Unitatea Aritmetică MMX

Universitatea Tehnică din Cluj-Napoca
Brînzoi Ion-Robert
Grupa 30231

Cuprins

1 Introducere

Ideea principală a tehnologiei de tip MMX este reprezentată de execuția in paralel a unei instrucțiuni pe date care au structura de tip vector sau matrice, punându-se accent pe timpul de execuție.

1.1 Cerința

Scopul proiectului constă în proiectarea și implementarea a 6 instrucțiuni de tip MMX x86, și anume:

- PANDN (bitwise logical AND NOT)
- PADDSB/PADDSW (add packed signed byte/word integers with signed saturation)
- PCMPEQB/PCMPEQW/PCMPEQD (compare packed bytes/words/doublewords for equal)
- PMADDWD (multiply and add packed word integers)
- PSRLW/PSRLD/PSRLQ (shift packed words/doublewords/quadword right logical)
- PUNPCKLWD (unpack low-order words)

1.2 Specificația

Proiectul va fi scris în limbajul VHDL, simulat și ulterior încărcat pe placa FPGA Basys 3. Va permite implementarea instrucțiunilor menționate anterior și rezultatele aplicării acestora pe seturile de date va fi evidențiat pe afișorul pe 7 segmente. Dacă rezultatul este prea mare pentru cele 4 cifre hex afișate la un moment dat, care arată maxim 2 octeți la un moment dat, se va permite alternarea între fiecare sfert al rezultatului prin intermediul unor switch-uri.

2 Studiu bibliografic

Instrucțiunile MMX sunt realizate folosindu-se 8 registre: MM0, MM1, ..., MM7, fiecare având 64 de biți. Există astfel următoarele tipuri de date care pot fi prelucrate: [1]

- packed byte (8 octeți)

63 - 56	55 - 48	47 - 40	39 - 32	31 - 24	23 - 16	15 - 8	7-0

- packed word (4 cuvinte)

63 - 48	47 - 32	31 - 16	15 - 0

- packed dword (2 dublu cuvinte)

63 - 32	31 - 0

- quadword (un singur qcuvânt)

63 - 0

Instrucțiunile MMX pot lucra pe mai mulți întregi simultan cu un singur apel al instrucțiunii.

Întregii cu semn vor fi reprezentați în complement față de doi, primul bit fiind considerat bitul de semn (0 pentru pozitive, 1 pentru negative). Complementul se obține prin inversarea tuturor biților și adăugarea unui 1 numărului inversat.

La efectuarea operațiilor, unele rezultate pot fi în afara domeniului reprezentabil pentru tipul de date respectiv, iar tehnologia MMX oferă două metode pentru tratarea acestor cazuri: [1]

- **Wraparound (Întoarcere)** - rezultatul se trunchează (sunt păstrați cei mai puțini semnificativi biți)
- **Saturare** - rezultatul este înlocuit cu limita corespunzătoare domeniului său, conform tabelului de mai jos.

Tabela 1: Limitele de saturare[1]

Tip de date	Limita inferioară	Limita superioară
octet (fără semn)	00h	FFh
octet (cu semn)	80h	7Fh
cuvânt (fără semn)	0000h	FFFFh
cuvânt (cu semn)	8000h	7FFFh

2.1 PANDN

PANDN dest, src

Instrucțiunea PANDN efectuează operația logică pe biți NOT pe operandul sursă, și apoi AND pe biți cu cel de-al doilea operand, iar rezultatul se salvează în operandul destinație.[2] Aceasta operează pe valori de 64 de biți, fără a se face distincția între octeți, cuvinte, dublu cuvinte sau quad cuvănt, întrucât rezultatul ar fi identic în toate cazurile.

2.2 PADDSB/PADDSW

PADDSB/PADDSW op1, op2

PADDSB face operația de adunare pe octeți împachetați și stochează rezultatul în operandul destinație (op1). Când un rezultat individual de tip octet este în afara domeniului de reprezentare a unui octet întreg cu semn (adică mai mare ca 7Fh sau mai mic de 80H), valoarea saturată de 7Fh sau 80h, respectiv, este scrisă in operandul destinație. [2]

Similar se execută și PADDSW, doar că operează pe cuvinte, iar limitele vor fi 7FFFh, respectiv 8000h.

2.3 PCMPEQB/PCMPEQW/PCMPEQD

PCMPEQx op1, op2

Instrucțiunea performă compararea de egalitate pe octeți, cuvinte sau dublu cuvinte împachetate în operandul destinație (primul operand) și operandul sursă(al doilea operand). Dacă o pereche de elemente este egală, elementul corespunzător în operandul destinație va avea toți biții setați pe 1; în caz contrar, vor fi setați pe 0.[2]. Pentru compararea rezultatelor se vor folosi porți XOR, al căror rezultate vor comanda multiplexoare care vor avea ca intrari fie un număr cu toți biții activați, fie unul nul.

2.4 PMADDWD

PMADDWD op1, op2

Instrucțiunea interpretează datele de intrare ca și cuvinte, iar rezultatul va fi exprimat în dublu cuvinte, după urmatorul model:

result[31:0] = op1[15:0] * op2[15:0] + op1[31:16] * op2[31:16]
result[63:32] = op1[63:48] * op2[63:48] + op1[47:32] * op2[47:32]
op1 = result

Atunci când ambele perechi de cuvinte vor avea valorile 8000h, se va face wraparound, iar rezultatul va fi 80000000h.[2]

2.5 PSRLW/PSRLD/PSRLQ

PSRLx operand, count

Aceste instrucțiuni execută shiftarea la dreapta a primului operand cu numărul de poziții specificat ca și al doilea argument, care poate fi o valoare imediată pe 8 biți sau citită dintr-un alt registru MMX[2]. Pentru shiftarea cuvintelor, maximul este de 15 poziții, pentru dublu cuvinte este de 31, iar pentru quad cuvânt este de 63. Depășirea acestor valori duce la un rezultat nul în toate dintre cele 3 cazuri. Pentru implementarea instrucțiunii vor fi necesare circuite de deplasare, care vor fi implementate cu multiplexoare.

2.6 PUNPCKLWD

PUNPCKLWD op1, op2

Rezultatul va fi salvat în primul operand, iar instrucțiunea de despachetare a cuvintelor low-order se execută în următorul mod:

result[63:48] = op2[31:16]
result[47:32] = op1[31:16]
result[31:16] = op2[15:0]
result[15:0] = op1[15:0]
op1 = result

3 Analiza

3.1 Propunerea proiectului

La finalul implementării, mă aștept ca proiectul să poată efectua totalitatea operațiilor menționate și descrise mai sus, să treacă toate testele și să fie funcțional pe placă. Voi utiliza switch-urile, butoanele și afișorul pe 7 segmente de pe Basys 3 ca și instrumente de introducere a datelor și de afișare a rezultatului.

Va avea loc următoarea mapare a switch-urilor:

- sw[15:13] - pentru selectarea adresei din memorie dacă al doilea operand va fi un imediat.
- sw[12] - va avea valoarea zero dacă instrucțiunea va fi pe 2 registre, și 1 logic dacă va fi între un registru și un imediat
- sw[11:9], sw[8:6] - pentru selectarea adreselor celor 2 registre care vor fi manipulate
- sw[5:2] - pentru selectarea instrucțiunii
- sw[1:0] - pentru navigarea între cele 4 părți rezultatului.

Singurul buton folosit va fi BTNC, ca și buton de start, iar LED-urile vor fi mapate switch-urilor, pentru a facilita introducerea datelor.

Mod de utilizare

După încărcarea proiectului pe placă, utilizatorul poate naviga printre valorile regiștrilor folosind și switch-urile sw[8:6], respectiv sw[1:0], vizualizând astfel valoarea primului operand din blocul de registre. Pentru inițierea unei instrucțiuni, se vor seta adresele dorite pentru date, fie regiștrii fie din memorie, după care se va seta switch-ul 12 în modul dorit (pentru operații pe regiștri sau pe valori imediate). Se introduce apoi codul instrucțiunii, care va fi pe 4 biți (sw [5:2]). După setarea datelor, se va apăsa butonul de start (BTNC), și se va putea observa modificarea datelor afișate.

3.2 Plan de dezvoltare

- săptămâna 1 (03.10 - 09.10) - Alegere proiect
- săptămâna 2 (10.10 - 16.10) - Efectuarea introducerii și a studiului bibliografic
- săptămâna 3 (17.10 - 23.10) - Augumentarea informațiilor despre instrucțiuni din studiul bibliografic și proiectarea unităților top level: memoria de date, unitatea de execuție, mașina de stări (organigrama)
- săptămâna 4 (24.10 - 30.10) - Inceperea detalierii componentelor particulare care execută o instrucțiune în particular: unitatea de shiftare, de adunare
- săptămâna 5 (31.10 - 06.01) - Terminarea detalierii componentelor rămase din partea de design, precum și inceperea implementării în VHDL
- săptămâna 6 (7.11 - 13.11)- Terminarea implementării în VHDL și inceperea stadiului de testare.
- săptămâna 7 (14.11 - 20.11)- Testarea funcționalităților în Vivado și rezolvarea eventualelor probleme de implemetare
- săptămâna 8 (21.11 - 27.11) - Încărcarea proiectului pe placa FPGA

3.3 Descrierea funcționalităților

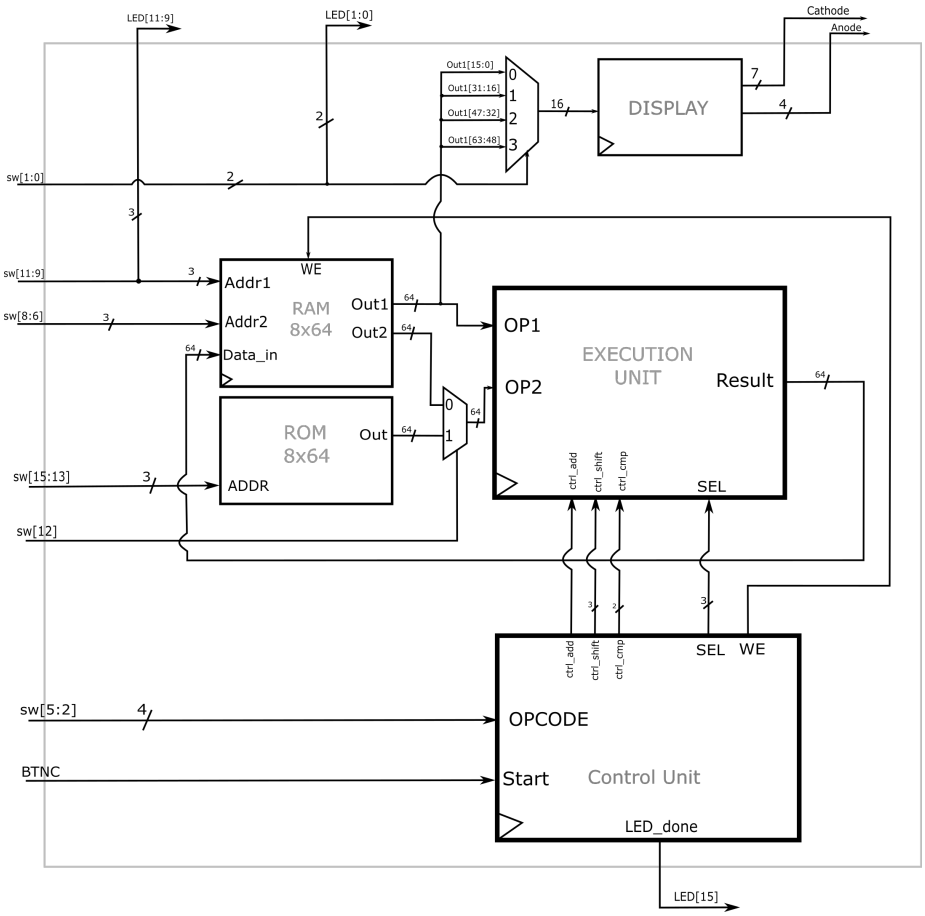

Figura 1: Unitățile necesare pentru implementarea instrucțiunilor.

Așa cum se vede în figura 1, voi folosi o memorie ROM pentru stocare valorilor imediate, iar memoria RAM reprezintă blocul de registre, având 2 adrese de citire, dar scrierea se face doar la prima, deoarece în fiecare instrucțiune implementată, destinația este reprezentată de primul operand.

Unitate de execuție primește de la unitatea de control semnalele care comandă ce fel de tip de operație trebuie să execute unitățile de adunare, shiftare sau comparare (adică dacă trebuie să execute operații pe byte, word, dword sau qword, în funcție de caz). Tot unitatea de control comandă începerea operațiilor de înmulțire, dacă se execută instrucțiunea PMADDWD. La finalizarea acesteia, se va trimite de către unitatea de execuție un semnal care indică faptul ca operația s-a executat, și că datele sunt pregătite de scriere.

Când rezultatul a fost calculat, prin semnalul SEL se va decide ce valoare trebuie scrisă, se va activa intrarea de scriere a blocului de registre, iar datele vor fi încărcate la adresa corespunzătoare. Se va trece apoi în prima stare, așteptându-se o nouă comandă de la utilizator. Funcționalitatea descrisă anterior este ilustrată în figura 2.

Tabela 2: Codificarea operațiilor

Instrucțiune	OPCODE
PANDN	0000
PMADDWD	0001
PADDSB	0010
PADDSW	0011
PSRLW	0100
PSRLD	0101
PSRLQ	0110
PCMPEQB	1000
PCMPEQW	1001
PCMPEQD	1010
PUNCPCKLWD	1111

3.4 Algoritmul de inmulțire

Înmulțirea va fi implementată după algoritmul de înmulțire prin adunări și deplasări repetate. Pentru acesta vom folosi un registru de stocare a rezultatelor parțiale și a celui final. Vom verifica pe rând, începând de la cel mai puțin semnificativ bit al înmulțitorului. Dacă acesta este zero, atunci facem deplasare la dreapta a produsului parțial, iar dacă este unu, la produsul parțial se va adăuga deînmulțitul, după care se realizează deplasarea. Se trece la următorul bit al înmulțitorului, și se repetă algoritmul până au fost verificați toti biții acestuia.

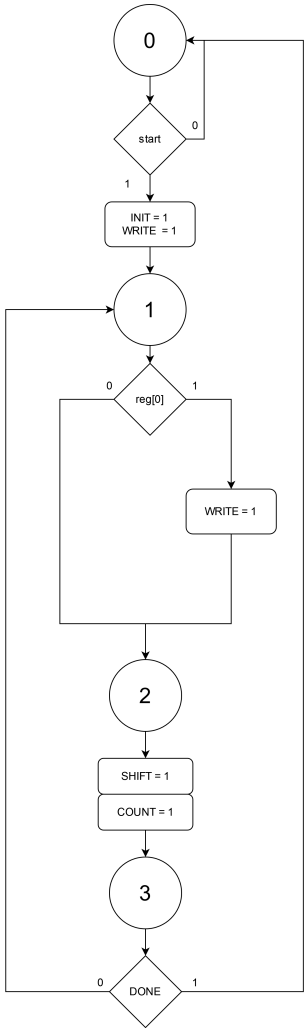

Figura 2: Organigrama ce descrie unitatea de control a înmulțitorului.

3.5 Unitatea de control

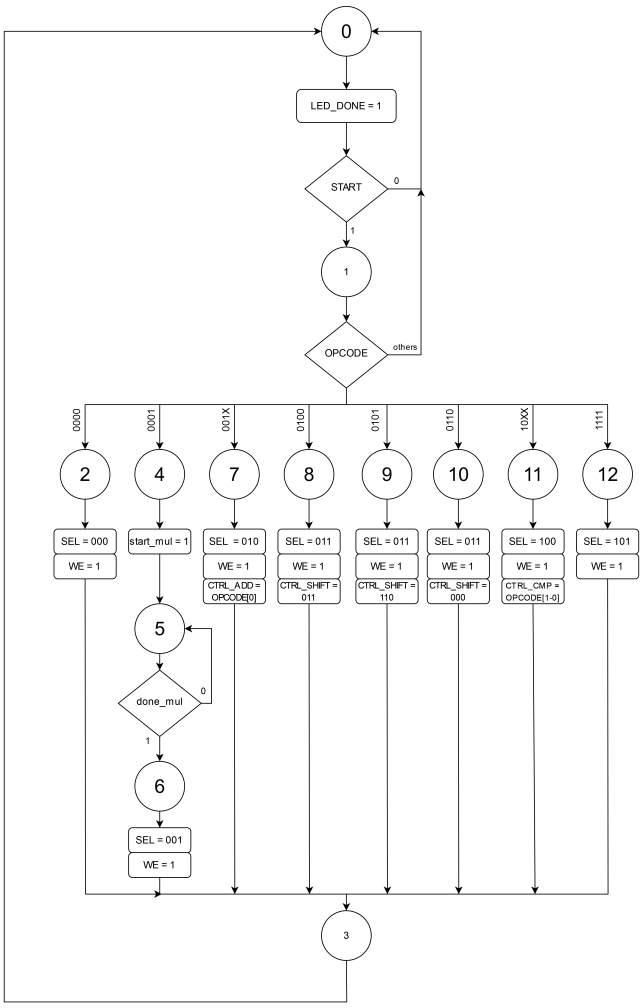

Figura 3: Organigrama care descrie unitatea de control a unității MMX.

4 Design

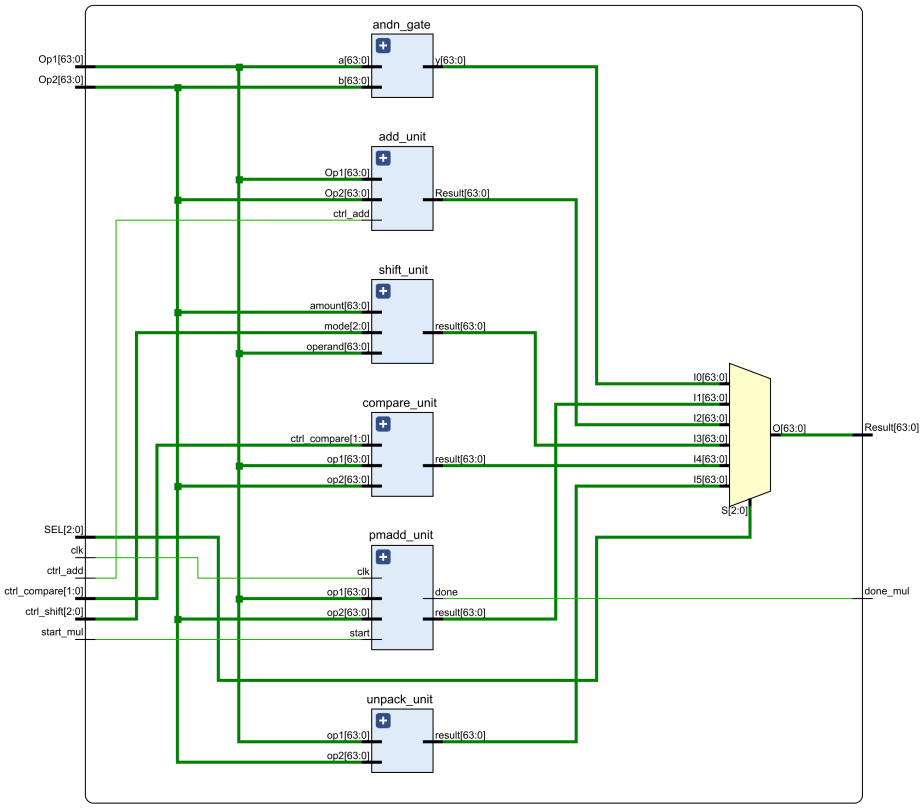

Figura 4: Componentele unității de execuție (ALU).

4.1 ANDN

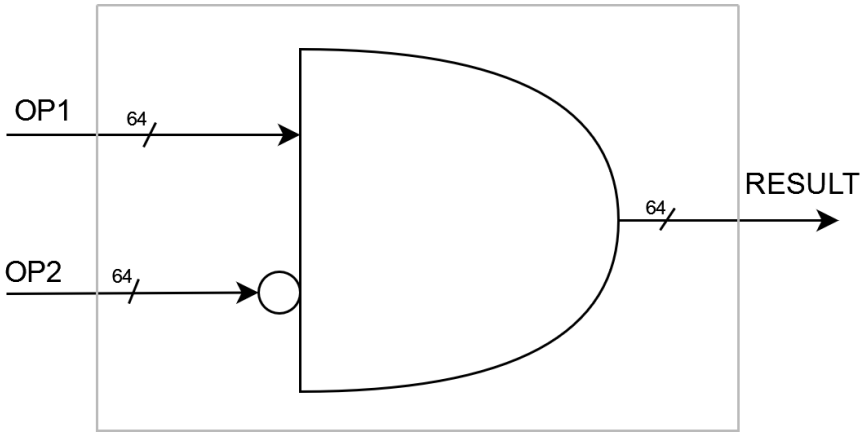

Figura 5: Unitatea ANDN.

Pentru instrucțiunea ANDN, am definit o poartă și cu o intrare negată, pe 64 de biți.

4.2 Unitatea de adunare

Pentru instrucțiunea de adunare, am folosit numărătoare pe 8 biți, care vor fi cascadate 2 câte 2, deoarece această operație lucrează pe octeți și pe cuvinte. Tocmai de aceea, transportul unui sumator trece printr-o poartă și cu semnalul ctrl_add. Când acesta este zero, transportul nu se propagă la următorul, deci se efectuează adunare pe byte, iar dacă este 1, se va face pe word.

Având în vedere că adunarea este cu semn, am salvat de la fiecare sumator un semnal de overflow, care îmi spune dacă s-a depășit domeniul de reprezentare. Pentru un sumator pe 8 biți format prin cascadare de sumatoare pe un bit, semnalul de overflow este dat de sau exclusiv între ultimele 2 transporturi (cout6 xor cout7). Overflow apare doar între adunare pe numere care au același semn. Pentru fiecare sumator în parte, calculăm limita care trebuie pusă în locul rezultatului, dacă este cazul, astfel: dacă avem overflow și rezultatul are cel mai semnificativ bit zero, punem limita inferioară (80h, 8000h), altfel, pe cea superioară (7Fh, 7FFFh).

Dacă suntem la nivel de byte, punem limita când avem overflow și semnalul de control este zero, altfel transmitem mai departe rezultatul adunării. Dacă operăm pe cuvinte, păstrăm rezultatul obținut pe primul nivel (al octeților), și decidem între acesta și limita domeniului, în cazul în care avem overflow și semnalul de control este pe 1 logic.

Logica adunării este identică pe prima jumătate a operanzilor cu cea de pe jumatatea inferioară a acestora.

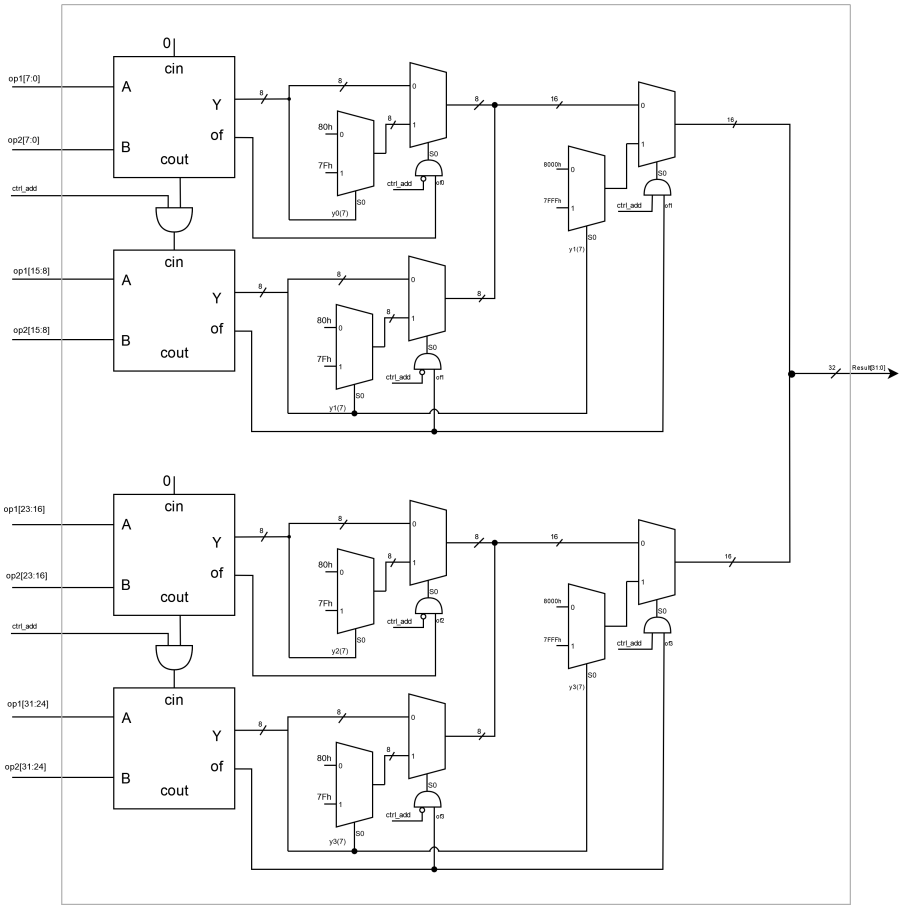

Figura 6: Componentele unității de adunare (partea inferioară).

4.3 Unitatea de egalitate

Are rolul de a implementa instrucțiunea de verificare a egalității pe octeți, cuvinte și dublu cuvinte. Pentru a verifica egalitatea între doi octeți, am efectuat operația sau exclusiv între aceștia. Dacă ar fi egali, rezulatul ar fi zero. Verific dacă rezultatul a fost sau nu zero printr-o operație sau nu asupra biților săi. Vom avea asfel un semnal pe un bit care ne va spune dacă octeții au fost sau nu egali.

Pentru PCMPEQB, folosim bitul de egalitate corespunzător fiecărei perechi de octeți ca și selecție a unui multiplexor. Când este 1, rezultatul va fi FFh, iar în caz contrar, 00h.

Pentru PCMPEQW, pentru ca o pereche de cuvinte să fie egale, biții de egalitate corespunzători ambilor octeți trebuie să fie pozitivi. Dacă sunt, punem rezultatul obținut pe nivelul anterior (care va fi FFFFh), altel 0000h. Trebuie să avem grijă în cazul în care vrem să facem operația doar pe octeți, să trimitem rezultatul nivelului precedent chiar dacă ambele perechi nu sunt egale, așa că multiplexorului îi vom adăuga condiția ca să treacă rezultatul precedent și atunci când semnalul de control a fost cel pentru PCMPEQB (adică 00).

Similar procedăm și pentru PCMPEQD, acum fiind necesară verificarea a 4 biți de egalitate. La fel ca în cazul precedent, trebuie să transmitem rezultatul nivelului anterior doar atunci când semnalul de control a fost 00 sau 01 (adică atunci când bitul cel mai semnificativ nu a fost 1).

Mai pe scurt, fiecare nivel este delimitat de multiplexoare, care au deci rolul de a transmite rezultatul până la ultimul nivel, și de a alege între calcularea altui rezultat conform nivelului din urmă și a semnalului de comandă. Partea superioară și partea inferioară a acestei componentei împart aceeași logică.

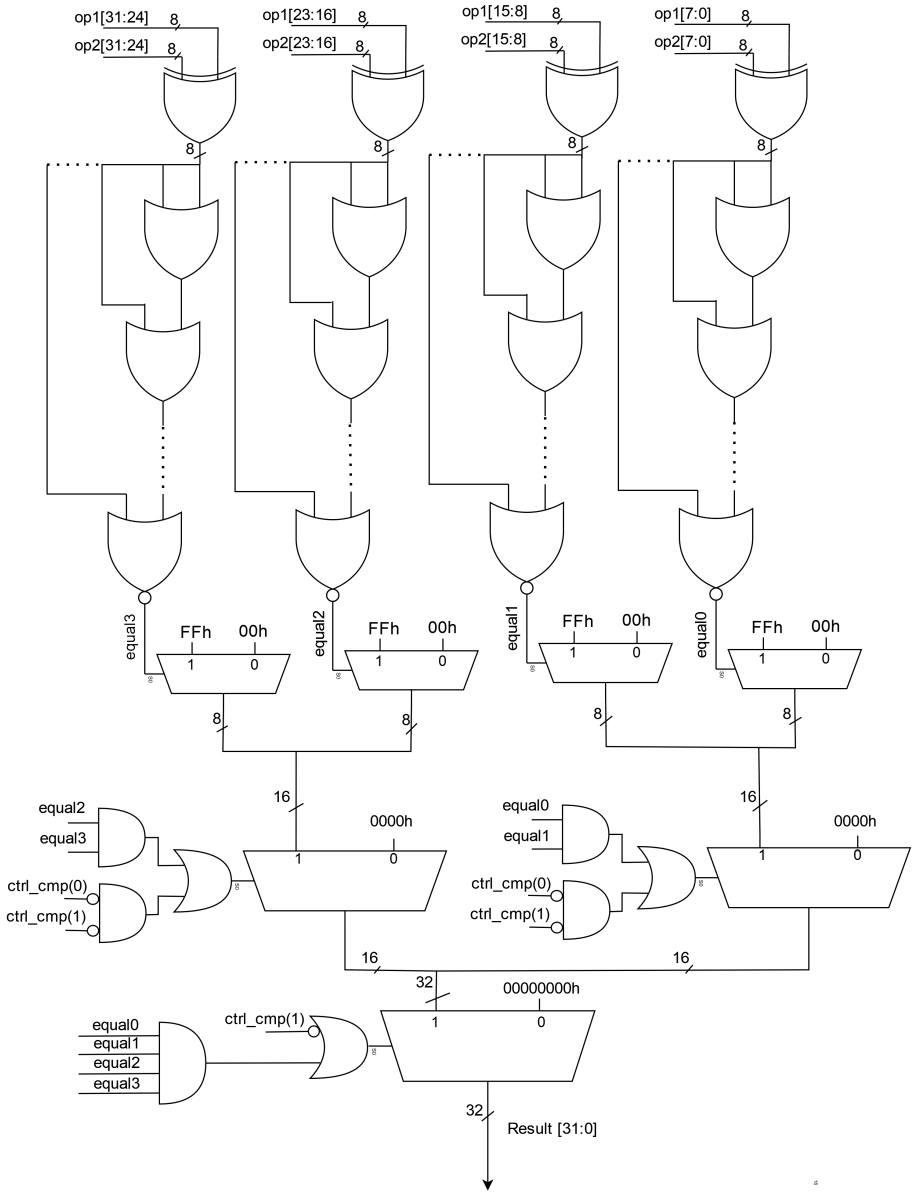

Figura 7: Componentele unității de egalitate (partea inferioară).

4.4 Unitatea PMADD

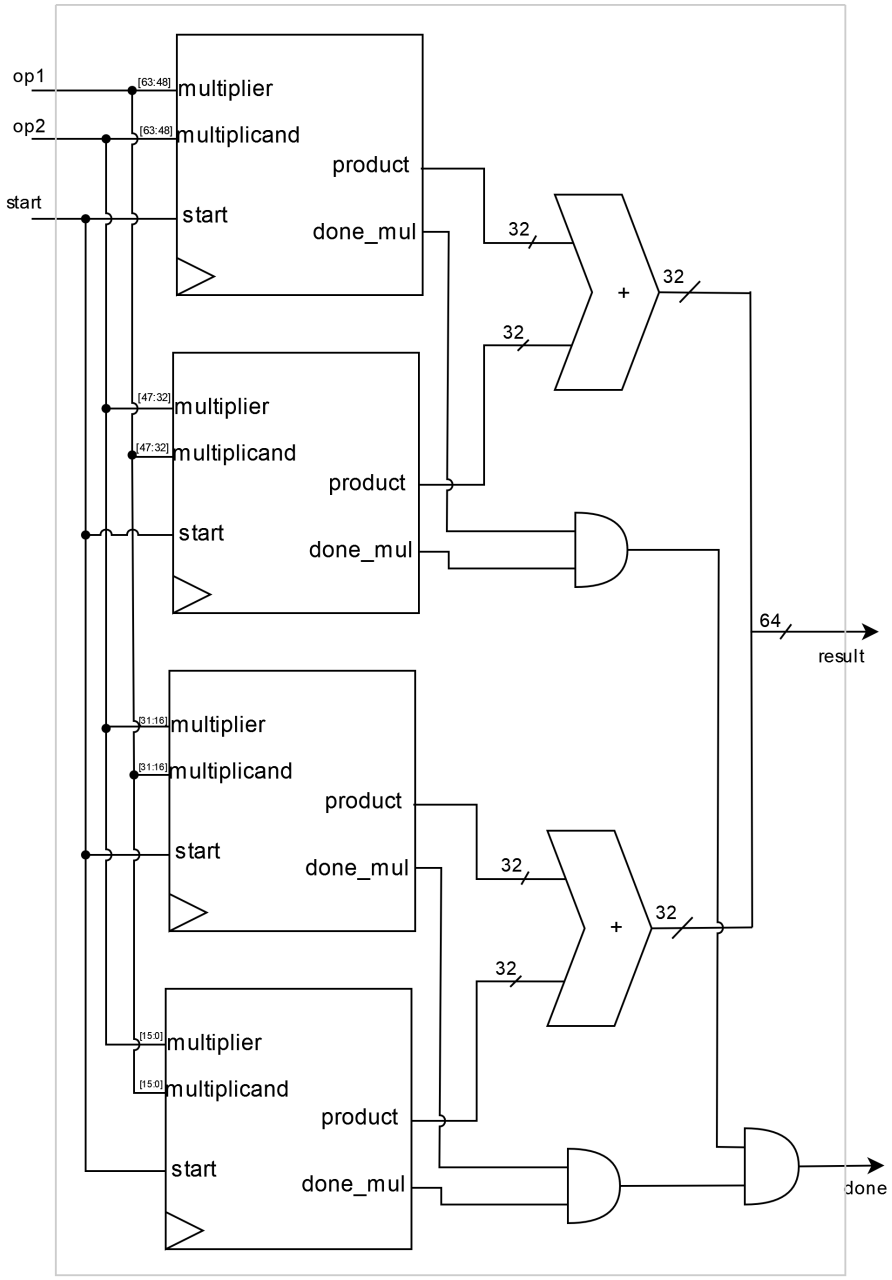

Figura 8: Componentele unității de efectuare a instrucțiunii PMADD.

Cum înmulțirea trebuie executată pe mai multe cicluri de ceas, unitatea are un semnal de clock, precum și o intrare de start care va fi comandată de mașina de stări, iar rezultatul va fi preluat doar atunci când se va activa ieșirea "done". Pentru implementarea acesteia, au fost necesare în principal două sumatoare pe 32 de biți, care adună rezultatele a câtor două înmulțitoare de operanzi pe 16 biți.

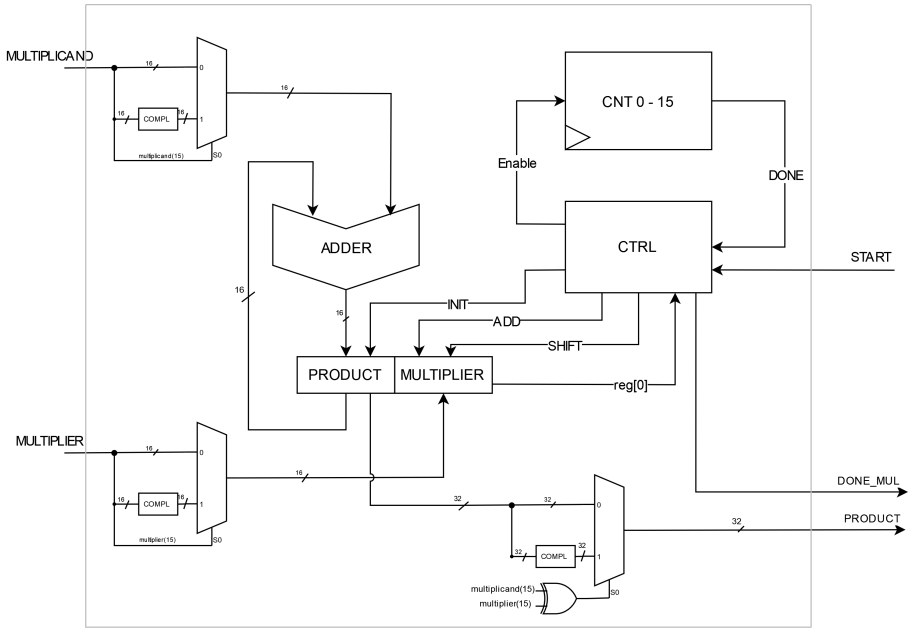

Figura 9: Inmulțitorul pe 16 biți.

Așa cum este menționat la capitolul de analiză, înmulțirea este implementată după modelul "Shift and Add". In loc de a se folosi două registre, unul pe 32 de biți pentru rezultat și unul de 16 pentru înmulțitor, vom folosi doar pe cel de 32 de biți. La semnalul de start, mini-unitatea de control va initializa registrul astfel: în jumatatea din stânga va fi zero, iar în cea din dreapta, înmulțitorul. Se parcurg biții înmulțitorului pe rând, utilizându-se cel mai din dreapta bit al registrului. Dacă este zero, vom face doar o deplasare la dreapta, iar dacă este 1, vom adăuga deînmulțitul rezultatului din jumătatea din stânga a registrului, după care vom face deplasarea la dreapta. De fiecare dată când se face această deplasare, avem un contor care se incrementează, și când ajunge la valoarea 16, se activează semnalul de terminare (done_mul). După fiecare deplasare, se verifică acest semnal, iar dacă nu este încă activat, se continuă verificarea biților înmulțitorului. La final, rezultatul va fi regăsit în registru.

4.5 Unitatea de deplasare

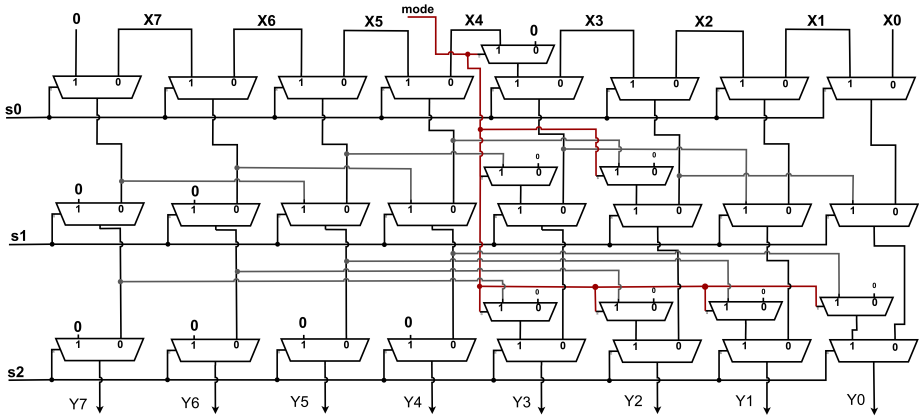

Figura 10: Exemplu de circuit de deplasare.

Deoarece deplasarea cu circuite combinaționale (multiplexoare) necesită un număr mare de componente, am inclus un exemplu restrâns, de deplasare al unui număr care poate fi interpretat fie pe 8 biți, fie ca și două numere pe 4 biți. Va fi compus din 3 nivele, primul deplasând cu o poziție, al doilea cu 2 poziții, iar al treilea cu 4 poziții (deci puteri ale lui 2). Astfel, selecțiile multiplexoarelor vor forma exact numărul de poziții cu care se face deplasarea.

Partea care permite interpretarea numărului ca 2 numere diferite sunt multiplexoarele care aleg dacă se salvează sau nu biții deplasati precedent. Semnalul mode ne permite acest lucru astfel: când este zero, nu vom păstra bitul deplasat de către jumatatea din stânga, si vom pune zero logic, altfel alegem ca intrare bitul shiftat (pentru interpretarea numarului pe 8 biti). Numărul de multiplexoare adăugate suplimentar pe fiecare nivel este determinat de numărul de poziții cu care se deplasează pe fiecare nivel, menționate mai sus.

Această logică am folosit-o și am extins-o pentru deplasarea numerelor de tip word, dword si qword, folosind 3 semnale similare cu mode care decid ce tip de deplasare se execută.

4.6 Unitatea de despachetare

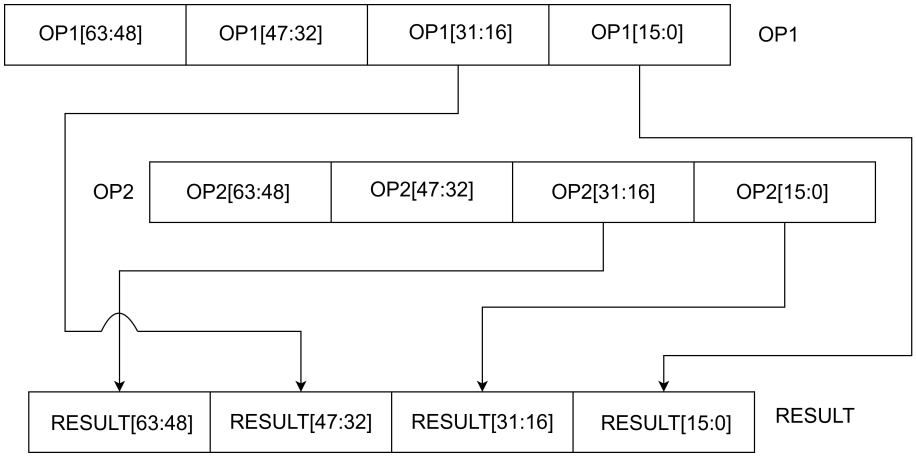

Figura 11: Unitatea de despachetare.

5 Implementare

5.1 Blocul de regiștri

Semnal	**Tip**	**Descriere**
addr1	Intrare	Adresa de la care se citește primul operand, și la care se face scrierea
addr2	Intrare	Adresa de la care se citește al doilea operand
clk	Intrare	Semnalul de ceas
WE	Intrare	Write Enable, când este 1 logic se face scrierea
data_in1	Intrare	Datele ce vor fi scrise în registrul de la addr1
data_out1	Ieșire	Primul operand, citit de la addr1
data_out2	Ieșire	Al doilea operand, citit de la addr2

Tabela 3: Intrările și ieșirile blocului de regiștri

Datele vor fi stocate într-un array de 8 elemente de tip std_logic_vector(63 downto 0), care a fost definit ca și un tip nou de date, denumit MMX_type. Am definit un semnal de acest tip, MMX_reg, care a fost inițializat cu niște date pentru testarea corectitudinii.

Scrierea se face într-un proces, pe front ascendent de ceas, atunci cand semnalul WE = 1. Citirea se face întotdeauna, fiind definită în afara procesului.

5.2 Memoria de date

Semnal	Tip	Descriere
addr1	Intrare	Adresa de la care se citește operandul imediat
data_out1	Ieșire	Operandul,citit de la addr1

Tabela 4: Intrările și ieșirile memoriei de date

Similar ca blocul de regiștri, memoria de date este un tablou de vectori STD LOGIC. Este implementat ca un ROM asincron, dar putea fi implementat și ca un RAM deoarece pe placa FPGA, după oprirea alimentării, datele se pierd oricum.

5.3 Unitatea de control (Mașina de stări)

Semnal	Tip	Descriere
opcode	Intrare	Tipul de instrucțiune executată
start	Intrare	Activarea acestuia înseamnă începerea exectuției
clk	Intrare	Semnalul de ceas
done_mul	Intrare	Activat atunci când s-a terminat înmulțirea
sel	Ieșire	Alege care rezultat va fi preluat din ALU
ctrl_add	Ieșire	Comandă ce tip de adunare se face (pe byte sau pe word)
ctrl_shift	Ieșire	Comandă ce tip de deplasare se face (pe word, dword sau qword)
ctrl_cmp	Ieșire	Comandă ce fel de comparație se face (pe byte, word sau dword)
start_mul	Ieșire	Comandă începerea înmulțirii
WE	Ieșire	Comandă scrierea în blocul de regiștrii
done	Ieșire	Activat doar în starea inițială

Tabela 5: Intrările și ieșirile unității de control

In arhitectură am declarat două semnale, unul care să conțină starea curentă, iar celălalt, starea următoare. Stările au fost codificate pe 4 biți.

Implementarea se face cu două procese. Primul face ca starea curentă să devină starea următoare pe front ascendent de ceas. Al doilea se declanșează la modificarea stării curente sau a semnalelor de intrare (start, opcode, done_mul) și decide în funcție de starea curentă și uneori de anumite intrări care va fi starea următoare. Totodată, la începutul acestui proces, toate ieșirile se dezactivează, urmând ca în funcție de starea curentă, să se activeze cele corespunzătoare.

5.4 Unitatea de execuție (ALU)

5.4.1 Poarta ANDN

Semnal	Tip	Descriere
a	Intrare	Primul operand
b	Intrare	Al doilea operand
y	Ieșire	Rezultatul

Tabela 6: Intrările și ieșirile porții ANDN

Implementarea este simplă, primindu-se două intrări pe 64 de biți și obținerea unui rezultat tot pe 64 de biți. Arhitectura este de tip flux de date.

5.4.2 Unitatea de adunare

Semnal	Tip	Descriere
a	Intrare	Primul operand
b	Intrare	Al doilea operand
cin	Intrare	Carry In
cout	Ieșire	Carry Out
overflow	Ieșire	Bitul care spune dacă s-a depășit domeniul de reprezentare
y	Ieșire	Rezultatul adunării

Tabela 7: Intrările și ieșirile sumatorului pe 8 biți

Componente Pentru implementare, am folosit ca și componente 4 sumatoare complete pe 8 biți, obținute prin cascadare de sumatoare complete pe un bit. Aceste sumatoare au în plus o ieșire de overflow, care se obține prin operația sau-exclusiv între transporturile de ieșire (carry out) ale ultimelor două sumatoare pe un bit din structura acestora.

Semnal	Tip	Descriere
op1	Intrare	Primul operand
op2	Intrare	Al doilea operand
ctrl_add	Intrare	Venit de la UC, 1 pentru PADDSW, 0 pentru PADDSB)
result	Ieșire	Rezultatul adunării

Tabela 8: Intrările și ieșirile unității de adunare

Implementarea adunării

Sumatoarele au fost instațiate în arhitectură și mapate. Acestea sunt legate câte două prin semnalele de transport, trecute prin porți și cu bitul ctrl_add. Când acesta este zero, sumatoarele primesc transport de intrare zero, și rezultatul este interpretat pe octeți.

Limitele se calculează întotdeauna, dar acestea sunt alese doar dacă a avut loc overflow și în funcție de semnalul de control. Dacă rezultatul unei adunări a fost pozitiv, limita trebuie să fie cea inferioară, 80h sau 8000h, deoarece s-ar fi adunat două numere negative și s-ar obținut unul pozitiv, și viceversa. Acest lucru este implementat cu multiplexoare, definite direct în arhitectură cu instrucțiunea when. La final, rezultatul se obține prin concatenarea rezultatelor finale.

5.4.3 Unitatea de deplasare

Semnal	**Tip**	**Descriere**
operand	Intrare	Primul operand (care trebuie deplasat)
amount	Intrare	Al doilea operand (cât trebuie deplasat)
mode	Intrare	000 - PSRLQ, 011 - PSRLW, 110 - PSRLD)
result	Ieșire	Rezultatul deplasării

Tabela 9: Intrările și ieșirile unității de deplasare

Implementarea se realizează cu multiplexoare. Maximul de poziții shiftate fiind 63, vor fi necesare 6 nivele. Pentru fiecare nivel, am folosit instrucțiunea generate pentru a lega multiplexoarele care nu au intrarea dependentă de semnalul mode, iar celorlalte le-am făcut maparea separat.

Indexarea nivelelor este de la zero, fiecare va fi separat în patru părți, iar nivelul i face deplasare cu 2^i poziții. Fiind 4 părți, iar cea din stânga primind mereu zero ca intrare, vom avea nevoie de 3 * 2^i multiplexoare care să decidă păstrarea bitului shiftat anterior sau nu. Multiplexoarele din mijloc vor fi comandate de semnalul mode(1), iar celelalte de mode(0). Acest lucru se menține până la nivelul 3. La nivelul 4 deja nu se mai pot deplasa cuvintele, ci doar dublu cuvintele, deci vor fi necesare doar 2^4 multiplexoare, comandate de semnalul mode(2). Nivelul 5 nu necesită modificări.

Dacă semnalul mode = 000, atunci toate multiplexoarele salvează bitul anterior, deci se deplasează un qword. Dacă mode = 011, toate multiplexoarele pun zero pe post de bit anterior, în afară de cel de pe nivelul 4, deci se deplasează

doar cuvinte. Dacă mode = 110, se separă qword-ul în 2, și se face deplasarea pe dword.

Am definit și 3 semnale auxiliare, amount63_6, amount63_5, amount63_4, care se activează atunci când cel puțin un bit din porțiunea corespunzătoare a semnalului amount este 1. In funcție de acestea și de semnalul mode se decide dacă shiftarea se face cu mai multe poziții decât este permis, și se pune zero pe rezultatul final. In caz contrar, se trimite rezultatul de pe ultimul nivel de multiplexoare.

5.4.4 Unitatea de egalitate

Semnal	**Tip**	**Descriere**
op1	Intrare	Primul operand
op2	Intrare	Al doilea operand
ctrl_compare	Intrare	00 - PCMPEQB, 01 - PCMPEQW, 10 - PCMPEQD)
result	Ieșire	Rezultatul comparației

Tabela 10: Intrările și ieșirile unității de egalitate

Pentru fiecare pereche de octeți am făcut sau exclusiv, și am salvat rezultatul într-un vector de bytes, byte_xor. După aceea, pentru fiecare byte rezultat am făcut sau logic pe biții acestuia, pentru a vedea dacă aceștia au fost egali, iar rezultatul negat a fost salvat într-un vector de STD LOGIC byte_equal_bool. Astfel, byte_equal_bool(i) = 1 dacă perechea i de octeți a fost egală.

```
   loop1: for i in 7 downto 0 generate
         byte_xor(i)<=op1((i+1)*8-1 downto i*8) xor
↪   op2((i+1)*8-1 downto i*8);
   end generate;
   loop2_1: for i in 7 downto 0 generate
         byte_equal_bool(i)<=not(byte_xor(i)(7) or
↪   byte_xor(i)(6) or byte_xor(i)(5) or byte_xor(i)(4) or
↪   byte_xor(i)(3) or byte_xor(i)(2) or byte_xor(i)(1) or
↪   byte_xor(i)(0));
   end generate;
```

Rezultatele sunt trimise apoi cu multiplexoare, logica fiind explicată la partea de design, iar acestea au fost implementate folosind instrucțiunea when.

5.4.5 Unitatea PMADD

Componente

Sumator complet generic. Are intrările și ieșirile unui sumator normal (a, b, cin, cout, y), dar mai are și un parametru n generic, care definește pe câți biți se face adunarea. Este implementat folosind un semnal auxiliar q pe n+1 biți pentru a salva transporturile, și instrucțiunea generate, pentru generarea și maparea sumatoarelor complete pe un bit.

Generator de complement. Implementarea acestuia se face tot cu un parametru generic n, care îmi spune pe câți biți este intrarea (x) și implicit rezultatul (s). Intrarea se neagă și intră împreună cu zero într-un sumator pe n biți al cărui cin este mapat valorii 1 logic. Ieșirea sumatorului este complementul numărului.

Inmulțitorul pe 16 biți. Este alcătuit din 2 părți, partea de execuție și cea de control.

Partea de execuție.

- numărător pe 16 biți pe front crescător cu semnal de enable (ce) și reset sincron mapat pe semnalul init. Conține numărul de iterații efectuate, și când se ajunge la 16, are rol de a activa semnalul de terminare a înmulțirii (done).
- registru de deplasare la dreapta cu load sincron. Acesta va conține rezultatele intermediare și apoi pe cel final. Pe intrarea de scriere va avea un semnal d_reg, care vine de la un multiplexor (implementat cu instrucțiunea when). Când semnalul init de la partea de comandă va fi 1 logic, vom incărca în jumătatea superioară zero și în cea inferioară multiplicandul. Când se va scrie un rezultat parțial, init va fi zero, iar d_reg va fi egal cu rezultatul parțial venit de la sumator, concatenat cu jumătatea inferioară a conținutului registrului, astfel realizând scrierea în prima parte și menținerea celei de-a doua părți.
- sumator generic mapat pe 16 biți, pentru a calcula rezultatele parțiale ce vor fi scrise în jumătatea superioară a registrului.

Partea de comandă. Primește semnalul start pentru începerea înmulțirii (init = 1), semnalul done de la partea de execuție, care marchează finalul înmulțirii, precum și ultimul bit al produsului (p0), pentru a decide dacă se face scrierea (write = 1) în prod_reg înainte de shiftare (shift = 1). Totodată comandă numărarea (ce = 1) pentru a ține cont de iterația la care am ajuns. Este implementată cu două procese, unul pentru asignarea stării următoare stării curente pe front crescător, și unul pentru deciderea stării următoare în funcție de starea curentă și intrări (state, start, done, p0).

Pentru obținerea înmulțitorului, vom lega partea de execuție cu partea de comandă. Operanzii se vor decide în funcție de cel mai semnificativ bit al acestora; dacă este 1, înseamnă ca operandul este negativ, deci vom lua operandul trecut prin generatorul de complement (mapat pe 16 biți). Se va alege între rezultat și complementul acestuia (obținut folosind un generator de complement mapat pe 32 de biți), în funcție de cei mai semnificativi biți ai operanzilor. Multiplexorul care alege între aceste valori va avea ca și selecție operația sau-exclusiv dintre biții de semn ai operanzilor (dacă este zero, adică semnul lor este identic, punem valoarea actuală a rezultului, atfel, complementul).

Semnal	**Tip**	**Descriere**
multiplier	Intrare	Înmulțitorul
multiplicand	Intrare	Deînmulțitul
clk	Intrare	Semnalul de ceas
start	Intrare	Comandă începerea înmulțirii
done_mul	Ieșire	Indică sfârșitul înmulțirii
product	Ieșire	Produsul

Tabela 11: Intrările și ieșirile înmulțitorului

Intrările și ieșirile unității PMADD constau în cei doi operanzi furnizați ca intrare unității de execuție, precum si cu rezultatul instrucțiunii PMADDWD, pe 64 de bit, cărora li se adaugă semnalele de clock, start și done_mul, care vor fi legate fiecărui înmulțitor. Vor fi necesare 4 astfel de înmulțitoare, fiecare operând pe cuvinte, precum și două sumatoare mapate pe 32 de biți, pentru obținerea rezultatului exprimat în 2 dublu cuvinte. Semnalul done_mul se activează doar când toate semnalele de terminare a fiecărui înmulțitor sunt activate.

5.4.6 Unitatea de despachetare

Semnal	**Tip**	**Descriere**
op1	Intrare	Primul operand
op2	Intrare	Al doilea operand
result	Ieșire	Rezultatul

Tabela 12: Intrările și ieșirile unității de despachetare

Despachetarea a fost implementată folosind instrucțiunile de asignare din VHDL pe porțiuni din operanzi.

```
tmp(63 downto 48)<=op2(31 downto 16);
tmp(47 downto 32)<=op1(31 downto 16);
tmp(31 downto 16)<=op2(15 downto 0);
tmp(15 downto 0)<=op1(15 downto 0);
result<=tmp;
```

5.5 MMX

Semnal	Tip	Descriere
clk	Intrare	Semnalul de ceas
addr1	Intrare	Adresa primului operand
addr2	Intrare	Adresa celui de-al doilea operand
addr_imm	Intrare	Adresa imediatului
opcode	Intrare	Codul instrucțiunii
start	Intrare	Determină începerea execuției
op_mode	Intrare	Tipul operației
LED_done	Ieșire	Semnifică terminarea execuției
op1_out	Ieșire	Valoarea operandului de la addr1

Tabela 13: Intrările și ieșirile unității MMX

Implementarea acestuia se face prin legarea unității de comandă, unității de execuție, a blocului de regiștri și a memoriei de date. Mai este necesar un multiplexor, care selectează al doilea operand al intrucțiunii (când op_mode = 1 atunci acesta va fi imediatul, altfel se preia din blocul de regiștri).

6 Testare și validare

6.1 ANDN

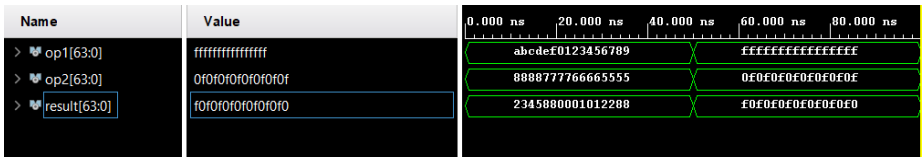

Figura 12: Testarea operației ANDN.

6.2 Adunare

Pentru adunare am dat valori care să cuprindă limitele superioare și inferioare, operanzi al căror rezultat să depașească limitele, adunare intre două numere negative, pozitive, sau unul negativ și celălalt pozitiv.

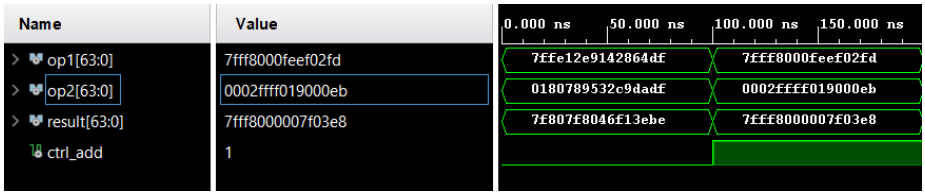

Figura 13: Testarea operației de adunare.

6.3 Deplasare

Pentru deplasare, am incercat, pentru fiecare operație, deplasarea la limite, deplasarea cu un multiplu de 4, deplasarea cu un număr care să nu fie multiplu de 4, deplasarea cu un număr care depășește limita.

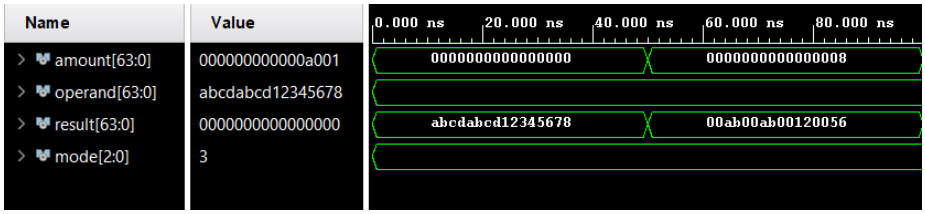

Figura 14: Testarea operației PSRLW (1).

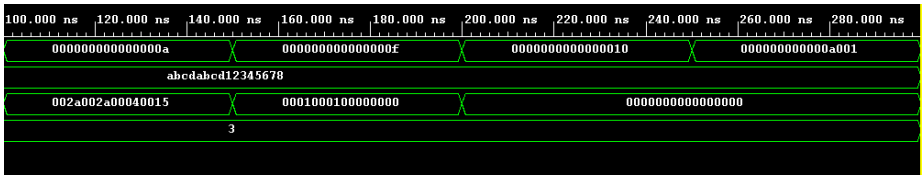

Figura 15: Testarea operației PSRLW (2).

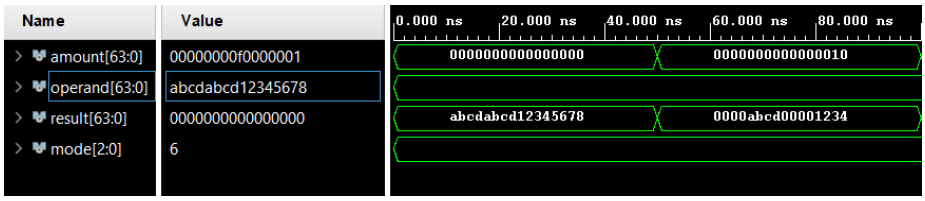

Figura 16: Testarea operației PSRLD (1).

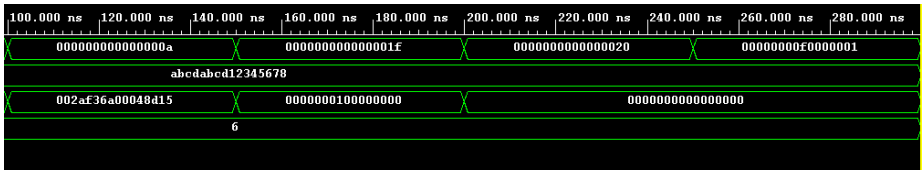

Figura 17: Testarea operației PSRLD (2).

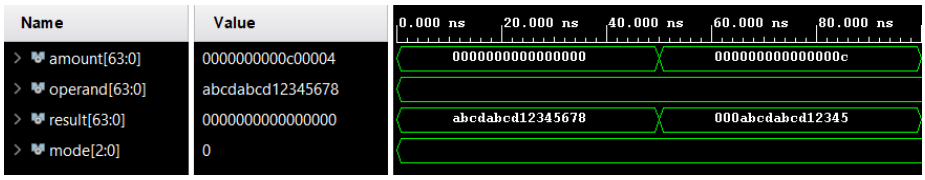

Figura 18: Testarea operației PSRLQ (1).

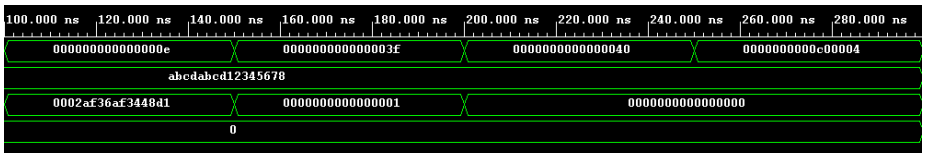

Figura 19: Testarea operației PSRLQ (2).

6.4 Comparare

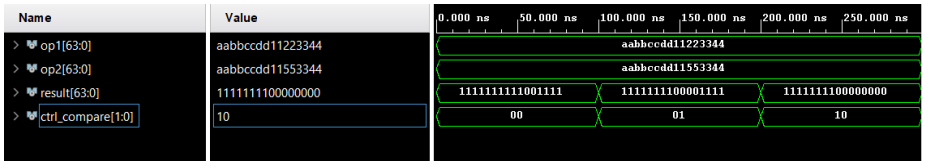

Figura 20: Testarea operației de verificare a egalității.

6.5 PMADD

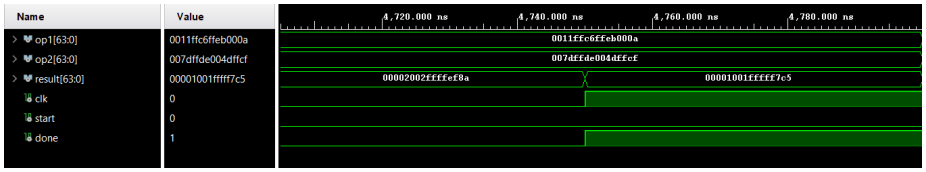

Figura 21: Testarea operației PMADDWD.

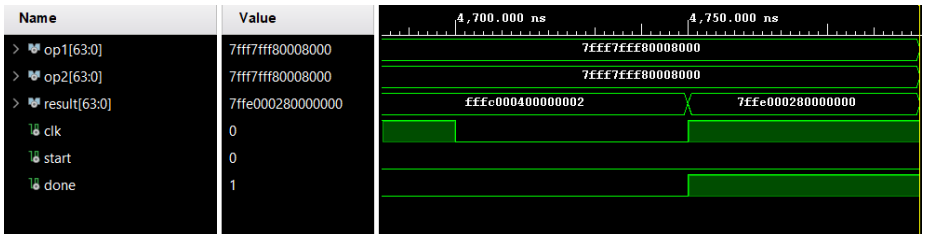

Figura 22: Testarea operației PMADDWD la limite.

6.6 Despachetare

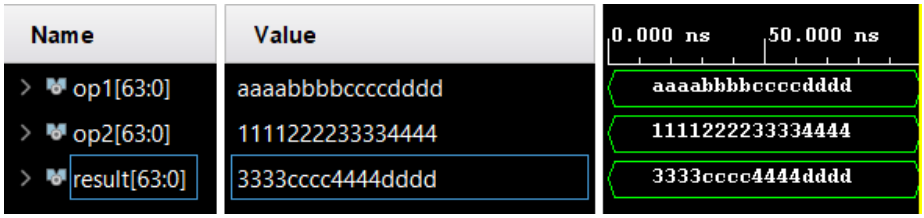

Figura 23: Testarea operației de despachetare.

6.7 Testarea unității MMX

La secțiunile anterioare am testat corectitudinea efectuării instrucțiunilor de către unitatea de execuție, iar la această secțiune vom testa corectitudinea efectuării și încărcarea rezultatului la registrul corespunzător. Deoarece acesta se stochează la prima adresă (addr1), am afișat operandul respectiv. Se va observa modificarea acestuia după ce semnalul LED_done se activează, semnificând terminarea execuției. Mai jos am inclus un tabel cu valorile existente în blocul de registre și în memorie, folosite pentru validare.

Adresa	Valoarea registrului	Valoarea din memorie
0	ABCD1234DCBA5678h	FF00FF0000FF00FFh
1	12345678ABCDEF00h	0000000000000004h
2	1111222233334444h	CCCCDDDDEEEEFFFFh
3	0011FFC6FFEB000Ah	007DFFDE004DFFCFh
4	AABBCCDD12345678h	AADDCCDD12345678h
5	7FFE12E9142864DFh	0180789532C9DADFh
6	7FFF8000FEEF02FDh	0002FFFF019000EBh
7	7FFF7FFF80008000h	7FFF7FFF80008000h

Tabela 14: Valorile regiștrilor și a memoriei, exprimate în hexazecimal.

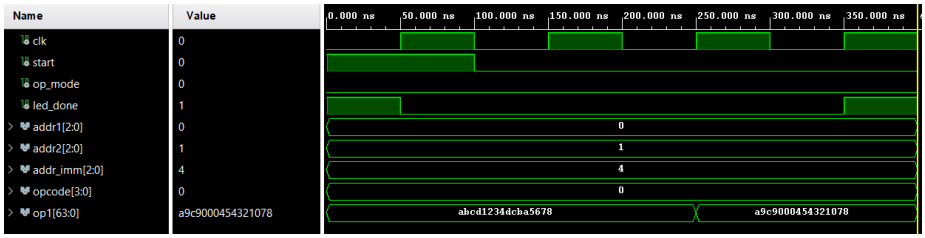

Figura 24: pandn reg[0], reg[1]

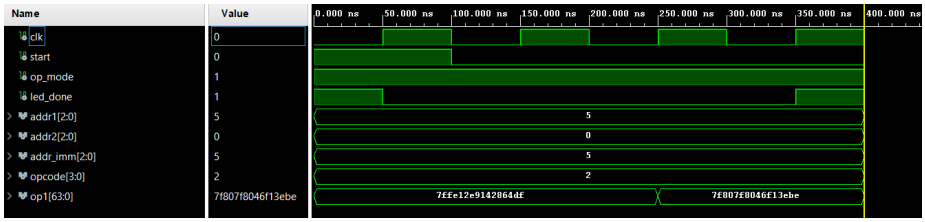

Figura 25: paddsb reg[5], mem[5]

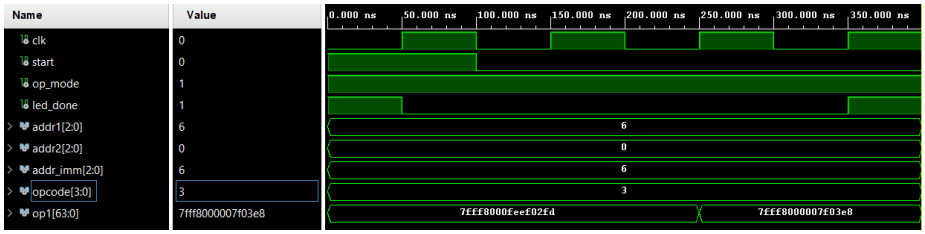

Figura 26: paddsw reg[6], mem[6]

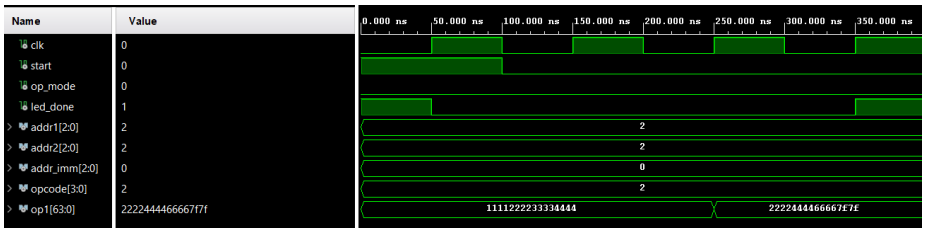

Figura 27: paddsb reg[2], reg[2]

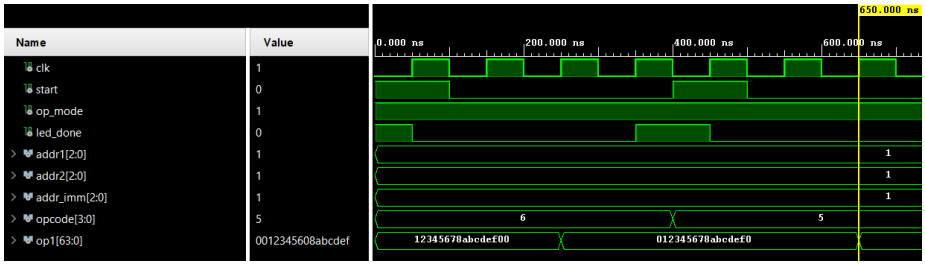

Figura 28: psrlq reg[1],mem[1]

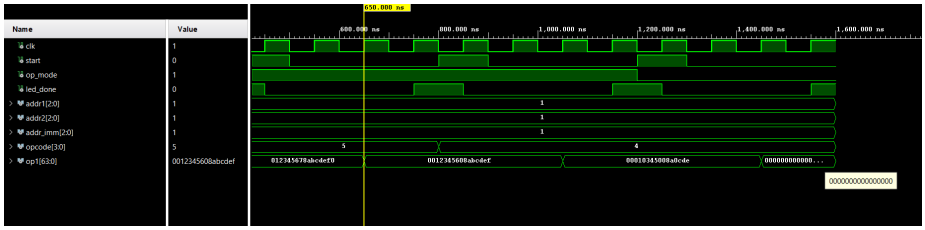

Figura 29: psrld reg[1], mem[1]; psrlw reg[1], mem[1]; psrlw reg[1], reg[1]

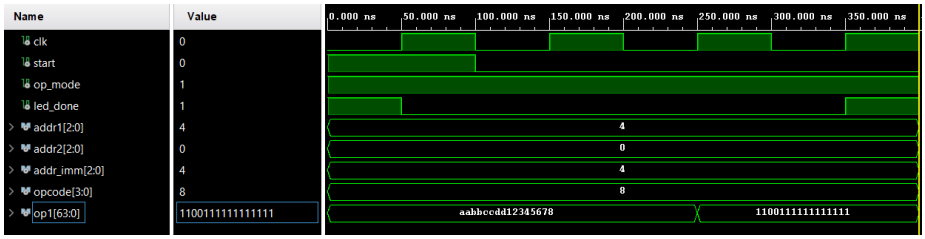

Figura 30: pcmpeqb reg[4], mem[4]

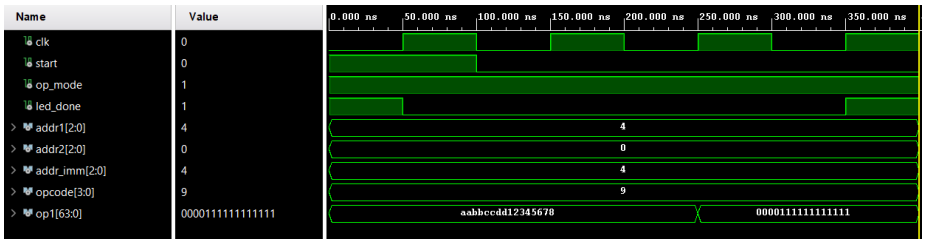

Figura 31: pcmpeqw reg[4], mem[4]

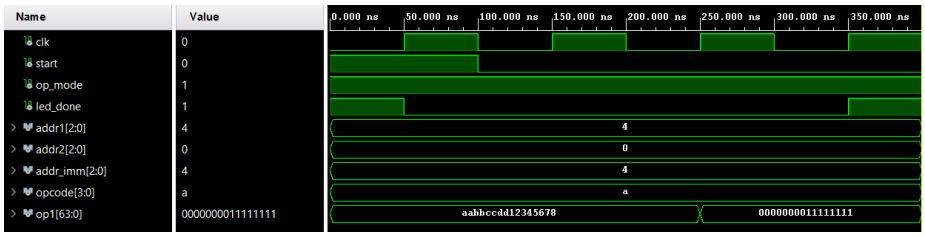

Figura 32: pcmpeqd reg[4], mem[4]

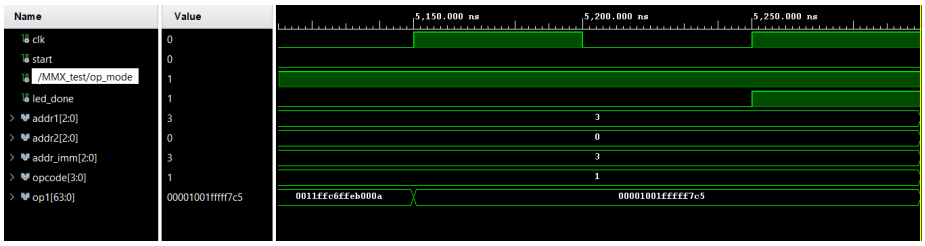

Figura 33: pmadd reg[3], mem[3]

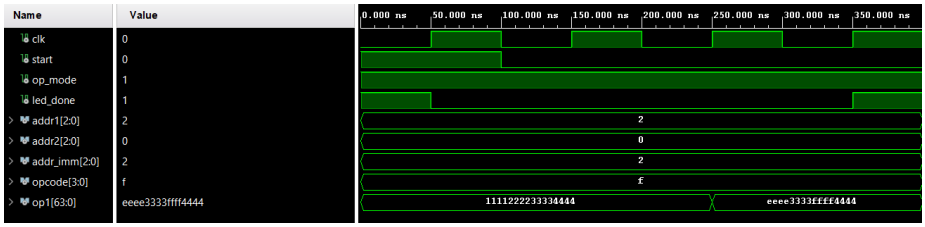

Figura 34: unpack reg[2], mem[2]

7 Concluzii

Scopul acestui proiect a fost implementarea celor 6 instrucțiuni alese, și a fost atins cu succes, funcționalitatea lor find reflectată atât în simulări cât și pe placa FPGA Basys 3, reușind respectarea planului de dezvoltare.

Cel mai mult mi-a plăcut să proiectez schemele logice pentru instrucțiuni, mai ales cele pentru adunare și comparare, și ulterior să le implementez. Acest proiect m-a ajutat să lucrez modular prin descrierea majoritar structurală a arhitecturilor, precum și să imi antrenez gândirea logică.

Cea mai complexă parte mi s-a părut partea de deplasare, deoarece în ciuda faptului că implementarea cu registre de deplasare ar fi fost o variantă mai ușoară, și pe care o aveam funțională deja, am vrut să o realizez combinațional, dar am reușit în final să găsesc logica corespunzătoare și să o descriu în cod.

Bibliografie

[1] Intel. *IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture*. 2005.

[2] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B, 2C 2D): Instruction Set Reference, A-Z*. 2022.