instrucțiune

Design-ul proiectului este foarte asemănător cu MIPS32. Astfel că vom avea aceleași componente, doar implementarea lor va fi puțin mai diferită și totodată mai minimalistă datorită numărului mic de instrucțiuni. Ceea ce se va schimba complet va fi unitatea ALU (unitatea care se ocupă cu calculul datelor) datorită instrucțiunilor MMX care operează pe seturi de biți.

3.2.1. **Instruction Fetch**

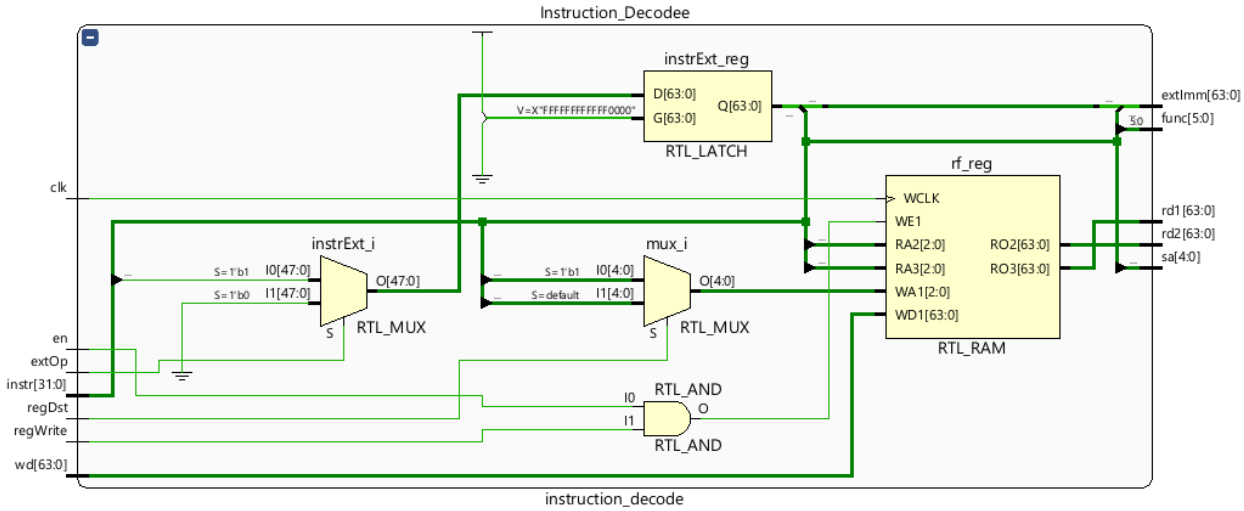
În această componentă se află codul mașină a fiecărei instrucțiuni pe care a introdus utilizatorul și componenta se ocupă cu parsarea biților instrucțiunilor la diferite componente pentru efectuarea

calculelor. Totodată, aici se va efectua trecerea la următoarea instrucțiune prin adunarea cu 1 la registrul Program Counter.



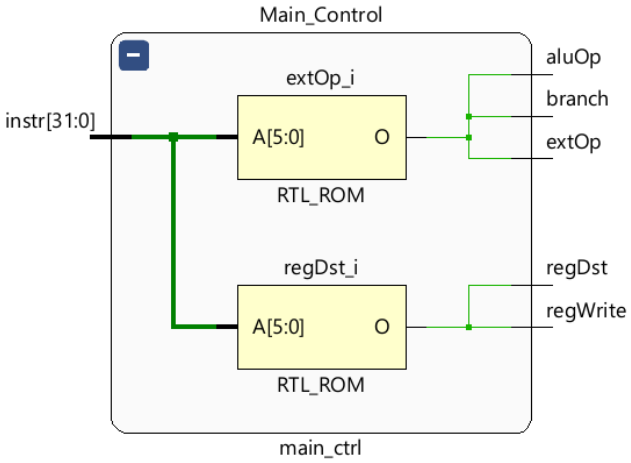
3.2.2. **Instruction Decode**

Aici se stochează datele pe care vom opera într-un ROM și se vor parsa biții din instrucțiune pentru stabilirea primului registru, respectiv celui de al doilea. Totodată, în această componentă se va realiza și operația de extindere.



3.2.3. **Main Control**

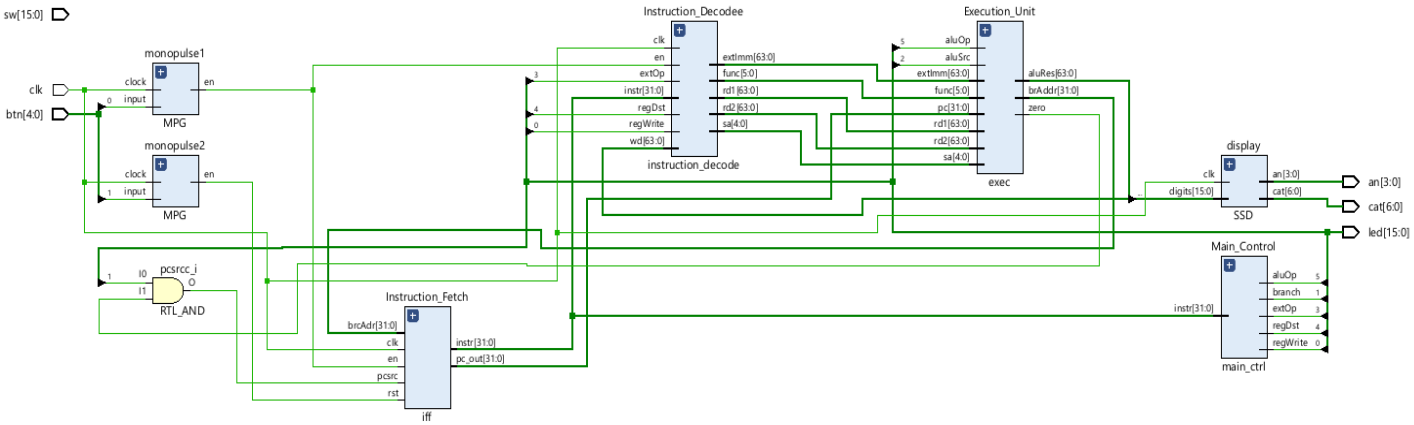
În această componentă se vor alege semnalele pentru fiecare tip de instrucțiune. Practic de aici se alege ce se va face cu fiecare instrucțiune în parte.



3.2.4. **Execution Unit**

Aici se produce calculul efectiv al fiecărei instrucțiuni în parte. În funcție de operație, se va alege pe ce ramură de calcul se va merge. De exemplu, pentru operația PADDB se va merge pe ramura cu codul aluCtrl respectiv și se va efectua operația de adunare. Vom intra mai în detaliu în această componentă în următorul capitol.

Designul schemei finale:



4. Implementare

În acest capitol este prezentată implementarea componentei de calcul, respectiv a componentei ALU din Execution Unit.

```
case aluCtrl is
    when "000" => cc0 <= ('0' & a(7 downto 0)) + ('0' & b(7 downto 0));
                  cc1 <= cc0(8) + ('0' & a(15 downto 8)) + ('0' & b(15 downto 8));
                  cc2 <= cc1(8) + ('0' & a(23 downto 16)) + ('0' & b(23 downto 16));
                  cc3 <= cc2(8) + ('0' & a(31 downto 24)) + ('0' & b(31 downto 24));
                  cc4 <= cc3(8) + ('0' & a(39 downto 32)) + ('0' & b(39 downto 32));
                  cc5 <= cc4(8) + ('0' & a(47 downto 40)) + ('0' & b(47 downto 40));
                  cc6 <= cc5(8) + ('0' & a(55 downto 48)) + ('0' & b(55 downto 48));
                  cc7 <= cc6(8) + ('0' & a(63 downto 56)) + ('0' & b(63 downto 56));
                  c <= cc7(7 downto 0) & cc6(7 downto 0) & cc5(7 downto 0) & cc4(7 downto 0) & cc3(7 downto 0) & cc2(7 downto 0) & cc1(7 downto 0) & cc0(7 downto 0);

    when "001" => c0 <= a(7 downto 0) - b(7 downto 0);
                  br <= not a(7 downto 0) and b(7 downto 0);

                  c1 <= a(15 downto 8) xor b(15 downto 8) xor br;
                  br <= ((not a(15 downto 8) and b(15 downto 8)) or (b(15 downto 8) and br) or (not a(15 downto 8) and br));

                  c2 <= a(23 downto 16) xor b(23 downto 16) xor br;
                  br <= ((not a(23 downto 16) and b(23 downto 16)) or (b(23 downto 16) and br) or (not a(23 downto 16) and br));

                  c3 <= a(31 downto 24) xor b(31 downto 24) xor br;
                  br <= ((not a(31 downto 24) and b(31 downto 24)) or (b(31 downto 24) and br) or (not a(31 downto 24) and br));

                  c4 <= a(39 downto 32) xor b(39 downto 32) xor br;
                  br <= ((not a(39 downto 32) and b(39 downto 32)) or (b(39 downto 32) and br) or (not a(39 downto 32) and br));

                  c5 <= a(47 downto 40) xor b(47 downto 40) xor br;
                  br <= ((not a(47 downto 40) and b(47 downto 40)) or (b(47 downto 40) and br) or (not a(47 downto 40) and br));

                  c6 <= a(55 downto 48) xor b(55 downto 48) xor br;
                  br <= ((not a(55 downto 48) and b(55 downto 48)) or (b(55 downto 48) and br) or (not a(55 downto 48) and br));

                  c7 <= a(63 downto 56) xor b(63 downto 56) xor br;

                  c <= c7 & c6 & c5 & c4 & c3 & c2 & c1 & c0;

    when "010" => c <= a xor b;

    when "011" => case sa is
                    when "00000" => c <= b;
                    when "00001" => c <= b(62 downto 0) & '0';
                    when "00010" => c <= b(61 downto 0) & "00";
                    when "00011" => c <= b(60 downto 0) & "000";
                    when "00100" => c <= b(59 downto 0) & "0000";
                    when "00101" => c <= b(58 downto 0) & "00000";
                    when others => c <= (others => 'X');
                  end case;

    when "100" => case sa is
                    when "00001" => c(63 downto 48) <= '0' & b(63 downto 49);
                                    c(47 downto 32) <= b(48) & b(47 downto 33);
                                    c(31 downto 16) <= b(32) & b(31 downto 17);
                                    c(15 downto 0) <= b(16) & b(15 downto 1);
                    when "00010" => c(63 downto 48) <= "00" & b(63 downto 50);
                                    c(47 downto 32) <= b(49 downto 48) & b(47 downto 34);
                                    c(31 downto 16) <= b(33 downto 32) & b(31 downto 18);
                                    c(15 downto 0) <= b(17 downto 16) & b(15 downto 2);
                    when "00011" => c(63 downto 48) <= "000" & b(63 downto 51);
                                    c(47 downto 32) <= b(50 downto 48) & b(47 downto 35);
                                    c(31 downto 16) <= b(34 downto 32) & b(31 downto 19);
                                    c(15 downto 0) <= b(18 downto 16) & b(15 downto 3);
                    when others => c <= (others => '0');
                  end case;

    when "101" => c <= a - b;

    when others => c <= (others => 'X');
end case;
end process;
```

PADDB

Se execută la “000”. La fiecare octet in parte se adună 0 pentru gestionarea mai ușoară a Carry-ului și se adună cu octetul corespunzător lui. La final se concatenează octeții pentru rezultatul final.

PSUBB

Se execută la “001”. Aproximativ pe același principiu ca și adunarea, insă nu ne mai folosim de incă un bit ‘0’ în plus și vom lucre cu operații logice. Operăm pe fiecare octet în parte și la final concatenăm toți octeții.

PXOR

Se execută la “010”. Se face o operație simpla de XOR.

PSLLQ

Se execută la “011”. Aici se concatenează biții de la indexul (63 – nr de zerouri) până la 0 cu atâția de zero câți se precizează în câmpul Shift amount.

PSRLW

Se execută la “100”. Aici se introduc la început atâtea zerouri câte se precizează în Shift amount și se iau pe rând biții din fiecare octet și se concatenează la finalul numarului.

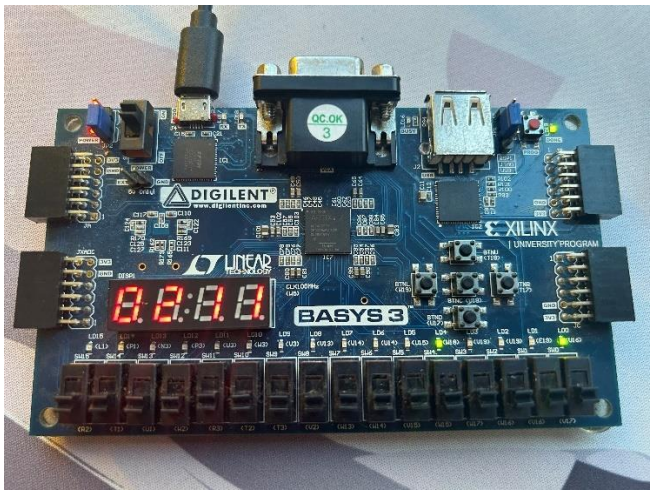
PCMPEQD

Se execută la “101”. Aici pur și simplu se efectuează o scădere simplă intre cei doi regiștrii.

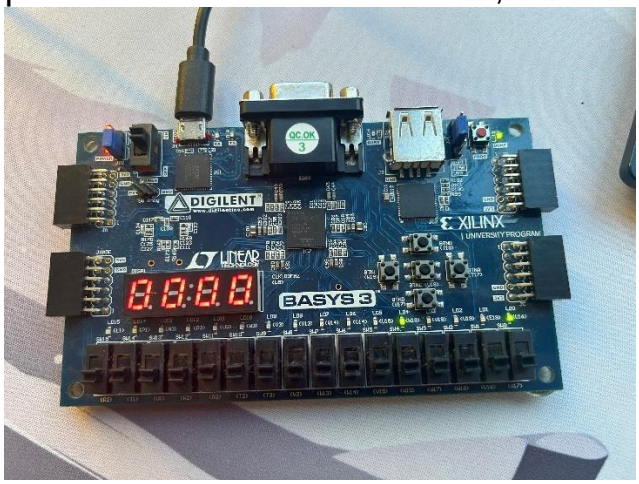
5. Testare și Validare

Proiectul a fost testat direct pe plăcuța Basys 3. Pe afișorul SSD vom urmări rezultatul fiecărei instrucțiuni. Cu ajutorul butonului U18 vom trece prin toate instrucțiunile, iar cu butonul T18 revenim la prima instrucțiune (reset). Rezultatul afișat pe plăcuță e in hexazecimal.

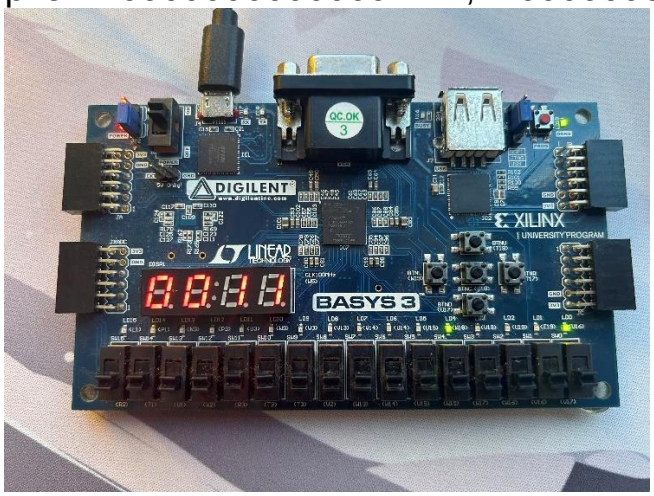
- paddb x"0000000000000111", x"0000000000000100" = 0211



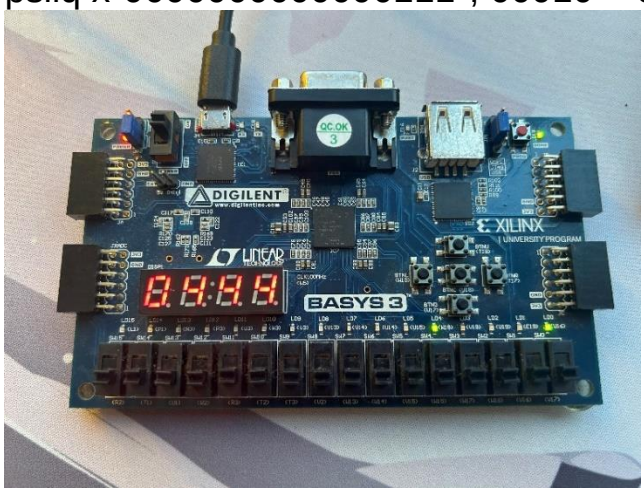
- psubb x"0000000000000111", x"0000000000000111" = 0000



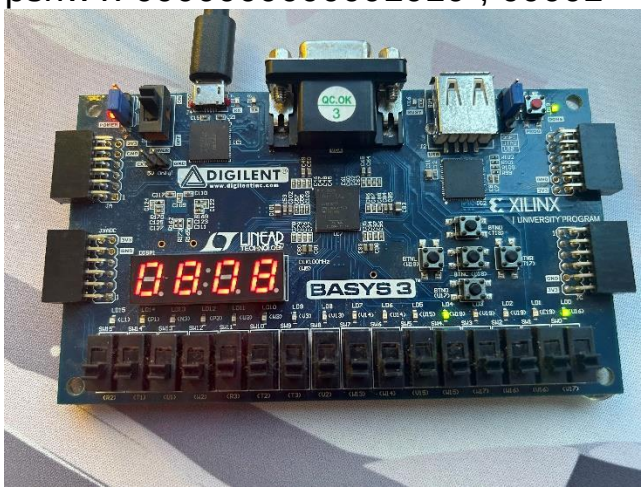
- pxor x"0000000000000111", x"0000000000000100" = 0011



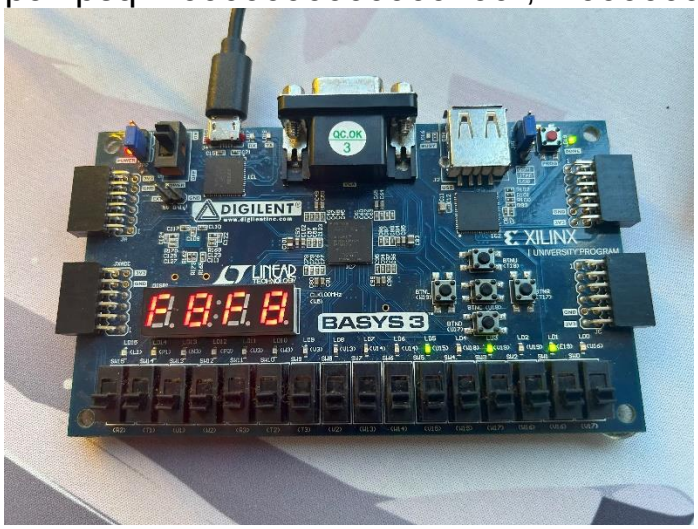
- psllq x"0000000000000111", 00010 = 0444



- psrlw x"0000000000001010", 00001 = 0808



- pcmpeq x"0000000000000100", x"0000000000000100" = F8F8



În cazul validării, toate instrucțiunile funcționează perfect, mai puțin PCMPEQ.

6. Concluzie

Proiectul a reprezentat o incursiune în lumea tehnologiilor avansate de procesare a datelor, concentrându-ne pe utilizarea instrucțiunilor MMX pentru manipularea eficientă a datelor pe 64 de biți. Prin această experiență, am explorat și am implementat un set variat de operații SIMD, îmbunătățind performanța și accelerând procesul de prelucrare a datelor multimedia.

Utilizând instrucțiunile MMX precum paddb, psubb, pxor, psllq, psrlw și pcmpeqd, am putut efectua operații de adunare, scădere, operații logice, deplasare și comparație între datele pe 64 de biți, deschizând astfel noi posibilități în domeniul prelucrării paralele a datelor. Pe viitor, instrucțiunea pcmpeqd va fi depanată pentru a putea realiza și salturi.

7. Bibliografie

https://docs.oracle.com/cd/E18752_01/html/817-5477/eojdc.html
https://ro.wikipedia.org/wiki/Pagina_principală