Proiectarea unei unități aritmetice MMX pe FPGA Nexys A7

Student: Tcaci Liviu

Proiect la Structura Sistemelor de Calcul

Universitatea Tehnică din Cluj-Napoca

6 Decembrie 2024

Cuprins

1 Introducere

1.1 Context

Evoluţia rapidă a tehnologiei şi creşterea continuă a volumului de date procesate în aplicaţii moderne – precum procesarea multimedia, analiza de semnal, învăţarea automată şi inteligenţa artificială – au generat o cerere sporită pentru soluţii hardware care pot asigura rate de calcul ridicate şi latenţe reduse. În acest context, arhitecturile hardware paralele, în special cele bazate pe principiul *Single Instruction, Multiple Data* (SIMD), au devenit fundamentale. Ele permit executarea aceleiaşi instrucţiuni simultan pe seturi multiple de date, exploatând masiv paralelismul la nivel de date.

Arhitectura **MMX** (MultiMedia eXtension), introdusă iniţial de Intel, extinde setul de instrucţiuni al unităţii centrale de prelucrare (CPU) cu instrucţiuni SIMD ce utilizează registre pe 64 de biţi. Prin intermediul acestor extensii, sunt posibile operaţii vectoriale eficiente pe elemente de 8, 16 sau 32 de biţi, optimizând task-urile legate de procesarea imaginilor, compresia audio-video şi alte operaţii cu volum mare de date. Astfel, MMX contribuie la îmbunătăţirea performanţei sistemelor moderne, reducând sarcina software-ului şi accelerând calculele intensive.

Proiectul de faţă îşi propune implementarea, într-o manieră modulară, a unei unităţi aritmetice MMX utilizând limbajul de descriere hardware *VHDL*, rulată pe platforma **FPGA Nexys A7**. Această abordare nu doar simulează funcţionalitatea instrucţiunilor SIMD specifice arhitecturii MMX, ci oferă şi un mediu propice pentru analiză, experimentare şi înţelegerea aprofundată a conceptelor hardware paralele. Alegerea FPGA-ului ca mediu de implementare permite testarea reală a designului, oferind o perspectivă practică şi educaţională asupra modului în care arhitecturile SIMD pot fi transpuse în soluţii hardware eficiente.

1.2 Obiective şi Motivaţii

Motivaţii: Creşterea exponenţială a volumului de date procesate în aplicaţii complexe – de la procesarea imaginilor şi compresia audio-video, până la algoritmi de învăţare automată şi analiză de semnal – a evidenţiat necesitatea arhitecturilor hardware capabile să exploateze masiv paralelismul. Arhitecturile SIMD, precum cea introdusă de extensiile MMX, permit executarea aceleiaşi instrucţiuni simultan asupra mai multor elemente de date, reducând semnificativ timpul de procesare comparativ cu abordările software tradiţionale.

Implementarea unei unităţi aritmetice MMX pe un FPGA (în cazul de faţă, Nexys A7) este motivată de dorinţa de a obţine un control fin asupra arhitecturii, pentru a studia în detaliu modul în care instrucţiunile SIMD pot fi transpuse în circuite logice dedicate. Această abordare oferă posibilitatea optimizării latenţei şi consumului energetic, precum şi o mai bună scalabilitate. În plus, proiectul are o valoare didactică importantă, facilitând înţelegerea principiilor SIMD, proiectarea modulară şi fluxul complet de dezvoltare hardware (de la descrierea în VHDL, la implementarea şi validarea pe FPGA).

Obiective: Proiectul îşi propune să proiecteze, implementeze şi valideze o unitate aritmetică compatibilă cu arhitectura MMX, care să asigure un set reprezentativ de operaţii SIMD şi să ofere o platformă solidă pentru teste şi evaluări. Principalele obiective sunt:

- **Implementarea operaţiilor fundamentale SIMD:** Adunări şi scăderi cu wrap-around (PADD, PSUB) pentru 8, 16 şi 32 biţi, înmulţirea cu reţinerea parţii inferioare sau superioare (PMUL), precum şi operaţii cu saturare (PADDS, PADDUS, PSUBS) pentru a preveni overflow-ul atât în cazul datelor semnate, cât şi nesemnate.

- **Integrare modulară:** Utilizarea unui design modular facilitează testarea, depanarea şi optimizarea. Componente periferice precum *MemoryUnit*, *BitSelector* şi *SSD* asigură gestionarea intrărilor şi ieşirilor, selecţia segmentelor relevante şi afişarea lizibilă a rezultatelor.
- **Flexibilitate şi scalabilitate:** Asigurarea suportului pentru date de 8, 16 şi 32 de biţi conferă o versatilitate crescută a designului, permiţând evaluarea performanţei în diferite scenarii.
- **Validare prin simulări şi testbench-uri:** Corectitudinea şi eficienţa soluţiei vor fi confirmate prin testări extinse în mediul *Xilinx Vivado*, simulări funcţionale şi rularea designului pe FPGA Nexys A7.

Prin atingerea acestor obiective, proiectul îşi propune să demonstreze atât potenţialul arhitecturilor SIMD în îmbunătăţirea performanţei sistemelor de calcul, cât şi valoarea adăugată a implementării hardware dedicate, subliniind avantajele şi provocările dezvoltării de soluţii paralele la nivel de circuit.

1.3 Contribuții şi Relevanţă

Prezentul proiect aduce o serie de contribuţii semnificative atât din perspectivă tehnică, cât şi didactică, evidenţiind beneficiile implementării hardware a arhitecturilor SIMD:

- **Implementare completă a instrucţiunilor MMX fundamentale:** Proiectul prezintă o unitate aritmetică care integrează operaţii SIMD esenţiale (PADD, PSUB, PMUL, precum şi variantele cu saturare). Prin abordarea modulară, fiecare operaţie este implementată sub formă de bloc hardware dedicat, facilitând astfel înţelegerea şi testarea individuală a fiecărei funcţionalităţi.
- **Abordare modulară şi scalabilă:** Separarea logicii în blocuri funcţionale independente simplifică depanarea, optimizarea şi potenţiala extindere ulterioară. Designul modular permite includerea rapidă a unor noi instrucţiuni SIMD sau adaptarea la alte dimensiuni de date, fără a perturba întregul sistem.
- **Validare pe platforma FPGA Nexys A7:** Integrarea şi testarea pe o platformă hardware reală au asigurat o evaluare concretă a performanţelor. Comparativ cu implementările software, soluţia hardware oferă latenţe reduse, eficienţă energetică sporită şi posibilitatea de a opera în timp real pe fluxuri mari de date.
- **Relevanţă didactică şi de cercetare:** Rezultatele şi metodologia pot fi folosite ca suport educaţional în cadrul cursurilor de arhitectură a calculatoarelor, sisteme paralele sau proiectare hardware. De asemenea, proiectul poate servi drept punct de plecare pentru cercetări viitoare, cum ar fi extinderea cu noi seturi de instrucţiuni, integrarea altor extensii SIMD sau optimizări suplimentare pentru aplicaţii specifice.

Prin aceste contribuţii, proiectul subliniază relevanţa arhitecturilor SIMD implementate la nivel hardware în contextul cerinţelor tot mai stringente de performanţă şi paralelism. Se demonstrează astfel că o soluţie bine optimizată şi testată pe FPGA poate aduce beneficii concrete, atât în domeniul aplicaţiilor practice, cât şi ca material de studiu şi cercetare.

2 Bibliographic Research

2.1 Contextul Arhitecturilor SIMD

Arhitecturile *Single Instruction, Multiple Data* (SIMD) reprezintă o abordare arhitecturală prin care un singur flux de instrucţiuni este aplicat asupra unui vector de date, procesând simultan elemente multiple într-un singur ciclu de ceas. Această paradigmă s-a dezvoltat ca răspuns la nevoia din ce în ce mai mare de a prelucra rapid volume mari de date, întâlnite frecvent în domenii precum grafica computaţională, procesarea audio-video, analiza semnalelor, criptografie sau în aplicaţiile ce implică operaţii matematice repetitive (de ex. filtre digitale, convoluţii sau transformate rapide Fourier).

Principiul de bază al arhitecturilor SIMD se fundamentează pe paralelizarea la nivel de date. Dacă o aplicaţie necesită executarea aceleiaşi operaţii (de exemplu, adunare, scădere, înmulţire) pe mai multe elemente independent, atunci în loc să se proceseze fiecare element secvenţial, o arhitectură SIMD permite procesarea lor în paralel. În acest mod, puterea de calcul se scalaează odată cu mărirea lăţimii registrelor vectoriale şi cu numărul de elemente prelucrate simultan.

Avantajele acestei abordări sunt multiple:

- **Creşterea performanţei:** Prin prelucrarea paralelă a datelor, SIMD reduce semnificativ timpul de execuţie pentru aplicaţiile masiv paralele.
- **Eficienţa energetică:** Operând pe vectori de date într-un mod concentrat, se diminuează numărul de instrucţiuni individuale, reducând consumul energetic pe instrucţie realizată.
- **Scalabilitate:** Lăţimea registrelor vectoriale poate fi extinsă pentru a suporta mai multe elemente, permiţând o actualizare relativ facilă a arhitecturii atunci când cerinţele aplicaţiilor cresc.

De-a lungul timpului, mai multe extensii şi arhitecturi SIMD au apărut sau s-au îmbunătăţit. De la instrucţiunile MMX şi SSE prezente în procesoarele x86, la AVX, NEON (pe arhitecturi ARM) sau AltiVec (PowerPC), evoluţia a continuat să furnizeze modalităţi mai eficiente de a utiliza resursele hardware pentru prelucrarea vectorială. Acest context istoric şi tehnologic subliniază relevanţa abordării SIMD ca punct de plecare pentru proiectul nostru, care îşi propune să implementeze un subset de instrucţiuni aritmetice şi logice MMX, demonstrând aplicabilitatea concretă a conceptului în mediul FPGA.

2.2 Arhitectura MMX și Rolul său Istoric

Arhitectura MMX (*MultiMedia eXtension*), introdusă de Intel la jumătatea anilor 1990, reprezintă una dintre primele extensii SIMD disponibile pe scară largă pentru procesoarele x86. Apariția MMX a avut un rol crucial în popularizarea și standardizarea conceptului de prelucrare vectorială pe sisteme desktop, marcând un punct de cotitură în evoluția procesoarelor generale către capacități sporite de prelucrare paralelă a datelor.

Contextul istoric:

- **Intel Pentium cu MMX (1997):** Prima implementare a setului de instrucțiuni MMX a debutat pe procesoarele Intel Pentium, inițial orientate către aplicații multimedia. Prin introducerea a opt registre de 64 de biți (denumite `MM0-MM7`), arhitectura permitea încărcarea, stocarea și procesarea simultană a mai multor elemente de date numerice (e.g., octeți, cuvinte, doubleword-uri).
- **Focalizare pe date întregi:** Instrucțiunile MMX au fost concepute pentru a accelera operații pe date întregi nesemnate sau semnate, cu precădere în domenii precum compresia

video, audio, procesarea imaginilor și a graficii 2D. Într-o perioadă în care streaming-ul multimedia și jocurile 3D începeau să crească în popularitate, MMX a furnizat o soluție practică și accesibilă pentru a obține performanță îmbunătățită fără creșteri majore de frecvență.

- **Portabilitate și standardizare:** Fiind integrată direct în arhitectura x86, MMX a fost adoptată rapid de industrie, având avantajul retrocompatibilității cu instrucțiunile existente. Astfel, dezvoltatorii de software puteau optimiza anumite segmente critice prin utilizarea instrucțiunilor MMX, fără a schimba platforma hardware.

Impact și moștenire: Deși ulterior au apărut extensii SIMD mai avansate (precum SSE, SSE2, AVX), MMX a rămas un reper istoric important. A demonstrat că o extensie SIMD integrată în arhitectura unui procesor general-purpose poate aduce beneficii notabile în aplicații larg răspândite, nu doar în sistemele specializate. Astfel, MMX a deschis calea pentru evoluții ulterioare, stabilind principiile de bază ale prelucrării vectoriale pe procesoare x86 și inspirând abordări similare în alte arhitecturi (ARM NEON, PowerPC AltiVec, etc.).

În contextul proiectului de față, implementarea unui subset de instrucțiuni MMX pe FPGA reprezintă nu doar o demonstrație tehnică a conceptelor SIMD, ci și o formă de a omagia rolul arhitecturii MMX în istoria dezvoltării procesoarelor cu capacități vectoriale.

2.3 Setul de Instrucțiuni Implementat în Proiect

Proiectul de față își propune să demonstreze conceptul de paralelism SIMD prin intermediul unui subset reprezentativ de instrucțiuni inspirate din arhitectura MMX. Aceste instrucțiuni au fost selectate pentru a evidenția varietatea de operații care pot fi aplicate simultan pe vectori de date, precum adunări, scăderi, înmulțiri și operații cu saturare. De asemenea, alegerea instrucțiunilor reflectă cerințele diverselor domenii de aplicație (procesarea imaginilor, semnalelor audio-video sau prelucrarea vectorilor numerici cu potențial de overflow).

Principalele instrucțiuni implementate:

1. **PADD (Packed Add)**: Adună elementele corespunzătoare din doi vectori MMX pe 64 de biți. Dimensiunile elementelor pot fi:
   - `PADDB` – adunare pe octeți (8 biți) cu wrap-around;
   - `PADDW` – adunare pe word-uri (16 biți) cu wrap-around;
   - `PADDD` – adunare pe doubleword-uri (32 biți) cu wrap-around.
2. **PSUB (Packed Subtract)**: Scade elementele corespunzătoare din doi vectori, cu logica de wrap-around similară.
   - `PSUBB` – scădere pe octeți (8 biți) cu wrap-around;
   - `PSUBW` – scădere pe word-uri (16 biți) cu wrap-around;
   - `PSUBD` – scădere pe doubleword-uri (32 biți) cu wrap-around.
3. **PMULL (Packed Multiply Low—High Word)**: Înmulțește elementele pe 16 biți (word) ale vectorilor MMX și reține doar partea inferioară (low word) sau partea superioară (high word) a rezultatului. Aceasta este suficientă pentru numeroase aplicații, precum calculul filtrelor digitale cu precizie redusă.
4. **PADDS, PADDUS, PSUBS (Packed Add/Sub with Saturation)**: Aceste instrucțiuni asigură că rezultatele nu depășesc limitele intervalului reprezentabil, prevenind overflow-ul prin saturare.
   - `PADDSB` – adunare cu saturare pentru octeți semnați (8 biți);
   - `PADDUSB` – adunare cu saturare pentru octeți nesemnați (8 biți);

- `PSUBSB` – scădere cu saturare pentru octeți semnați (8 biți).

Instrucțiune	Tip Operanți	Descriere
`PADD`	8/16/32 biți	Adunare cu wrap-around
`PSUB`	8/16/32 biți	Scădere cu wrap-around
`PMUL`	16 biți	Înmulțire, reținând partea inferioară/superioară
`PADDSB`	8 biți semnat	Adunare cu saturare
`PADDUSB`	8 biți nesemnat	Adunare cu saturare
`PSUBSB`	8 biți semnat	Scădere cu saturare

Tabela 1: Instrucțiunile implementate și caracteristicile lor

Tabel sinteză a instrucțiunilor: Prin acest set de instrucțiuni, proiectul ilustrează atât capacitățile generale ale arhitecturilor SIMD, cât și complexitatea adăugată de gestionarea overflow-ului (prin saturare) sau prin adaptarea logicii la dimensiunile variate ale elementelor vectoriale. Astfel, implementarea oferă o imagine clară asupra modului în care extensiile SIMD, precum MMX, pot fi folosite pentru a accelera operații intensive în medii hardware dedicate (precum FPGA) sau direct în procesoare comerciale.

2.4 Relevanța Instrucțiunilor SIMD în Aplicațiile Moderne

Arhitecturile care utilizează instrucțiuni SIMD, precum extensiile MMX, au devenit fundamentale în optimizarea performanțelor aplicațiilor contemporane ce manipulează volume mari de date într-un mod intensiv și repetitiv. Prin aplicarea simultană a aceleiași instrucțiuni pe seturi întregi de elemente de date, se obține o paralelizare intrinsecă a calculului, redusă la nivel hardware. Acest lucru are ca efect scurtarea semnificativă a timpului de execuție în raport cu abordările tradiționale, în care fiecare operație asupra unui element de date este efectuată secvențial.

Domenii de aplicație:

- **Procesare multimedia (imagini, audio, video):** Operațiile de tip PADD, PSUB, PADDS sau PADDUS se întâlnesc frecvent în algoritmii de filtrare a imaginilor, comprimare video, ajustare a nivelurilor de contrast sau luminozitate. Logica de saturare previne distorsiuni vizibile, iar wrap-around-ul asigură continuitatea calculelor acolo unde modelul de date o cere. În plus, instrucțiunile PMULLW/PMULHW pot fi folosite în calcule complexe pentru aplicații de filtrare sau efecte speciale, reducând latența calculelor prin paralelizare.
- **Analiză de semnal și domeniul DSP (Digital Signal Processing):** În prelucrarea audio, codarea și decodarea semnalelor audio, filtrarea în timp real, precum și în aplicațiile de recunoaștere a vorbirii, operațiile SIMD permit manipularea rapidă a unor vectori mari de date. Funcționalitățile cu saturare (PADDSB, PADDUSB, PSUBSB) previn depășirea intervalului reprezentabil, menținând integritatea semnalului.
- **Aplicații criptografice și securitate:** Operațiile vectoriale sunt utile în algoritmi ce implică transformări repetitive asupra datelor, cum ar fi criptarea bloc cu algoritmi simetrici (AES) sau hash-uri (SHA). Partea de înmulțire (PMULLW, PMULHW) poate accelera anumite secvențe de calcule, iar adunările și scăderile vectoriale ajută la manipularea rapidă a blocurilor de date în paralel.

- **Inteligență Artificială și învățare automată (Machine Learning):** Modelele de rețele neuronale convoluționale (CNN) sau alte tipuri de rețele utilizate în procesarea imaginilor, recunoașterea obiectelor sau clasificare pot beneficia de operații SIMD. Adunarea, scăderea și înmulțirea vectorizată reduc semnificativ timpul de inferență sau antrenament pentru rețele cu mulți parametri.

Beneficiile SIMD în contextul modern: Pe fondul cererii tot mai mari de putere de calcul, arhitecturile SIMD oferă un compromis excelent între cost, putere consumată și performanță. Într-o lume în care consumatorii de date multimedia și inteligență artificială devin normali, extensiile de tip MMX și instrucțiunile SIMD asociate reprezintă o unealtă puternică în arsenalul inginerilor hardware și software, ajutând la menținerea unor experiențe fluente și a unor timpi de răspuns scăzuți chiar și în scenarii cu cerințe computationale ridicate.

2.5 Soluții SIMD Paralele și Comparații

De-a lungul anilor, extensiile SIMD au evoluat constant pentru a răspunde cerințelor tot mai mari de performanță și flexibilitate. MMX a fost un prim pas în lumea instrucțiunilor vectoriale pentru procesoare x86, însă arhitecturile moderne au depășit limitările inițiale, oferind suport mai extins, dimensiuni mai mari ale registrelor și seturi de instrucțiuni mai sofisticate.

Compararea cu alte extensii SIMD pentru x86:

- **SSE (Streaming SIMD Extensions)** şi **SSE2, SSE3, SSE4:** După introducerea MMX, Intel a lansat SSE, care a adus registre pe 128 de biți (XMM) și suport pentru date în virgulă mobilă pe lângă cele integrale. SSE2 şi versiunile ulterioare au extins setul de instrucțiuni şi tipurile de date suportate, permițând operarea pe date double-precision şi oferind o mai mare flexibilitate. Astfel, SSE a devenit standardul de facto pentru procesarea vectorială pe arhitectura x86, înlocuind treptat utilizarea MMX.
- **AVX (Advanced Vector Extensions) și AVX-512:** Cu registre de 256 de biți (YMM) şi ulterior 512 de biți (ZMM), aceste extensii introduc un nivel superior de paralelism. De asemenea, AVX utilizează un model arhitectural care reduce necesitatea de a păstra compatibilitatea strictă cu MMX şi SSE, rezultând într-un design mai scalabil, potrivit pentru aplicații care implică calcule ştiințifice, analiză de date şi machine learning. AVX-512, prezent pe procesoare de server sau HPC, duce această paralelizare şi mai departe, acoperind un spectru larg de tipuri de date şi instrucțiuni specializate.

3 Analysis

3.1 Propunerea Proiectului

Proiectul urmărește dezvoltarea şi validarea unei unităţi aritmetice SIMD bazate pe un subset de instrucţiuni MMX, implementată complet în hardware pe o platformă FPGA (Nexys A7). Această unitate va permite execuţia paralelă a operaţiilor vectoriale asupra datelor de 8, 16 şi 32 de biţi, organizate într-un registru de 64 de biţi divizat în segmente independente. Fiecare segment reprezintă un operand elementar, asupra căruia se aplică simultan aceeaşi instrucţiune aritmetică, obţinând astfel un grad ridicat de paralelizare.

Instrucţiunile SIMD avute în vedere includ:

- **PADD şi PSUB (adunări şi scăderi vectoriale)**: Operaţii paralele pe elemente multiple, cu aritmetică tip wrap-around, care gestionează depăşirile prin reluarea de la zero a domeniului numeric.
- **PMULLW şi PMULHW (înmulţiri pe 16 biţi)**: Operaţii de înmulţire vectorială între elemente de 16 biţi, extrăgând fie partea inferioară (low) a rezultatului (PMULLW), fie partea superioară (high) (PMULHW), optimizând astfel complexitatea hardware.
- **PADDS, PADDUS, PSUBS (operaţii cu saturare)**: Aceste instrucţiuni previn overflow-ul şi underflow-ul pentru date semnate sau nesemnate, limitând rezultatele la intervalele permise. Ele sunt esenţiale în aplicaţii precum procesarea audio-video şi a imaginilor, unde acurateţea numerică este critică.

Flexibilitatea organizării datelor şi suportul pentru mai multe tipuri şi dimensiuni de elemente facilitează utilizarea unităţii aritmetice MMX într-o gamă largă de aplicaţii, de la procesarea grafică şi criptografie, până la analiza de semnale şi date complexe. Figura 1 ilustrează structura generală a datelor pe 64 de biţi, evidenţiind segmentarea acestora.

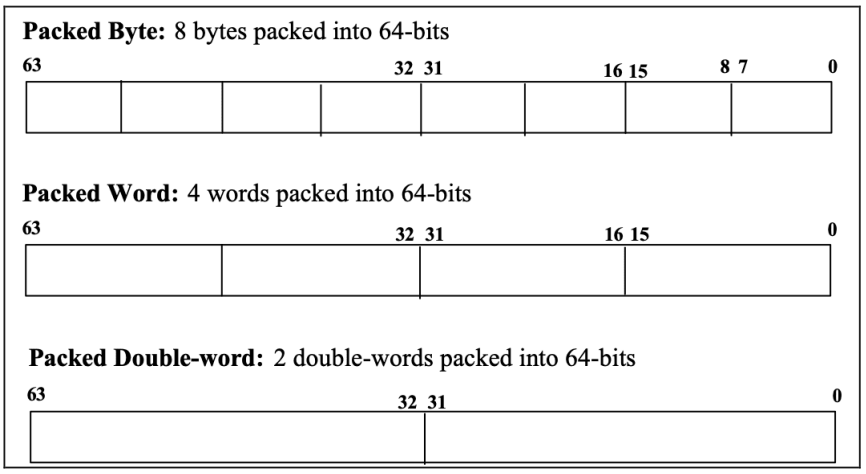

Figura 1: Structura datelor pe 64 de biți utilizată de instrucțiunile MMX.

Obiectivul principal al proiectului este de a demonstra beneficiile paralelizării hardware a operaţiilor vectoriale, utilizând o arhitectură modulară şi scalabilă. Acest obiectiv implică:

- Implementarea unui set de module dedicate fiecărei categorii de instrucţiuni (adunare, scădere, înmulţire, saturare).
- Dezvoltarea unui selector de operaţii (control logic) care interpretează opcoduri şi activează modulul corespunzător.

- Integrarea componentelor periferice (memorie pentru operanzi, afişaj cu 7 segmente, selector de biţi) pentru a facilita testarea şi vizualizarea rezultatelor în timp real.

Implementarea pe FPGA Nexys A7 asigură un mediu practic pentru testare, oferind resurse hardware adecvate şi flexibilitate în proiectarea cu VHDL. Testele vor acoperi:

- Scenarii tipice, pentru verificarea corectitudinii operațiilor uzuale.
- Cazuri limită, ce includ overflow, underflow şi saturare, pentru a evalua robustetea logicii de calcul.
- Comparaţii cu implementări software echivalente, pentru a demonstra îmbunătăţirea performanţei şi a latenţelor.

Prin această abordare modulară, proiectul nu se limitează la testarea funcţionalităţii iniţiale, ci oferă infrastructura necesară extinderii ulterioare cu instrucţiuni noi sau optimizări suplimentare. Astfel, proiectul reprezintă un cadru flexibil pentru studierea, validarea şi exploatarea arhitecturilor SIMD hardware, evidenţiind potenţialul lor în accelerarea aplicaţiilor moderne.

3.2 Analiza Arhitecturală și Hardware

Unitatea aritmetică MMX este concepută într-o manieră modulară, fiecare operație SIMD fiind implementată ca un modul hardware independent. Această abordare aduce avantaje semnificative: permite testarea și validarea individuală a fiecărei componente înainte de integrarea finală, reduce complexitatea depanării și asigură un grad ridicat de flexibilitate în extinderea sau optimizarea ulterioară a sistemului. În plus, structura modulară maximizează reutilizarea resurselor hardware, contribuind la o implementare eficientă din punct de vedere al consumului de LUT-uri, flip-flop-uri și memorie pe FPGA-ul Nexys A7.

Arhitectura globală a sistemului include următoarele componente principale:

1. **Module dedicate operațiilor vectoriale (PADD, PSUB, PMUL):** Aceste module procesează datele în paralel, pe segmente de 8, 16 sau 32 de biți, extrăgând rezultate partiale (în cazul PMULL/PMULHW, secțiunea low sau high a rezultatului). Aritmetica de tip wrap-around pentru PADD/PSUB și optimizările structurale pentru PMUL asigură latențe reduse și un throughput ridicat.
2. **Module pentru operații cu saturare (PADDS, PADDUS, PSUBS):** Aceste componente gestionează situațiile de overflow și underflow prin saturarea rezultatelor la limite predefinite, esențială în procesarea multimedia. Logica de saturare detectează rapid depășirile și ajustează rezultatul fără a penaliza excesiv performanța.
3. **Selectorul de operații (Control Logic):** Un modul combinatorial interpretează codurile de control (opcodurile) și activează modulul corespunzător. Astfel, numai resursele necesare pentru operația selectată sunt utilizate, sporind eficiența energetică și reducând latențele inutile.
4. **Mediul de testare și interfață cu utilizatorul:** Un modul de memorie interne stochează perechi de operanzi și asigură acces ușor la date. Un selector de biți permite afișarea părților relevante ale rezultatului final, iar afișajul cu 7 segmente prezintă valorile numeric. Butoanele FPGA oferă posibilitatea navigării prin seturile de operanzi și selectării operațiilor, facilitând testarea interactivă.

Din perspectiva performanței, procesarea paralelă a datelor pe segmente distincte este cheia îmbunătățirii latenței și creșterii ratelor de procesare (throughput). Arhitectura asigură un flux constant de date: în fiecare ciclu de ceas, unitatea primește operanzi noi și livrează rezultate, beneficiind de pipeline-urile interne și sincronizarea la semnalul global de ceas.

Optimizările la nivel de arhitectură (reutilizarea modulelor comune, partajarea magistralelor și utilizarea structurilor hardware eficiente precum Wallace Tree pentru înmulțire) contribuie la

o implementare echilibrată între consumul de resurse și performanță. Astfel, unitatea aritmetică MMX demonstrată pe FPGA evidențiază potențialul instrucțiunilor SIMD hardware de a accelera substanțial aplicațiile moderne, menținând totodată un design scalabil și ușor de extins.

3.3 Provocări și Decizii de Design

Proiectarea și implementarea unei unități aritmetice MMX pe FPGA au implicat înfruntarea unor provocări tehnice semnificative. Acestea au necesitat alegeri de proiect fundamentale pentru a asigura performanță, corectitudine și utilizarea eficientă a resurselor hardware. În plus, complexitatea operațiilor (de la adunări și scăderi cu wrap-around, până la saturare și multiplicări pe segmente de 16 biți) a impus abordări atent balansate între latență, consum de resurse și flexibilitatea designului.

Principalele provocări:

1. **Sincronizarea între module:** Fiecare operație SIMD are caracteristici diferite de latență, ceea ce a necesitat introducerea pipeline-urilor și a registrelor intermediare pentru alinierea semnalelor. Astfel, fluxul de date rămâne coerent în fiecare ciclu de ceas.
2. **Consumul de resurse hardware limitate:** FPGA-ul Nexys A7 are un număr finit de LUT-uri, flip-flop-uri și blocuri de memorie. A fost esențială reutilizarea modulelor comune (de exemplu, adunătoare și circuite de saturare) și partajarea magistralelor, pentru a obține un design compact și eficient.
3. **Gestionarea overflow-ului la operațiile cu saturare:** Operațiile PADDS, PADDUS și PSUBS au necesitat logică suplimentară de detecție și ajustare a rezultatelor, fără a afecta semnificativ performanța. Implementarea a implicat circuite combinate care reacționează rapid la depășirea limitelor.
4. **Verificare, testare și validare:** Pentru a asigura corectitudinea, a fost dezvoltat un mediu de test complet, cu memorie internă pentru date, afișaj cu 7 segmente și butoane pentru selecția operanzilor. Testarea manuală, combinată cu simulări software, a permis identificarea timpurie a erorilor și corectarea lor iterativă.
5. **Integrarea componentelor periferice:** Adăugarea modulului de memorie, a selectorului de biți și a afișajului a impus sincronizarea fluxului de date și gestionarea corectă a semnalelor de control. Interfața utilizatorului a fost proiectată astfel încât selecția și afișarea rezultatelor să fie intuitive și rapide.

Deciziile de design adoptate:

- **Arhitectură modulară:** Fragmentarea fiecărei operații SIMD în module independente a simplificat depanarea, testarea și eventualele extinderi.
- **Pipeline și sincronizare globală:** Introducerea pipeline-urilor și utilizarea semnalului global de ceas au asigurat latențe previzibile și un flux de date continuu, esențial pentru throughput ridicat.
- **Echilibru între performanță și resurse:** Prin reutilizarea circuitelor comune și optimizarea internă a modulelor (de exemplu, Wallace Tree pentru înmulțire), s-a obținut un design scalabil, cu un bun compromis între latență și consumul de resurse.
- **Interfață prietenoasă de testare și debugging:** Integrarea memoriei interne, a afișajului și a butoanelor de selecție a datelor a transformat procesul de verificare într-unul interactiv, permițând validarea în timp real a rezultatelor.

Astfel, provocările inițiale ale proiectului au condus la decizii de design solide, care au asigurat nu doar funcționalitatea și performanța cerute, ci și un grad înalt de extensibilitate și adaptabilitate pentru aplicații viitoare.

4 Design

4.1 Arhitectura Generală a Sistemului

Arhitectura unității aritmetice MMX este modulară, asigurând flexibilitate, scalabilitate și ușurință în testare. Fiecare modul are o funcționalitate clară, interconectată prin semnale de control și magistrale de date.

Sistemul include următoarele componente principale:

1. **MemoryUnit**: Stochează vectori pe 64 de biți, **A** și **B**, navigabili prin butoane FPGA. Asigură datele necesare execuției SIMD.
2. **MMX_Unit**: Nucleul sistemului, implementează instrucțiunile SIMD (`PADD`, `PSUB`, `PADDS`, `PADDUS`, `PSUBS`, `PMUL`), primind operanzi de la **MemoryUnit** și generând rezultate pe 64 de biți.
3. **BitSelector**: Selectează jumătatea inferioară/superioară a rezultatului (`SelectBit`), indicată vizual prin LED.
4. **SSD (Seven-Segment Display)**: Afișează rezultatul selectat, gestionând multiplexarea și decodificarea.

Componentele interacționează sincronizat printr-un semnal global de ceas (*clk*). **MemoryUnit** furnizează date către **MMX_Unit**, care procesează operațiile SIMD. Rezultatul este direcționat către **BitSelector** pentru selecție, apoi afișat pe **SSD**.

Figura 2 prezintă arhitectura top-level, evidențiind conexiunile dintre module. Structura modulară facilitează depanarea și extinderea cu noi funcționalități.

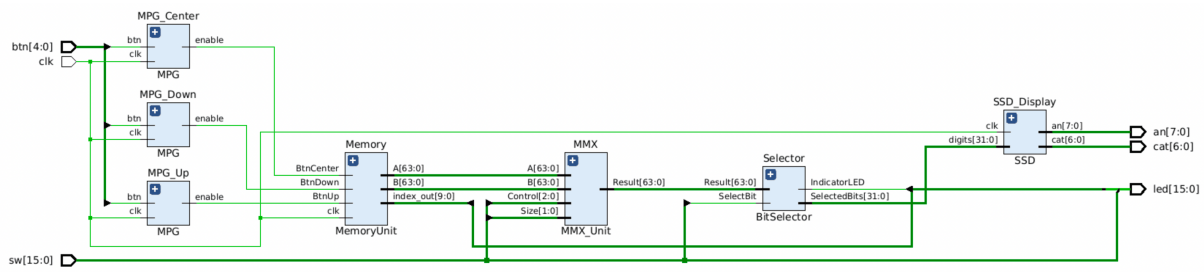

Figura 2: Arhitectura generală (top-level) a sistemului MMX.

4.2 Descrierea Modulelor Principale

Fundamentul designului arhitecturii MMX îl reprezintă modularitatea, principiul care asigură flexibilitate, scalabilitate şi simplifică testarea individuală a componentelor. În cadrul acestui sistem, fiecare modul îndeplineşte un rol clar, contribuind la fluxul general de prelucrare a datelor, de la stocarea lor până la afişarea rezultatului final. Modulul central, **MMX_Unit**, constituie nucleul operaţiunilor SIMD, fiind susţinut de **MemoryUnit**, **BitSelector** şi **SSD**.

4.2.1 MemoryUnit

Modulul **MemoryUnit** gestionează stocarea şi furnizarea operanzilor **A** şi **B** către **MMX_Unit**. Structura internă este organizată sub formă de perechi de vectori pe 64 de biţi, accesibili prin intermediul butoanelor de control (`BtnUp`, `BtnDown`, `BtnCenter`). Principalele sale caracteristici sunt:

- **Navigare între seturi de date**: Utilizatorul poate parcurge diferite perechi de operanzi, revenind la poziţia iniţială printr-un reset.

- **Compatibilitate pe 64 de biţi**: Vectorii **A** şi **B** sunt furnizaţi în format pe 64 de biţi, pregătiţi pentru execuţia operaţiilor SIMD.
- **Afişarea indexului curent**: Semnalul `index_out` indică poziţia curentă din memorie, oferind claritate în timpul testării şi utilizării.

4.2.2 MMX_Unit

MMX_Unit reprezintă centrul logicii SIMD, integrând un set de submodule dedicate fiecărei instrucţiuni implementate, cum ar fi `PADD` (adunare), `PSUB` (scădere), `PADDS` (adunare saturată), `PADDUS` (adunare saturată nesemnată), `PSUBS` (scădere saturată), și `PMULL` (înmulțire). Selecţia operaţiei corespunzătoare se realizează prin intermediul semnalului `Control`, care, reprezentat printr-un vector de 3 biți, determină care submodul va fi activat pentru a efectua operaţia solicitată. Această arhitectură permite activarea exclusivă a submodulului necesar, optimizând astfel fluxul de date și utilizarea resurselor hardware.

Figura 3 prezintă schema bloc a unității MMX, evidențiind interconectarea dintre submodule și modulul de control central. Fiecare submodul primește operandul `A` și operandul `B`, precum și semnalul `Size`, care indică dimensiunea operațiilor (8-bit, 16-bit sau 32-bit). Submodulele `PMULLW` și `PMULHW` sunt gestionate de componenta `PMUL`, care utilizează semnalul `Mode` pentru a selecta între partea inferioară și partea superioară a cuvintelor în procesul de multiplicare.

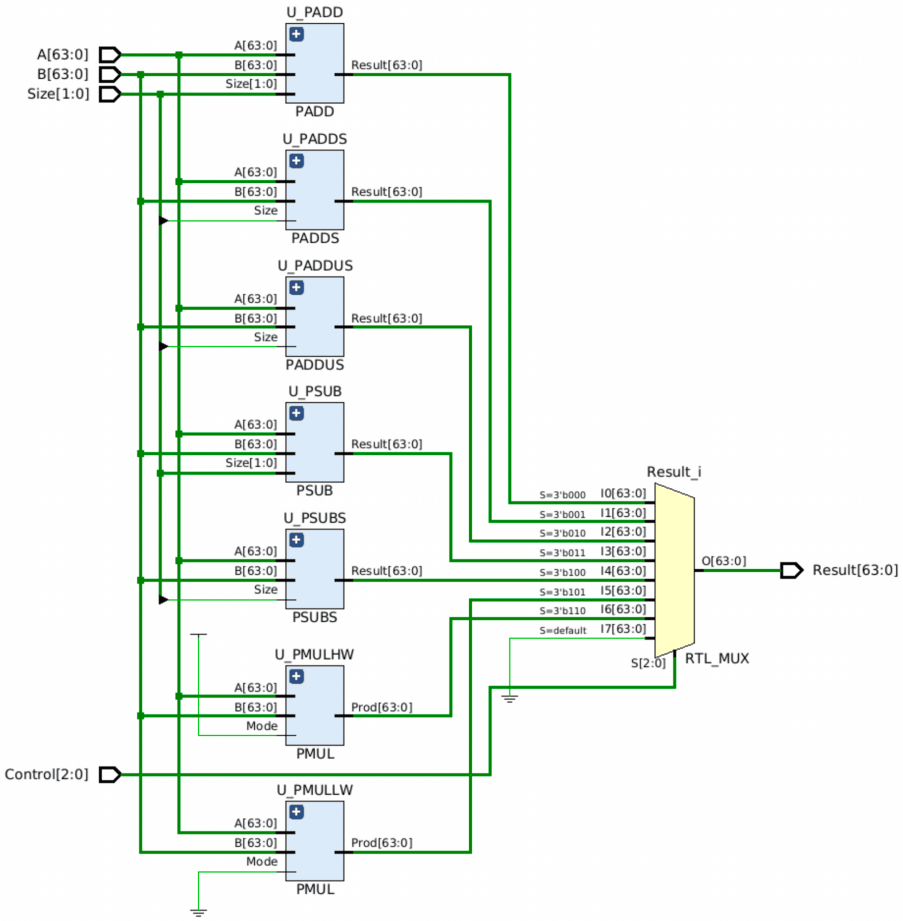

Figura 3: Schema bloc a unității MMX

Avantajele acestui design includ:

- **Modularitate**: Fiecare operaţie SIMD este un submodul independent, testabil individual, facilitând astfel identificarea și corectarea erorilor.
- **Selecţie centralizată**: Semnalul `Control` determină operaţia activă, conectând direct rezultatul submodulului la ieşirea finală, optimizând fluxul de date.
- **Extensibilitate**: Adăugarea de noi instrucţiuni SIMD sau optimizări devine facilă datorită structurii modulare a designului, permițând adaptarea la cerințe viitoare fără modificări semnificative ale arhitecturii existente.

4.2.3 BitSelector

Modulul **BitSelector** oferă o interfaţă flexibilă pentru afişarea rezultatelor procesate. Rezultatul de 64 de biţi din **MMX_Unit** poate fi restrâns la 32 de biţi, afişând doar partea inferioară sau superioară a vectorului. Caracteristicile sale includ:

- **Selecţie dinamică**: Semnalul `SelectBit` determină dacă se afişează partea inferioară (lower 32 bits) sau superioară (upper 32 bits).
- **Indicator LED**: Un LED semnalizează clar utilizatorului care segment de date este selectat.
- **Adaptabilitate**: Modulul acceptă un vector de intrare pe 64 de biţi şi furnizează o ieşire optimă (32 de biţi) pentru afişajul **SSD**.

4.2.4 SSD (Seven-Segment Display)

Modulul **SSD** converteşte datele pe 32 de biţi în caractere afişabile pe un panou cu 7 segmente. El gestionează multiplexarea dinamică a cifrelor şi decodificarea valorilor, oferind un feedback vizual uşor de interpretat. Principalele beneficii sunt:

- **Afişare lizibilă**: Valorile binare sunt translate în cifre şi litere, facilitând interpretarea rapidă a rezultatelor.
- **Multiplexare eficientă**: Selectează pe rând fiecare cifră pentru a minimaliza numărul de pini utilizaţi.
- **Integrare armonioasă**: **SSD** primeşte date direct de la **BitSelector**, permiţând utilizatorului să observe instantaneu rezultatele operaţiilor.

4.2.5 Integrarea modulelor

Fiecare modul funcţionează independent, dar integrarea lor asigură un flux coerent de date. **MemoryUnit** livrează vectorii către **MMX_Unit**, care execută operaţia selectată şi transmite rezultatul către **BitSelector**. Acesta alege segmentul de date ce urmează a fi afişat de **SSD**. Designul modular şi interconectarea simplă a modulelor asigură funcţionarea robustă şi facilitează depanarea, testarea şi extinderea sistemului.

4.3 Operațiile SIMD

Operațiile SIMD (Single Instruction, Multiple Data) reprezintă nucleul funcțional al unității aritmetice MMX, permițând aplicarea aceleiași instrucțiuni pe mai multe elemente de date în paralel. Prin acest mecanism, unitatea poate procesa eficient volume mari de date numerice, reducând semnificativ timpul de execuție față de soluțiile scalare tradiționale. În cadrul acestui proiect, au fost implementate un set reprezentativ de instrucțiuni MMX, acoperind aritmetica elementelor vectoriale pe 8, 16 și 32 de biți, precum și variante cu saturare sau înmulțire pe word-uri de 16 biți.

Instrucțiunile implementate includ:

- **Adunări vectoriale** (`PADD`): Operă pe elemente de 8, 16 sau 32 de biți, adunându-le cu wrap-around. Sunt implementate variante precum `PADDB`, `PADDW` și `PADDD`, fiecare tratând dimensiuni diferite ale datelor.
- **Scăderi vectoriale** (`PSUB`): Analog cu adunările, scăderile folosesc wrap-around pentru a menține integritatea rezultatului. Instrucțiunile `PSUBB`, `PSUBW` și `PSUBD` asigură flexibilitate în funcție de dimensiunea elementelor.
- **Operații cu saturare** (`PADDS`, `PADDUS`, `PSUBS`): Aceste instrucțiuni previn overflow-ul sau underflow-ul, limitând rezultatele la intervale maxime sau minime. Astfel, `PADDSB`, `PADDUSB` și `PSUBSB` devin esențiale în prelucrarea semnalelor audio-video și în alte aplicații ce necesită precizie controlată.
- **Înmulțiri pe word-uri de 16 biți** (`PMUL`): Operația de înmulțire este realizată prin instrucțiuni precum `PMULLW` și `PMULHW`, care calculează partea inferioară, respectiv superioară a produsului pe 32 de biți. Această abordare permite manipularea eficientă a datelor cu precizie redusă, fiind utilă în filtre digitale și alte aplicații ce nu necesită întreaga lățime a rezultatului.

Fiecare dintre aceste instrucțiuni este integrată modular în **MMX_Unit**, utilizând un semnal de control pentru selectarea operației active. Abordarea modulară permite testarea individuală, optimizarea fiecărei instrucțiuni și extinderea facilă cu noi funcționalități în viitor. Rezultatul final, adaptat la dimensiunea și tipul datelor, demonstrează versatilitatea arhitecturii și confirmă beneficiile procesării vectoriale în aplicații moderne ce necesită paralelizare și randament ridicat.

4.3.1 Conceptul SIMD în contextul MMX

Operațiile SIMD (Single Instruction, Multiple Data) permit executarea unei singure instrucțiuni asupra mai multor elemente de date în paralel. În cazul arhitecturii MMX, aceasta înseamnă că același tip de operație aritmetică (adunare, scădere, înmulțire sau variantă cu saturare) este aplicată simultan pe segmentele vectoriale componente ale unui registru pe 64 de biți.

Fiecare registru de 64 de biți este împărțit în elemente de dimensiuni fixe (8, 16 sau 32 de biți), dimensiunea fiind selectată dinamic prin semnale de control. Astfel, MMX poate opera eficient atât pe seturi mari de elemente mici (ex. octeți, pentru imagini sau audio), cât și pe elemente mai mari (word-uri sau doubleword-uri), adaptându-se la cerințele aplicației.

Conceptul SIMD în contextul MMX aduce următoarele beneficii:

- **Paralelizare sporită:** În loc să se execute mai multe instrucțiuni succesive pentru fiecare element, o singură instrucțiune acoperă un întreg vector de date.
- **Performanță îmbunătățită:** Prin reducerea ciclurilor de ceas necesare procesării unor seturi mari de date, SIMD scade latența totală și crește lățimea de bandă a calculelor.
- **Scalabilitate:** Operațiile sunt ușor extensibile pentru a suporta noi tipuri de instrucțiuni sau dimensiuni ale datelor, menținând aceeași structură modulară.

În implementarea curentă, logica de selecție a operațiilor și a dimensiunii elementelor funcționează integrat cu modulele aritmetice individuale. Astfel, sistemul poate trece transparent de la adunări pe bytes la înmulțiri pe word-uri sau scăderi cu saturare, fără a modifica arhitectura de bază. Acest design flexibil subliniază importanța și relevanța principiilor SIMD, oferind o fundație solidă pentru adaptarea la cerințele variate ale aplicațiilor moderne.

4.3.2 PADD – Adunări Vectoriale

Operația `PADD` (Packed ADD) reprezintă una dintre cele mai fundamentale instrucțiuni SIMD implementate în cadrul unității MMX. Scopul principal al acestei operații este de a efectua adunări paralele între elementele corespunzătoare ale vectorilor de intrare **A** și **B**, fiecare segment de date fiind tratat independent în funcție de dimensiunea specificată (8, 16 sau 32 de biți). Această abordare permite procesarea simultană a mai multor elemente de date într-un singur ciclu de ceas, maximizând astfel performanța și eficiența hardware.

Pentru a înțelege pe deplin implementarea și funcționarea operației `PADD`, este esențial să analizăm structura modulară a unității aritmetice MMX, în special interacțiunea dintre modulele `AdderSubtractor8bit` și `AdderSubtractor1bit`. Modulul `AdderSubtractor8bit` este responsabil pentru gestionarea adunărilor pe 8 biți, fiind compus din opt instanțe ale modulului `AdderSubtractor1bit`. Fiecare instanță `AdderSubtractor1bit` efectuează adunarea a doi biți individuali, gestionând carry-ul (transportul) între biți pentru a asigura corectitudinea rezultatului final.

Procesul începe cu semnalul `Size`, care determină dimensiunea fiecărui element de date (8, 16 sau 32 de biți). În funcție de această dimensiune, `AdderSubtractor8bit` activează segmentele corespunzătoare ale vectorilor **A** și **B**, propagând carry-ul între blocuri pentru a realiza adunările corecte. Fiecare bloc `AdderSubtractor1bit` primește un bit din vectorii de intrare și efectuează adunarea acestuia, propagând carry-ul către următorul bloc de biți. Rezultatul fiecărei adunări pe 8 biți este apoi combinat pentru a forma rezultatul final pe 64 de biți.

Un aspect esențial al operației `PADD` este gestionarea *wrap-around*-ului, care se referă la comportamentul ciclic al rezultatelor atunci când acestea depășesc limita maximă reprezentabilă pentru dimensiunea specificată. De exemplu, pentru operații pe 8 biți, o sumă care depășește 255 va reveni la 0, asigurând astfel că rezultatele rămân în intervalul valid. Această abordare este crucială în aplicații precum procesarea imaginilor și a semnalelor audio, unde integritatea datelor trebuie menținută fără a introduce erori numerice semnificative.

Implementarea hardware a `PADD` utilizează o arhitectură modulară, unde fiecare bloc `AdderSubtractor1bit` este responsabil pentru o parte a adunării. Această separare permite o paralelizare eficientă și o utilizare optimă a resurselor hardware disponibile pe FPGA-ul Nexys A7. În plus, structura modulară facilitează extinderea și întreținerea designului, permițând adăugarea ușoară a unor noi operații sau optimizări fără a afecta funcționalitatea existentă.

Figura 4 ilustrează diagrama bloc a operației `PADD`, evidențiind interacțiunea dintre modulele `AdderSubtractor8bit` și `AdderSubtractor1bit`. Diagrama arată cum fiecare segment de 8 biți este procesat independent, cu carry-ul fiind propagat între segmente pentru a asigura corectitudinea adunărilor. De asemenea, figura evidențiază modul în care semnalele de control (`Size`) influențează selecția segmentelor active și gestionarea wrap-around-ului.

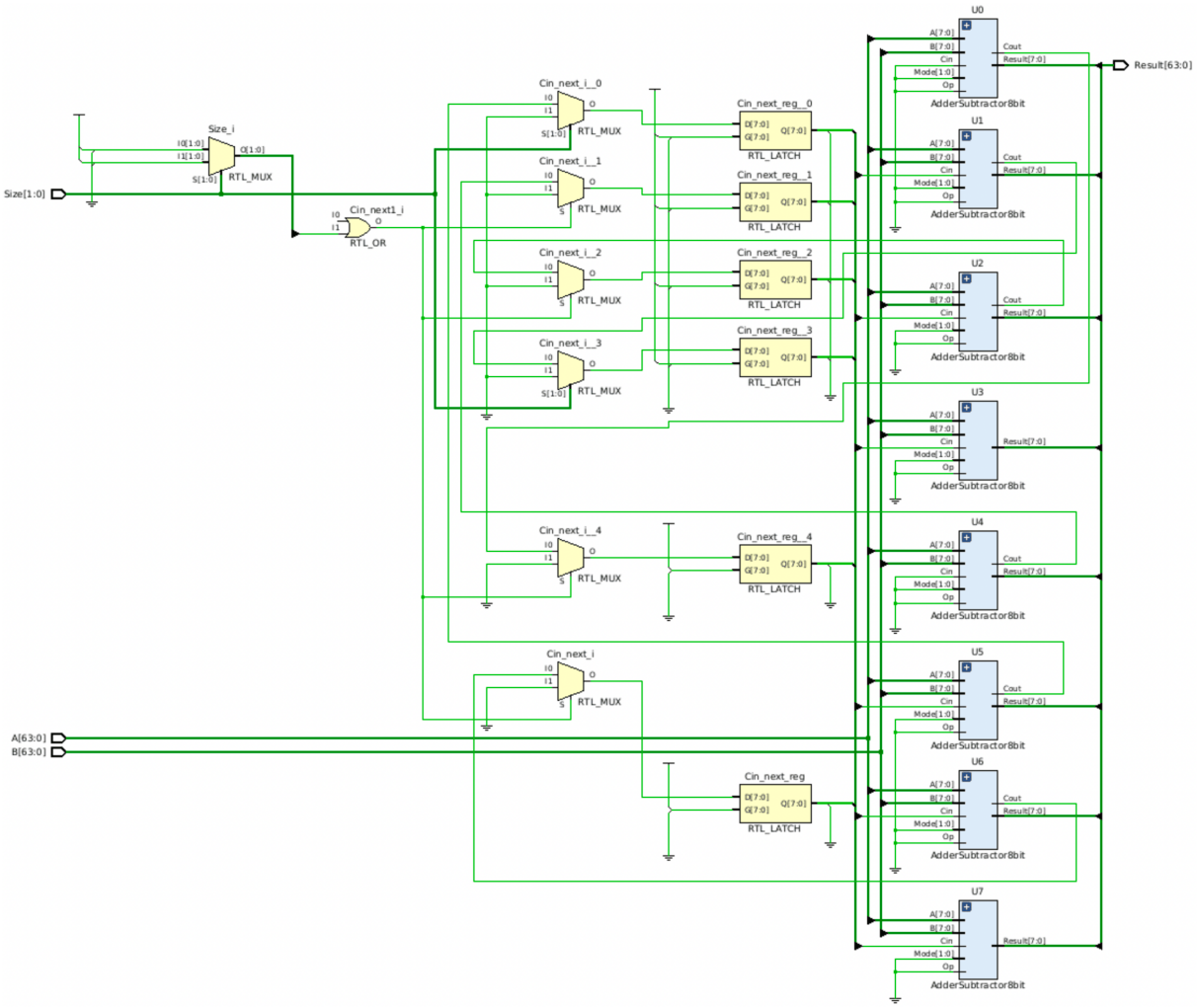

Figura 4: Diagramă bloc pentru implementarea operației `PADD`, evidențiind structura modulară și adaptabilă la diferite dimensiuni ale datelor.

Fluxul Operațional al `PADD`:

1. **Primirea Operandilor**: Vectorii **A** și **B** sunt încărcați în modulul `MemoryUnit`, care le furnizează către `MMX_Unit`.
2. **Configurarea Dimensiunii**: Semnalul `Size` determină dimensiunea elementelor de date și activează segmentele corespunzătoare în `AdderSubtractor8bit`.
3. **Executarea Adunării**: Fiecare bloc `AdderSubtractor1bit` efectuează adunarea a doi biți individuali, propagând carry-ul către următorul bloc.
4. **Gestionarea Wrap-Around**: În cazul în care rezultatul unei adunări depășește limita maximă pentru dimensiunea specificată, valoarea rezultată se întoarce la 0, asigurând comportamentul ciclic.
5. **Combinarea Rezultatelor**: Rezultatele parțiale din fiecare bloc de 8 biți sunt combinate pentru a forma rezultatul final pe 64 de biți.
6. **Transmiterea Rezultatului**: Rezultatul este trimis către `BitSelector` pentru selecția segmentului dorit și afișare pe `SSD`.

Această secțiune a documentației detaliază implementarea `PADD` în contextul unității MMX, evidențiind modul în care arhitectura modulară și utilizarea blocurilor dedicate de adunare asigură performanță ridicată și flexibilitate. Prin combinarea operațiilor paralele cu gestionarea

eficientă a wrap-around-ului, `PADD` contribuie semnificativ la capacitatea generală a unității MMX de a executa operații aritmetice complexe într-un mod eficient și fiabil.

Interacțiunea cu `AdderSubtractor8bit` și `AdderSubtractor1bit` Modulul `AdderSubtractor8bit` joacă un rol crucial în implementarea `PADD`, orchestrând operațiile de adunare pe 8 biți prin intermediul celor opt instanțe ale modulului `AdderSubtractor1bit`. Fiecare instanță `AdderSubtractor1bit` este responsabilă pentru un bit individual al segmentului de 8 biți, efectuând adunarea a doi biți și gestionând carry-ul (transportul) între biți.

În cadrul `AdderSubtractor8bit`, semnalul `Cin` este configurat pentru a inițializa carry-ul la începutul fiecărui segment și pentru a propaga carry-ul de la un segment la altul, în funcție de dimensiunea specificată de semnalul `Size`. Acest mecanism asigură că adunările sunt realizate corect și că rezultatele sunt consistente cu așteptările pentru fiecare dimensiune de date.

Modulul `AdderSubtractor1bit` este implementat pentru a efectua adunarea unui singur bit, gestionând atât carry-ul în cazul operațiilor de adunare, cât și borrow-ul în cazul operațiilor de scădere. În contextul `PADD`, acesta este configurat să execute doar operațiile de adunare, utilizând semnalul de control `Op` pentru a diferenția între adunare și scădere.

Prin utilizarea acestor module de bază, operația `PADD` poate fi extinsă și adaptată pentru diferite dimensiuni de date fără a necesita modificări semnificative ale structurii hardware. Această flexibilitate este esențială pentru asigurarea scalabilității și eficienței unității MMX în procesarea paralelă a datelor.

4.3.3 PSUB – Scăderi Vectoriale Generalizate

Operația `PSUB` (Packed SUBtract) extinde funcționalitatea unității MMX prin introducerea capacității de a efectua scăderi vectoriale paralele. Similar cu `PADD`, `PSUB` operează pe vectori de 64 de biți, efectuând scăderi între elementele corespunzătoare ale vectorilor de intrare **A** și **B**. Fiecare segment de date este tratat independent, în funcție de dimensiunea specificată (`Size` = "00" pentru 8 biți, `Size` = "01" pentru 16 biți, și `Size` = "10" pentru 32 biți), permițând astfel procesarea simultană a mai multor diferențe într-un singur ciclu de ceas.

Implementarea operației `PSUB` se bazează pe structura modulară a unității aritmetice MMX, utilizând modulele `AdderSubtractor8bit` și `AdderSubtractor1bit`. Modulul `AdderSubtractor8bit` gestionează scăderile pe 8 biți, fiind compus din opt instanțe ale modulului `AdderSubtractor1bit`. Fiecare instanță `AdderSubtractor1bit` este responsabilă pentru calculul diferenței a doi biți individuali și pentru gestionarea *borrow*-ului între biți, asigurând corectitudinea rezultatului final.

Procesul de scădere începe cu semnalul `Size`, care determină dimensiunea fiecărui element de date. În funcție de această dimensiune, `AdderSubtractor8bit` activează segmentele corespunzătoare ale vectorilor **A** și **B**, propagând *borrow*-ul între blocuri pentru a realiza scăderile corecte. Fiecare bloc `AdderSubtractor1bit` primește un bit din vectorii de intrare și efectuează scăderea acestuia, propagând *borrow*-ul către următorul bloc de biți. Rezultatul fiecărei scăderi pe 8 biți este apoi combinat pentru a forma rezultatul final pe 64 de biți.

Un aspect esențial al operației `PSUB` este gestionarea *wrap-around*-ului negativ, care se referă la comportamentul ciclic al rezultatelor atunci când acestea scad sub limita minimă reprezentabilă pentru dimensiunea specificată. De exemplu, pentru operații pe 8 biți, o diferență care scade sub 0 va reveni la 255, asigurând astfel că rezultatele rămân în intervalul valid. Această abordare este crucială în aplicații precum procesarea audio, unde diferențele negative apar frecvent, sau în implementarea unor algoritmi ciclici în grafica computerizată.

Implementarea hardware a PSUB utilizează o arhitectură modulară similară cu cea a PADD, unde fiecare bloc AdderSubtractor1bit este responsabil pentru o parte a scăderii. Această separare permite o paralelizare eficientă și o utilizare optimă a resurselor hardware disponibile pe FPGA-ul Nexys A7. În plus, structura modulară facilitează extinderea și întreținerea designului, permițând adăugarea ușoară a unor noi operații sau optimizări fără a afecta funcționalitatea existentă.

Fluxul Operațional al PSUB:

1. **Primirea Operandilor**: Vectorii **A** și **B** sunt încărcați în modulul MemoryUnit, care le furnizează către MMX_Unit.
2. **Configurarea Dimensiunii**: Semnalul Size determină dimensiunea elementelor de date și activează segmentele corespunzătoare în AdderSubtractor8bit.
3. **Executarea Scăderii**: Fiecare bloc AdderSubtractor1bit efectuează scăderea a doi biți individuali, propagând *borrow*-ul către următorul bloc.
4. **Gestionarea Wrap-Around Negativ**: În cazul în care rezultatul unei scăderi scade sub limita minimă pentru dimensiunea specificată, valoarea rezultată se "înfășoară" la valoarea maximă a intervalului, asigurând comportamentul ciclic.
5. **Combinarea Rezultatelor**: Rezultatele parțiale din fiecare bloc de 8 biți sunt combinate pentru a forma rezultatul final pe 64 de biți.
6. **Transmiterea Rezultatului**: Rezultatul este trimis către BitSelector pentru selecția segmentului dorit și afișare pe SSD.

Interacțiunea cu AdderSubtractor8bit și AdderSubtractor1bit:

Modulul AdderSubtractor8bit coordonează operațiile de scădere pe 8 biți prin intermediul celor opt instanțe ale modulului AdderSubtractor1bit. Fiecare instanță AdderSubtractor1bit gestionează scăderea a doi biți individuali și propagă *borrow*-ul între biți pentru a asigura corectitudinea rezultatului. Semnalul Cin este configurat pentru a inițializa *borrow*-ul la începutul fiecărui segment și pentru a-l propaga de la un segment la altul, în funcție de dimensiunea specificată de semnalul Size.

Modulul AdderSubtractor1bit este proiectat pentru a efectua scăderea unui singur bit, gestionând *borrow*-ul în cazul operațiilor de scădere. În contextul PSUB, acesta este configurat să execute doar operațiile de scădere, utilizând semnalul de control Op pentru a diferenția între adunare și scădere.

Prin utilizarea acestor module de bază, operația PSUB poate fi extinsă și adaptată pentru diferite dimensiuni de date fără a necesita modificări semnificative ale structurii hardware. Această flexibilitate este esențială pentru asigurarea scalabilității și eficienței unității MMX în procesarea paralelă a datelor, permițând astfel adaptarea la cerințele diverse ale aplicațiilor moderne.

4.3.4 PADDS, PADDUS și PSUBS – Operații de Saturare

Operațiile cu saturare (PADDS, PADDUS și PSUBS) sunt esențiale în prelucrarea datelor vectoriale, asigurând că rezultatele operațiilor aritmetice nu depășesc intervalele permise de reprezentarea datelor. Aceste instrucțiuni previn apariția overflow-ului sau underflow-ului nedorit, menținând integritatea datelor în aplicații critice precum procesarea audio, imagine și semnale digitale.

PADDS – Adunare Vectorială cu Saturare Semnată PADDS (Packed Add with Saturation) efectuează adunări paralele pe date semnate, limitând rezultatele la intervalul $[-128, 127]$ pentru fiecare segment de 8 biți. Dacă suma calculată depășește 127, rezultatul este saturat la 127. În mod similar, dacă diferența scade sub -128, rezultatul este saturat la -128. Acest mecanism

previne depășirile semnate, menținând rezultatele în intervalul valid și evitând erorile numerice în aplicații critice.

PADDUS – Adunare Vectorială cu Saturare Nesemnată `PADDUS` (Packed Add Unsigned with Saturation) operează pe date nesemnate, menținând rezultatele în intervalul $[0, 255]$ pentru fiecare segment de 8 biți. Dacă suma depășește 255, aceasta este limitată la 255. În cazul scăderilor nesemnate, underflow-ul este gestionat astfel încât rezultatul să rămână în intervalul valid, deși în mod normal, scăderile nesemnate nu ar trebui să producă valori negative.

PSUBS – Scădere Vectorială cu Saturare Semnată `PSUBS` (Packed Subtract with Saturation) efectuează scăderi paralele pe date semnate, limitând rezultatele la intervalul $[-128, 127]$ pentru fiecare segment de 8 biți. Dacă diferența scade sub -128, rezultatul este saturat la -128, iar dacă depășește 127, este saturat la 127. Aceasta asigură că scăderile nu duc la valori în afara intervalului permis, menținând integritatea datelor.

Implementarea Operațiilor de Saturare Implementarea hardware a operațiilor de saturare utilizează componentele `AdderSubtractor8bit` pentru a gestiona fiecare segment de 8 biți individual. Fiecare bloc `AdderSubtractor8bit` este responsabil pentru efectuarea adunărilor sau scăderilor pe 8 biți, detectând și gestionând overflow-ul sau underflow-ul în funcție de modul de operare (`Mode`).

Detectarea Overflow-ului și Underflow-ului Fiecare instanță a `AdderSubtractor8bit` include logica necesară pentru detectarea overflow-ului și underflow-ului. Acest lucru se realizează prin monitorizarea semnelor operandilor și rezultatelor. În cazul în care se detectează un overflow pozitiv sau un underflow negativ, rezultatul este ajustat la valoarea maximă sau minimă permisă, respectiv, în funcție de operația efectuată.

Gestionarea Rezultatelor Saturate După detectarea unei depășiri a intervalului permis, rezultatul este saturat la valoarea limită corespunzătoare. Pentru `PADDS` și `PSUBS`, acest lucru implică setarea rezultatului la 127 sau -128 pentru date semnate, și la 255 pentru `PADDUS`. Această logică este integrată direct în procesul de calcul al fiecărui segment de date, permițând operațiilor să fie efectuate în paralel fără a introduce latențe semnificative.

Fluxul Operațional al Operațiilor de Saturare:

1. **Primirea Operandilor**: Vectorii de intrare **A** și **B** sunt încărcați în modulul `MemoryUnit`, care le furnizează către `MMX_Unit`.
2. **Configurarea Modului de Operare**: Semnalele `Size` și `Mode` determină dimensiunea segmentelor de date și tipul de saturare aplicată (`PADDS`, `PADDUS`, `PSUBS`).
3. **Executarea Operației Aritmetice**: Fiecare bloc `AdderSubtractor8bit` efectuează adunarea sau scăderea pe 8 biți, propagând carry-ul sau borrow-ul între segmente.
4. **Detectarea Saturării**: Logica de saturare detectează dacă rezultatul unei operații depășește intervalul permis și determină necesitatea ajustării rezultatului.
5. **Aplicarea Saturării**: În caz de overflow sau underflow, rezultatul segmentului de date este saturat la valoarea limită corespunzătoare.
6. **Combinarea Rezultatelor**: Rezultatele parțiale din fiecare bloc de 8 biți sunt combinate pentru a forma rezultatul final pe 64 de biți.
7. **Transmiterea Rezultatului**: Rezultatul saturat este trimis către `BitSelector` pentru selecția segmentului dorit și afișare pe `SSD`.

Prin utilizarea modulelor `AdderSubtractor8bit` și `AdderSubtractor1bit`, operațiile `PADDS`, `PADDUS` și `PSUBS` sunt realizate eficient și corect, asigurând că rezultatele rămân în intervalele permise și contribuind la integritatea datelor procesate de unitatea MMX.

4.3.5 PMULLW și PMULHW – Înmulțirea pe Word-uri

Operațiile `PMULLW` (*Packed Multiply Low Word*) și `PMULHW` (*Packed Multiply High Word*) extind capacitățile unității MMX în domeniul înmulțirilor vectoriale pe 16 biți. Acestea permit calculul produselor pentru perechi de elemente pe 16 biți din vectorii **A** și **B**, separând rezultatul de 32 de biți într-o parte inferioară (low word) și una superioară (high word).

PMULLW calculează produsul fiecărei perechi de elemente pe 16 biți și extrage doar partea inferioară a rezultatului (16 biți din 32 disponibili). Această abordare este utilă în aplicații unde precizia completă nu este necesară, reducând complexitatea hardware și utilizarea resurselor. Spre exemplu, în filtrarea digitală a semnalelor, unde contează doar o parte din rezultatul final, `PMULLW` optimizează fluxul de date eliminând biții nefolositori.

PMULHW, pe de altă parte, oferă acces la partea superioară a rezultatului înmulțirilor. În situații în care valorile multiplicate pot depăși semnificativ domeniul inferior de 16 biți, partea superioară devine relevantă pentru calcule ulterioare. Astfel, `PMULHW` permite obținerea informațiilor suplimentare despre dimensiunea reală a produsului, element esențial în anumite algoritmi de procesare a datelor numerice sau în situații care necesită menținerea preciziei.

Implementarea internă a `PMULLW` și `PMULHW` utilizează multiplicatori pe 16 biți care operează în paralel pe perechile de cuvinte din vectori. După calculul produsului pe 32 de biți, un mecanism intern selectează partea relevantă (inferioară pentru `PMULLW`, superioară pentru `PMULHW`). Această partajare a multiplicatorilor și logica de selecție asigură utilizarea optimă a resurselor hardware, menținând în același timp flexibilitatea și performanța unității MMX.

Combinarea celor două instrucțiuni, `PMULLW` și `PMULHW`, permite fie obținerea unor rezultate cu precizie redusă (dar cu latență și consum de resurse minime), fie menținerea accesului la informația completă asupra domeniului rezultatului. Această versatilitate face din setul de operații de înmulțire o componentă esențială a infrastructurii SIMD, adaptabilă atât scenariilor cu cerințe de performanță ridicate, cât și celor care necesită un control atent al preciziei rezultate.

4.3.6 Integrarea operațiilor în MMX_Unit

Toate operațiile SIMD discutate anterior sunt integrate într-un singur modul central, **MMX_Unit**. Acest modul acționează ca un centru de comandă și rutare a datelor, coordonând execuția fiecărei instrucțiuni în funcție de semnalul de control `Control` și de dimensiunea operației (`Size`). Prin intermediul unei logici de decodare, **MMX_Unit** identifică ce operație trebuie să fie activată și direcționează vectorii **A** și **B** către modulul corespunzător.

Implementarea modulară asigură că fiecare operație—fie că este vorba de adunare, scădere, înmulțire sau saturare—poate fi testată și optimizată independent. Odată ce rezultatele sunt generate, **MMX_Unit** combină ieșirile provenite de la diverse submodule și oferă un rezultat coerent la ieșire, sub forma unui vector de 64 de biți. În acest fel, orice modificare sau îmbunătățire adusă unei singure operații nu perturbă funcționalitatea celorlalte, facilitând mentenanța și extinderea arhitecturii.

Această abordare modulară în **MMX_Unit** are mai multe avantaje practice:

- **Scalabilitate**: Adăugarea de noi operații SIMD se poate realiza prin includerea unui nou submodul și extinderea logicii de selecție, fără a altera structura de bază.
- **Depanare simplificată**: Problemele apărute într-un anumit modul pot fi izolate și rezolvate individual, reducând semnificativ timpul de depanare.
- **Reutilizare a resurselor**: Unele circuite, cum ar fi cele pentru saturare sau unitățile elementare de adunare/scădere, pot fi utilizate în mai multe operații, optimizând consumul de resurse hardware.

În final, **MMX_Unit** asigură un flux unitar de date și comenzi, oferind consistență în execuția instrucțiunilor SIMD, menținând în același timp flexibilitatea necesară adaptării la cerințe viitoare. Astfel, integrarea tuturor operațiilor într-un singur modul central este un pas cheie în designul unei unități aritmetice MMX eficiente și ușor de extins.

4.4 Fluxul de Date în Arhitectura Sistemului

Înțelegerea fluxului de date este esențială pentru a aprecia modul în care componentele sistemului MMX colaborează, de la preluarea operanzilor până la afișarea rezultatelor. Arhitectura este concepută pentru a asigura un parcurs coerent al informației, menținând un echilibru între latență, utilizarea resurselor hardware și ușurința în testare.

4.4.1 Descriere generală a fluxului de date

Fluxul de date debutează în **MemoryUnit**, care stochează și furnizează vectorii **A** și **B**. Utilizatorul poate naviga prin diferite seturi de date utilizând butoanele plăcii FPGA (`BtnUp`, `BtnDown`, `BtnCenter`). Indexul curent (`index_out`) indică perechea de operanzi procesată în prezent. Astfel, **MemoryUnit** asigură livrarea datelor brute către **MMX_Unit** în format pe 64 de biți.

MMX_Unit, nucleul arhitecturii, preia acești operanzi și, în funcție de semnalul de control `Control` și dimensiunea specificată (`Size`), execută operația SIMD dorită (adunare, scădere, înmulțire, saturare etc.). Rezultatul, tot pe 64 de biți, este direcționat către **BitSelector**, care permite selectarea părții inferioare sau superioare a vectorului, în funcție de semnalul `SelectBit`.

Datele astfel selectate sunt convertite într-un format afișabil și transmise către **SSD**, modulul responsabil cu afișarea pe 7 segmente. Acest afișaj, alimentat de semnalul de ceas global, oferă feedback vizual instantaneu, permițând utilizatorului să observe rezultatele operațiilor în timp real.

4.4.2 Conexiuni între module

1. **MemoryUnit–MMX_Unit**: **MemoryUnit** livrează vectorii **A** și **B** către **MMX_Unit**. Semnalele de control provenite de la utilizator determină selecția setului de date, iar `index_out` asigură transparență asupra poziției curente din memorie.
2. **MMX_Unit–BitSelector**: După procesarea SIMD, rezultatul de 64 de biți este transmis către **BitSelector**, care extrage segmentul corespunzător (lower sau upper 32 de biți) în funcție de `SelectBit`.
3. **BitSelector–SSD**: Valoarea selectată este trimisă către **SSD**, care o afișează pe 7 segmente. LED-ul indicator semnalează vizual selecția curentă, facilitând interpretarea rapidă a datelor.

4.4.3 Rolul fluxului de date în performanță

Un flux de date coerent și bine organizat minimizează latența și optimizează utilizarea resurselor hardware. Fiecare modul este responsabil pentru o etapă clar definită din lanțul de prelucrare, iar sincronizarea prin semnalul de ceas global garantează consecvența execuției. În plus, arhitectura modulară oferă flexibilitate în testare: fiecare modul poate fi validat independent, simplificând procesul de depanare și îmbunătățire a performanțelor.

Astfel, fluxul de date rezultă într-un sistem echilibrat, eficient și ușor de extins, evidențiind beneficiile abordării SIMD în implementările hardware dedicate.

4.5 Alegerea Designului Arhitectural și Justificări

Deciziile arhitecturale adoptate în acest proiect au fost ghidate de obiectivul principal de a obține un sistem SIMD eficient, flexibil și ușor de extins. Arhitectura modulară, împreună cu selecția platformei FPGA Nexys A7, reflectă un echilibru între performanță, costuri și complexitate.

4.5.1 Principiile și criteriile de selecție

În conceperea designului, au fost avute în vedere următoarele principii:

- **Modularitate și scalabilitate:** Separarea funcționalităților în module distincte (ex. **MMX_Unit**, **MemoryUnit**, **BitSelector**, **SSD**) facilitează înțelegerea structurii, testarea independentă a componentelor și adăugarea ulterioară de instrucțiuni SIMD suplimentare.
- **Performanță și paralelism:** Arhitectura SIMD valorifică execuția paralelă a instrucțiunilor pe mai multe elemente de date, maximizând throughput-ul și minimizând latența. Prin utilizarea hardware dedicat, operațiile devin considerabil mai rapide decât echivalentele software.
- **Optimizarea resurselor hardware:** Implementarea pe FPGA permite ajustarea fină a designului pentru a utiliza eficient LUT-uri, flip-flop-uri și blocuri de memorie, menținând în același timp un nivel înalt de flexibilitate. În plus, FPGA-ul oferă un mediu excelent pentru testare, simulare și depanare.
- **Ușurința testării și depanării:** Structura modulară simplifică identificarea și remedierea erorilor, permițând validarea componentelor în mod individual. De asemenea, accesul la semnale interne și posibilitatea de monitorizare în timp real contribuie la scurtarea ciclului de dezvoltare.
- **Extensibilitate:** Deciziile de design asigură o cale deschisă pentru integrarea unor noi operații, tipuri de date sau optimizări. Astfel, unitatea poate fi adaptată nevoilor specifice ale aplicațiilor viitoare fără a necesita o reproiectare completă.

4.5.2 Justificarea opțiunilor tehnice

Alegerea platformei FPGA Nexys A7 a fost motivată de resursele sale hardware suficiente pentru implementarea operațiilor SIMD, precum și de mediul de dezvoltare robust care suportă simulări precise și tehnici avansate de depanare. Lipsa constrângerilor severe în ceea ce privește consumul de energie sau dimensiunea fizică a circuitului a permis concentrarea pe optimizarea latenței și pe ușurința testării.

În plus, separarea clară a operațiilor SIMD în module individuale (de ex. `PADD`, `PSUB`, `PMULL`, `PADDUS`, `PADDS`, `PSUBS`) a asigurat că fiecare funcționalitate poate fi perfecționată și analizată independent. Această decizie a simplificat mult procesul de testare, depanare și eventuală extindere, menținând designul coerent și gestionabil.

4.5.3 Beneficiile abordării alese

În ansamblu, alegerea designului arhitectural și justificările tehnice asociate reflectă o strategie orientată spre viitor. Arhitectura modulară, implementarea pe FPGA și atenția la paralelizare și saturare asigură un echilibru optim între performanță și flexibilitate. Astfel, proiectul demonstrează nu doar eficiența execuției instrucțiunilor SIMD, ci și capacitatea de a evolua odată cu cerințele aplicațiilor, facilitând integrarea ușoară a noilor funcționalități și optimizări în viitor.

5 Implementare

5.1 Prezentare Generală a Implementării

Implementarea reprezintă etapa crucială în care arhitectura conceptuală a unității MMX este transpusă într-un design hardware concret, pregătit pentru sintetizare și rulare pe FPGA. Obiectivul principal în această fază este de a realiza un sistem coerent, modular și ușor de extins, în care fiecare operație SIMD și fiecare componentă periferică funcționează armonios, conform specificațiilor inițiale.

În această etapă, arhitectura sistemului, descrisă anterior, a fost codificată în limbaj **VHDL**, adoptând o abordare orientată pe module independente. Această strategie de proiectare a permis:

- **Dezvoltare incrementală**: Fiecare modul a fost implementat și testat separat, reducând riscurile de erori și facilitând localizarea rapidă a problemelor.
- **Reutilizare și scalabilitate**: Modulele dedicate pentru operații SIMD (precum `PADD`, `PSUB`, `PADDS`, `PADDUS`, `PSUBS`, `PMUL`) pot fi reutilizate, modificate sau înlocuite cu ușurință. Aceeași flexibilitate se aplică și componentelor periferice (**MemoryUnit**, **BitSelector**, **SSD**).
- **Adaptabilitate la diverse dimensiuni ale datelor**: Dimensiunea operanzilor (8, 16 sau 32 de biți) este selectabilă dinamic, logica interioară ajustându-se automat. Această caracteristică este integrată direct în codul modulelor de adunare, scădere și înmulțire, precum și în componentele pentru saturare.

Platforma **Nexys A7** a oferit mediul practic pentru testare. Datorită accesului la butoane, comutatoare și afișaj, s-a putut evalua interactiv funcționarea sistemului. Astfel, **MemoryUnit** oferă date dintr-un set predefinit, selectate prin butoane, **BitSelector** permite vizualizarea părții dorite din rezultatul pe 64 de biți, iar **SSD** afişează rezultatele sub formă lizibilă. Acest cadru a facilitat depistarea și corectarea erorilor încă din stadiile timpurii.

Din perspectiva codului, s-a optat pentru includerea doar a fragmentelor relevante în documentație, evitând supraîncărcarea cu detalii nenecesare. Fragmentele ilustrative arată cum sunt instanțiate componentele cheie, cum se propagă semnalele de control și cum este gestionat fluxul de date. Acest stil de prezentare asigură înțelegerea logicii interne a sistemului, fără a compromite claritatea documentației.

5.2 Implementarea Operațiilor SIMD în MMX_Unit

Unitatea **MMX_Unit** reprezintă componenta centrală care integrează toate operațiile SIMD discutate anterior. Obiectivul este de a gestiona dinamica selecției instrucțiunilor pe baza semnalelor de control, activând doar blocul corespunzător operației dorite, în funcție de tipul de operație (adunare, scădere, saturare, înmulțire) și dimensiunea datelor (8, 16 sau 32 de biți).

5.2.1 Logica internă a operațiilor

Intern, **MMX_Unit** se bazează pe semnalele `Control` și `Size` pentru a determina care modul de operație SIMD trebuie activat. Semnalul `Control` codifică tipul de operație, în timp ce `Size` precizează dimensiunea datelor (8, 16 sau 32 de biți). Astfel, MMX_Unit instanțiază componentele pentru fiecare operație (`PADD`, `PSUB`, `PADDS`, `PADDUS`, `PSUBS`, `PMULLW`, `PMULHW`) și apoi selectează rezultatul final în mod combinatorial, pe baza acestor semnale.

Logica de selecție este simplă și eficientă: fiecare operație își calculează rezultatul în paralel, dar doar rezultatul relevant este trimis ca ieșire. Astfel, unitatea nu necesită timpi suplimentari de comutare între operații — doar logica de multiplexare determină care rezultat ajunge la ieșire.

În plus, semnalul `Size` influențează modul de funcționare internă a unor module (precum `PADD`, `PSUB`), adaptându-se automat la dimensiunea specificată. De exemplu, pentru 8 biți, vor fi procesate 8 segmente; pentru 16 biți, segmentele vor fi grupate câte două; pentru 32 biți, patru segmente vor fi combinate pentru a forma cuvinte de dimensiunea dorită.

5.2.2 Prezentarea schematică a selecției operațiilor în MMX_Unit

La un nivel conceptual, **MMX_Unit** primește vectorii de intrare **A** și **B**, precum și semnalele `Control` și `Size`. Intern, fiecare operație este instanțiată ca un modul separat, producând un rezultat intermediar. Un multiplexor condiționat prin instrucțiunea `with ... select` alege care dintre aceste rezultate va fi trimis ca ieșire finală.

Această abordare modulară asigură ușurința testării (fiecare operație poate fi testată independent) și scalabilitatea (noi operații pot fi adăugate prin crearea unui nou modul și extinderea instrucțiunii `with ... select`).

Următorul fragment de cod ilustrează cum **MMX_Unit** selectează operația finală pe baza semnalului `Control`, instanțiind mai întâi toate componentele operațiilor, apoi utilizând o instrucțiune `with ... select` pentru alegerea rezultatului final.

```
-- Instanțierea componentelor pentru fiecare operație
U_PADD: PADD port map(A => A, B => B, Size => Size, Result => padd_result);
U_PADDS: PADDS port map(A => A, B => B, Size => Size(0), Result => padds_result);
...

-- Selectarea rezultatului pe baza semnalului Control
with Control select
    Result <= padd_result    when "000", -- PADD
              padds_result   when "001", -- PADDS
              paddus_result  when "010", -- PADDUS
              psub_result    when "011", -- PSUB
              psubs_result   when "100", -- PSUBS
              pmullw_result  when "101", -- PMULLW
              pmulhw_result  when "110", -- PMULHW
              (others => '0') when others; -- Implicit, rezultatul este 0
```

Figura 5: Fragment de cod din *MMX_Unit.vhd*, ilustrând selecția operației pe baza `Control`.

În acest exemplu, observăm cum fiecare operație (`PADD`, `PADDS`, `PADDUS`, `PSUB`, `PSUBS`, `PMULLW`, `PMULHW`) este instanțiată și cum semnalul `Control` determină, prin instrucțiunea `with ... select`, care rezultat este ales la ieșire. Dacă `Control` corespunde, de exemplu, valorii `"000"`, atunci `Result` primește `padd_result`, executând operația de adunare la nivel de byte (`PADD`).

Astfel, **MMX_Unit** asigură un mecanism simplu și coerent pentru configurarea și executarea dinamică a operațiilor SIMD, menținând arhitectura modulară și extensibilă prezentată anterior.

5.2.3 Operațiile de Adunare și Scădere (PADD, PSUB)

Operațiile **PADD** și **PSUB** reprezintă baza aritmeticii vectoriale din unitatea MMX, permițând efectuarea simultană de adunări sau scăderi între elementele corespunzătoare din vectorii **A** și **B**. Ambele operații se adaptează dinamic la dimensiunea datelor (8, 16 sau 32 de

biți) prin semnalul `Size`, și utilizează principiul *wrap-around* pentru a gestiona depășirile de interval.

PADD (Packed Add) Operația `PADD` adună elementele vectorului **A** cu cele ale vectorului **B**, segmentate pe 8, 16 sau 32 de biți. În cazul în care rezultatul unei adunări depășește limita maximă pentru dimensiunea respectivă (de exemplu, 255 pentru 8 biți), valoarea „se întoarce" la 0 (*wrap-around*). Această abordare asigură un comportament previzibil și simplu, util în aplicații multimedia sau filtrare digitală unde suprasaturarea nu este necesară.

```
-- Setăm semnalul Cin pentru fiecare bloc explicit
    Cin_next(0) <= '0'; -- Primul bloc are carry-in fix 0
    Cin_next(1) <= CoutFlags(0) when (Size = "01" or Size = "10") else '0';
    Cin_next(2) <= CoutFlags(1) when (Size = "10") else '0';
    Cin_next(3) <= CoutFlags(2) when (Size = "01" or Size = "10") else '0';
    Cin_next(4) <= '0'; -- Blocurile superioare sunt independente pentru dimensiuni mici
    Cin_next(5) <= CoutFlags(4) when (Size = "01" or Size = "10") else '0';
    Cin_next(6) <= CoutFlags(5) when (Size = "10") else '0';
    Cin_next(7) <= CoutFlags(6) when (Size = "01" or Size = "10") else '0';

    -- Instanțierea explicită a blocurilor AdderSubtractor8bit
    U0: AdderSubtractor8bit port map(A(7 downto 0), B(7 downto 0), Cin_next(0), '0', "00",
    ↪  PartialResults(7 downto 0), CoutFlags(0), open);
    U1: AdderSubtractor8bit port map(A(15 downto 8), B(15 downto 8), Cin_next(1), '0', "00",
    ↪  PartialResults(15 downto 8), CoutFlags(1), open);
    ...

    -- Rezultatul final în funcție de dimensiune
    Result <= PartialResults;
```

Figura 6: Fragment din implementarea `PADD`, evidențiind configurarea adunătorului pe 8 biți pentru wrap-around.

În exemplul de mai sus, fiecare segment de 8 biți este prelucrat printr-un modul `AdderSubtractor8bit`, cu `Op = '0'` (operație de adunare) și `Mode = "00"` (wrap-around nesemnat). Semnalul `Size` determină care segmente sunt active, permițând extinderea operației la 16 sau 32 de biți prin combinarea segmentelor.

PSUB (Packed Subtract) Operația `PSUB` este analogă cu `PADD`, însă în loc să adune elementele vectorului **A** cu cele ale lui **B**, le scade. Similar, dacă rezultatul scăderii coboară sub limita minimă (de exemplu, 0 pentru date nesemnate pe 8 biți), valoarea „se întoarce" la maximul intervalului, recreând efectul de *wrap-around* în sens opus. Această metodă asigură simetria comportamentului față de `PADD` și permite implementarea ușoară a unor algoritmi care necesită aritmetică modulară.

5.2.4 Operațiile cu Saturare (PADDS, PADDUS, PSUBS)

Operațiile cu saturare, precum **PADDS** (Packed Add with Signed Saturation), **PADDUS** (Packed Add with Unsigned Saturation) și **PSUBS** (Packed Subtract with Signed Saturation), asigură că rezultatele aritmetice nu depășesc intervalul de valori admis pentru tipul de date. În cazul adunărilor și scăderilor fără saturare, valoarea rezultată dintr-o depășire (*overflow*) se *înfășoară* înapoi, conform principiului *wrap-around*. Cu saturare, rezultatele sunt plafonate la maximul sau minimul posibil, garantând integritatea numerică.

Principiul saturării pe 16 biți: Să analizăm, spre exemplu, cum logica de saturare pe 16 biți este implementată în **PADDS** (operație care adună elemente pe 8 sau 16 biți, cu saturare semnată). Pentru date pe 16 biți, vectorul de 64 de biți este împărțit în patru perechi a câte 16 biți. Fiecare pereche este formată din două secțiuni de 8 biți care au fost procesate individual. În cazul în care oricare dintre aceste două secțiuni de 8 biți a semnalat *overflow*, întreg cuvântul de 16 biți este setat la valoarea de saturație corespunzătoare (maxim sau minim pe 16 biți semnat).

```
-- Rezultatul final în funcție de dimensiune
    process(PartialResults, OverflowFlags, Size)
    begin
        Result <= PartialResults; -- Implicit, wrap-around
        if Size = '0' then
            -- 8 biți: verificăm overflow pentru fiecare secțiune
            for i in 0 to 7 loop
                if OverflowFlags(i) = '1' then
                    Result((i+1)*8-1 downto i*8) <=
                        "01111111" when PartialResults((i+1)*8-1) = '0' else
                        "10000000";
                end if;
            end loop;
        elsif Size = '1' then
            -- 16 biți: verificăm overflow pentru perechi de secțiuni
            for i in 0 to 3 loop
                if OverflowFlags(2*i) = '1' or OverflowFlags(2*i+1) = '1' then
                    Result((2*i+2)*8-1 downto 2*i*8) <=
                        "0111111111111111" when PartialResults((2*i+2)*8-1) = '0' else
                        "1000000000000000";
                end if;
            end loop;
        end if;
    end process;
```

Figura 7: Fragment de cod din `PADDS` care tratează overflow-ul pe 16 biți.

În fragmentul de cod de mai sus, pentru **Size = '1'** (16 biți):

- Se verifică fiecare pereche de 8 biți (`OverflowFlags(2*i)` și `OverflowFlags(2*i+1)`) pentru a detecta overflow.
- Dacă apare overflow în oricare dintre cele două secțiuni de 8 biți, rezultatul pe 16 biți este reîncadrat la valoarea de saturație:
  - `"0111111111111111"` pentru saturație pozitivă (maxim: +32767 pentru semnat pe 16 biți).
  - `"1000000000000000"` pentru saturație negativă (minim: -32768 pentru semnat pe 16 biți).

Această logică asigură că rezultatele depășite sunt plafonate la limitele intervalului reprezentabil, evitând erori numerice care pot apărea dacă s-ar fi utilizat doar wrap-around.

Rolul saturării: Saturarea este esențială în aplicații unde overflow-ul nu este dorit, cum ar fi prelucrarea semnalelor audio sau imagini. În astfel de domenii, este preferabil ca un pixel

sau un eșantion audio să atingă valoarea maximă posibilă în loc să se răstoarne la o valoare mică, rezultând artefacte vizuale sau sonore nedorite. Operând direct în hardware, saturarea îmbunătățește calitatea și stabilitatea rezultatelor fără a fi necesare calcule suplimentare în software.

Astfel, **PADDS**, **PADDUS** și **PSUBS** reprezintă o extensie naturală a operațiilor de bază **PADD** și **PSUB**, adăugând control precis asupra depășirilor și menținând integritatea numerică în aplicații critice.

5.2.5 Operațiile de Înmulțire (PMULLW, PMULHW)

Operațiile de înmulțire implementate în cadrul **MMX_Unit** includ **PMULLW** (Packed Multiply Low Word) și **PMULHW** (Packed Multiply High Word). Ambele operează pe perechi de word-uri (16 biți) din vectorii de intrare, însă diferă în privința părții din rezultatul de 32 de biți care este selectată:

- **PMULLW (Mode = '0')**: Păstrează partea inferioară (low word) a fiecărui produs de 32 de biți, potrivită pentru calcule ce necesită doar acuratețe limitată.
- **PMULHW (Mode = '1')**: Păstrează partea superioară (high word) a rezultatului, utilă atunci când sunt necesare valorile cu magnitudini mai mari sau componentele de ordin superior.

Selecția dintre **PMULLW** și **PMULHW** se face prin semnalul `Mode`, astfel încât aceeași infrastructură de multiplicare poate fi reutilizată pentru ambele tipuri de operații. Fragmentul de cod de mai jos ilustrează logica de selecție a părții inferioare sau superioare din rezultatele fiecărui multiplicator pe 16 biți:

```
1   -- Selectăm partea inferioară sau superioară din fiecare produs pe baza semnalului Mode
2   Prod <= (prod3(31 downto 16) & prod2(31 downto 16) & prod1(31 downto 16) & prod0(31 downto
    ↪  16)) when Mode = '1' else
3       (prod3(15 downto 0) & prod2(15 downto 0) & prod1(15 downto 0) & prod0(15 downto 0));
```

Figura 8: Fragment de cod din modulul `PMUL` care selectează partea inferioară (PMULLW) sau superioară (PMULHW) a produsului.

Astfel, printr-un simplu bit de control (`Mode`), întregul flux de multiplicare devine versatil, permițând optimizarea resurselor hardware și adaptarea la nevoile specifice ale aplicației, fără a modifica arhitectura sau logica internă a multiplicatorului.

5.3 Modulele de Bază: AdderSubtractor8bit și Multiplier8bit

5.3.1 AdderSubtractor8bit și AdderSubtractor1bit

Operațiile de adunare și scădere la nivel de biți constituie fundamentul aritmetic al unității MMX. Aceste operații sunt implementate prin module de bază `AdderSubtractor1bit` și `AdderSubtractor8bit`, care permit manipularea eficientă a datelor vectoriale pe diferite dimensiuni (8, 16 sau 32 de biți). În această secțiune, ne concentrăm pe modulul `AdderSubtractor8bit`, evidențiind modul în care acesta integrează opt blocuri `AdderSubtractor1bit` și gestionează detectarea overflow-ului.

Structura `AdderSubtractor8bit` Modulul `AdderSubtractor8bit` este construit din opt instanțieri ale modulului `AdderSubtractor1bit`, fiecare reprezentând un bit al operației aritmetice. Acest

design modular facilitează gestionarea operațiilor pe multiple biți și permite propagarea corectă a carry-ului sau borrow-ului între biți.

```
U0: AdderSubtractor1bit port map(A(0), B(0), Cin, Op, Ext_Result(0), Carry(0));
U1: AdderSubtractor1bit port map(A(1), B(1), Carry(0), Op, Ext_Result(1), Carry(1));
-- ...
U7: AdderSubtractor1bit port map(A(7), B(7), Carry(6), Op, Ext_Result(7), Carry(7));
Ext_Result(8) <= Carry(7); -- Extinderea rezultatului pentru overflow
```

Figura 9: Fragment de cod din `AdderSubtractor8bit.vhd`, ilustrând instanțierea blocurilor `AdderSubtractor1bit` și gestionarea overflow-ului.

În fragmentul de cod de mai sus, observăm următoarele componente cheie:

- **Instanțierea Blocurilor** `AdderSubtractor1bit`: Fiecare instanțiere (`U0` până la `U7`) reprezintă un bit al operației de adunare sau scădere. Semnalul `Carry` este propagat de la un bloc la altul pentru a asigura corectitudinea rezultatelor.
- **Ext_Result**: Acest semnal extinde rezultatul la 9 biți, permițând detectarea overflow-ului la nivel de întreg cuvânt.
- **Detectarea Overflow-ului**: Semnalele `Overflow_Pos` și `Overflow_Neg` sunt utilizate pentru a determina dacă a avut loc un overflow pozitiv sau negativ, în funcție de operația efectuată.
- **Calculul Rezultatului Final**: Rezultatul este ajustat în funcție de modul de operare (`Mode`), aplicând logica de wrap-around sau saturare.

Logica de Detectare a Overflow-ului Detectarea overflow-ului este esențială pentru a asigura integritatea rezultatelor aritmetice, mai ales în contextul operațiilor cu saturare. Modulul `AdderSubtractor8bit` verifică semnalele de overflow pozitiv și negativ și ajustează rezultatul în consecință.

```
Overflow_Pos <= '1' WHEN (A(7) = '0' AND B(7) = '0' AND Ext_Result(7) = '1') ELSE
                '1' WHEN (A(7) = '0' AND B(7) = '1' AND Op = '1' AND Ext_Result(7) = '1') ELSE
                ↪  '0';
Overflow_Neg <= '1' WHEN (A(7) = '1' AND B(7) = '1' AND Ext_Result(7) = '0') ELSE
                '1' WHEN (A(7) = '1' AND B(7) = '0' AND Op = '1' AND Ext_Result(7) = '0') ELSE
                ↪  '0';
Overflow <= Overflow_Pos OR Overflow_Neg;
```

Figura 10: Fragment de cod pentru detectarea overflow-ului în `AdderSubtractor8bit.vhd`.

În fragmentul de cod de mai sus, se observă cum semnalele `Overflow_Pos` și `Overflow_Neg` sunt calculate în funcție de semnele operandilor și ale rezultatului. Aceste semnale sunt apoi combinate pentru a determina semnalul final `Overflow`.

Gestionarea Modului de Operare (`Mode`) Semnalul `Mode` determină modul în care rezultatul final este calculat, aplicând fie wrap-around, fie saturare, în funcție de necesități. Aceasta permite flexibilitatea operațiilor aritmetice, adaptându-se la diferite scenarii de prelucrare a datelor.

```
result_config_mode.vhdResult <=
    Ext_Result(7 downto 0) WHEN (Mode = "00") ELSE
    "11111111" WHEN (Mode = "10" AND Ext_Result(8) = '1' AND Op = '0') ELSE
    "00000000" WHEN (Mode = "10" AND Ext_Result(8) = '1' AND Op = '1') ELSE
    "01111111" WHEN (Mode = "11" AND Overflow_Pos = '1') ELSE
    "10000000" WHEN (Mode = "11" AND Overflow_Neg = '1') ELSE
    Ext_Result(7 downto 0);
```

Figura 11: Fragment de cod pentru configurarea rezultatului final în funcție de modul de operare în `AdderSubtractor8bit.vhd`.

În fragmentul de cod de mai sus, rezultatul este ajustat în funcție de semnalul `Mode`. În cazul wrap-around, rezultatul este pur și simplu extins, în timp ce în cazul saturării, rezultatul este limitat la valorile maxime sau minime posibile în funcție de semnul overflow-ului detectat.

Structura `AdderSubtractor1bit` Modulul `AdderSubtractor1bit` este componenta elementară care efectuează operații de adunare sau scădere pe un singur bit. Acesta calculează rezultatul individual și propagatează carry-ul sau borrow-ul în funcție de operația selectată.

```
Result <= A XOR B XOR Cin; -- Rezultatul pentru fiecare bit
Carry <= (A AND B) OR (B AND Cin) OR (Cin AND A); -- Carry pentru adunare
Borrow <= (NOT A AND B) OR (B AND Cin) OR (Cin AND NOT A); -- Borrow pentru scădere
Cout <= Carry WHEN Op = '0' ELSE Borrow; -- Propagarea Cout
```

Figura 12: Fragment de cod din `AdderSubtractor1bit.vhd`, ilustrând calculul carry-ului și borrow-ului și selecția finală a `Cout`.

În fragmentul de cod de mai sus, se evidențiază următoarele aspecte:

- **Calculul Rezultatului**: Rezultatul este obținut prin operația XOR între biții `A`, `B` și `Cin`, reprezentând suma sau diferența bitului curent.
- **Calculul `Carry` și `Borrow`**:
  - `Carry`: Calculat pentru operația de adunare, determinând dacă se propagă un carry către bitul următor.
  - `Borrow`: Calculat pentru operația de scădere, determinând dacă se propagă un borrow către bitul următor.
- **Selecția `Cout`**: În funcție de operația selectată (`Op` = '0' pentru adunare și `Op` = '1' pentru scădere), semnalul `Cout` este atribuit fie carry-ului, fie borrow-ului.

Prin combinarea a opt blocuri `AdderSubtractor1bit` într-un singur modul `AdderSubtractor8bit`, se facilitează realizarea de operații aritmetice complexe pe multiple biți, asigurând în același timp gestionarea corectă a overflow-ului și a propagării semnalelor de carry sau borrow.

5.3.2 Multiplier8bit și Multiplier16bit

Multiplicarea este o operație fundamentală în multe aplicații SIMD, inclusiv procesarea semnalelor și a imaginilor. În cadrul unității MMX, operațiile de înmulțire sunt implementate prin modulele `Multiplier8bit` și `Multiplier16bit`. În această secțiune, ne vom concentra pe `Multiplier8bit`, explicând modul în care acesta generează produsele parțiale și utilizează adder-ele elementare pentru a construi rezultatul final. De asemenea, vom oferi o explicație detaliată a modulului `Multiplier16bit`, concentrându-ne pe logica sa internă.

Structura `Multiplier8bit` Modulul `Multiplier8bit` este responsabil pentru înmulțirea a doi vectori de 8 biți, generând un produs de 16 biți. Acest modul se bazează pe generarea produselor parțiale prin operații AND între biții individuali ai vectorilor de intrare și apoi adunarea acestor produse parțiale folosind modulele `AdderSubtractor1bit`. Această abordare modulară permite construirea eficientă a multiplicatorului prin combinarea unor operații aritmetice simple.

Generarea Produselor Parțiale Produsele parțiale sunt generate prin operații AND între fiecare bit al vectorilor de intrare `A` și `B`. Aceste operații creează o matrice de produse parțiale care sunt ulterior adunate pentru a forma produsul final.

```
p0(0) <= B(0) and A(0);
p0(1) <= B(0) and A(1);
p0(2) <= B(0) and A(2);
-- and so on for all bits
```

Figura 13: Fragment de cod din `Multiplier8bit.vhd`, ilustrând generarea produselor parțiale prin operații AND.

În fragmentul de cod din Figura 13, fiecare bit al vectorului `B` este înmulțit cu întregul vector `A` utilizând operația AND. De exemplu, `p0(0)` reprezintă `B(0)` AND `A(0)`, `p0(1)` reprezintă `B(0)` AND `A(1)`, și așa mai departe pentru toți biții.

Sumarea Produselor Parțiale După generarea produselor parțiale, acestea sunt adunate utilizând modulele `AdderSubtractor1bit`. Fiecare instanțiere a modulului `AdderSubtractor1bit` primește doi biți (sau un bit și un carry) și generează un rezultat și un carry. Acest proces se repetă pe parcursul celor opt biți, asigurând că toate produsele parțiale sunt adunate corect și că carry-urile sunt propagate corespunzător.

```
-- First Addition Step
f11: AdderSubtractor1bit port map(p0(0), '0', '0', '0', s11, c11);
f12: AdderSubtractor1bit port map(p0(1), p1(0), c11, '0', s12, c12);
...
f19: AdderSubtractor1bit port map('0', p1(7), c18, '0', s19, c19);

-- Subsequent Addition Steps (Second to Seventh)
-- Similar instantiations of AdderSubtractor1bit for each step
```

Figura 14: Fragment de cod din `Multiplier8bit.vhd`, ilustrând utilizarea modulelor `AdderSubtractor1bit` pentru sumarea produselor parțiale.

În fragmentul de cod din Figura 14, fiecare instanțiere a modulului `AdderSubtractor1bit` (`f11` până la `f19`) combină doi biți ai produselor parțiale și propagă carry-ul către următorul bit. Acest proces se repetă pentru fiecare linie de adunare, asigurând că rezultatul final este corect calculat.

Asamblarea Produsului Final După efectuarea tuturor pașilor de adunare, semnalele de sumă și carry sunt combinate pentru a forma produsul final de 16 biți.

```
-- Assigning the final summed signals to the product output
p(0) <= s71;
p(1) <= s72;
...
p(15) <= c715;
```

Figura 15: Fragment de cod din `Multiplier8bit.vhd`, ilustrând asamblarea produsului final prin atribuirea semnalelor de sumă și carry.

În fragmentul de cod din Figura 15, rezultatele sumelor intermediare (`s71` până la `s715`) și carry-urile finale (`c715`) sunt atribuite vectorului de ieșire `P`, formând astfel produsul final de 16 biți al multiplicatorului pe 8 biți.

Structura `Multiplier16bit` Modulul `Multiplier16bit` extinde capacitatea multiplicatorului pe 8 biți, permițând înmulțirea a doi vectori de 16 biți pentru a obține un produs de 32 de biți. Acest modul combină două instanțieri ale `Multiplier8bit` pentru a gestiona fiecare pereche de 8 biți din vectorii de intrare. Rezultatele parțiale sunt apoi shiftate și adunate pentru a forma produsul final pe 32 de biți.

Gestionarea Semnului Semnul rezultatului este determinat prin XOR-ul semnelor operandilor. Dacă semnele operandilor diferă, rezultatul final va fi negativ, altfel va fi pozitiv.

```
Sign <= A(15) xor B(15);

process(A, B)
begin
    if A(15) = '1' then
        Abs_A <= std_logic_vector(-signed(A));
    else
        Abs_A <= A;
    end if;

    if B(15) = '1' then
        Abs_B <= std_logic_vector(-signed(B));
    else
        Abs_B <= B;
    end if;
end process;
```

Figura 16: Fragment de cod din `Multiplier16bit.vhd`, ilustrând gestionarea semnului rezultatului prin XOR-ul semnelor operandilor.

În fragmentul de cod din Figura 16, semnul rezultatului (`Sign`) este determinat prin XOR-ul semnelor vectorilor de intrare `A` și `B`. Ulterior, valorile absolute ale operandilor sunt calculate pentru a simplifica procesul de multiplicare.

Combinarea Produselor Parțiale Produsele parțiale generate de instanțierile `Multiplier8bit` sunt combinate prin shift-uri și adunări pentru a obține produsul final de 32 de biți.

```
sum_part := LL_ext + (LH_ext sll 8) + (HL_ext sll 8) + (HH_ext sll 16);

if Sign = '1' then
    sum_part_signed := -signed(sum_part);
else
    sum_part_signed := signed(sum_part);
end if;
```

Figura 17: Fragment de cod din `Multiplier16bit.vhd`, ilustrând combinarea produselor parțiale prin shift-uri și adunări.

În fragmentul de cod din Figura 17, produsele parțiale (`LL`, `LH`, `HL`, `HH`) sunt shiftate corespunzător și adunate pentru a forma produsul final. În funcție de semnul rezultatului, produsul este ajustat pentru a reflecta corectitatea semnului.

Selectorul de Mod (`Mode`) Semnalul `Mode` determină dacă se va extrage partea joasă sau înaltă a produsului final, utilizând `PMULLW` sau `PMULHW` respectiv.

```
if Mode = '0' then
    -- PMULLW: lowest 16 bits
    Result <= std_logic_vector(sum_part_signed(15 downto 0));
else
    -- PMULHW: highest 16 bits
    Result <= std_logic_vector(sum_part_signed(31 downto 16));
end if;
```

Figura 18: Fragment de cod din `Multiplier16bit.vhd`, ilustrând selecția părții finale a produsului în funcție de modul de operare.

În fragmentul de cod din Figura 18, în funcție de valoarea semnalului `Mode`, se selectează fie partea joasă (lowest 16 biți) a produsului final (`PMULLW`), fie partea înaltă (highest 16 biți) (`PMULHW`).

Fluxul Operațional al Multiplicatorului

1. **Separarea Operandilor**: Vectorii de intrare `A` și `B` sunt împărțiți în părți inferioare (`A_low`, `B_low`) și superioare (`A_high`, `B_high`).
2. **Generarea Produselor Parțiale**: Fiecare pereche de 8 biți este multiplicată folosind instanțierile `Multiplier8bit`, generând patru produse parțiale (`LL`, `LH`, `HL`, `HH`).
3. **Combinarea Produselor**: Produsele parțiale sunt shiftate și adunate pentru a obține produsul final de 32 de biți.
4. **Aplicarea Semnului**: Produsul final este ajustat în funcție de semnul rezultatului înmulțirii operandilor.
5. **Selecția Rezultatului Final**: În funcție de semnalul `Mode`, se extrag fie cele mai puține 16 biți, fie cei mai înalți 16 biți ai produsului final.

Prin această abordare, unitatea MMX poate efectua multiplicări eficiente pe 8 și 16 biți, gestionând corect semnele și asigurând rezultate precise și consistente în aplicații diverse.

5.4 Componente Periferice

5.4.1 MemoryUnit

Modulul `MemoryUnit` este responsabil pentru stocarea și accesarea perechilor de vectori de 64 de biți (`A` și `B`) utilizați în operațiile SIMD. Acesta permite navigarea printr-un set predefinit de perechi de vectori folosind butoanele de incrementare (`BtnUp`), decrementare (`BtnDown`) și resetare (`BtnCenter`).

Structura `MemoryUnit` Modulul `MemoryUnit` este structurat pentru a gestiona eficient stocarea vectorilor de intrare și pentru a facilita accesul rapid la diferitele perechi de vectori necesare pentru operațiile SIMD. Structura memoriei este definită utilizând tipuri 'record' și 'array', permițând organizarea logică și accesarea eficientă a datelor.

```
type memory_element is record
    vecA : STD_LOGIC_VECTOR(63 downto 0); -- Vector A
    vecB : STD_LOGIC_VECTOR(63 downto 0); -- Vector B
end record;

type memory_type is array (0 to MEM_DEPTH-1) of memory_element;

signal memory : memory_type := (
    -- Predefined memory content here
);
```

Figura 19: Definirea structurii memoriei în `MemoryUnit.vhd`, utilizând tipuri `record` și `array`.

În fragmentul de cod din Figura 19, se definește structura memoriei utilizând un 'record' pentru a reprezenta fiecare element de memorie (`memory_element`), care conține doi vectori de 64 de biți (`vecA` și `vecB`). Apoi, este definit un tip de memorie (`memory_type`) ca un array de astfel de elemente, permițând stocarea unui număr specificat de perechi de vectori.

Inițializarea Memoriei Memoria este inițializată cu perechi de vectori predefinite, fiecare corespunzând unei operații specifice, cum ar fi `PADDS`, `PADDUS`, `PSUBS`, etc. Această inițializare permite testarea și validarea operațiilor SIMD fără a fi necesară încărcarea datelor din exterior.

Gestionarea Indexului Curent Indexul curent (`index`) este gestionat prin intermediul unui proces sincronizat cu semnalul de ceas (`clk`). Utilizatorul poate modifica indexul folosind butoanele `BtnUp`, `BtnDown` și `BtnCenter`, care permit incrementarea, decrementarea sau resetarea indexului la poziția 0.

```vhdl
process(clk)
begin
    if rising_edge(clk) then
        if BtnCenter = '1' then
            index <= 0; -- Reset to position 0
        elsif BtnUp = '1' then
            if index < MEM_DEPTH-1 then
                index <= index + 1; -- Increment index
            end if;
        elsif BtnDown = '1' then
            if index > 0 then
                index <= index - 1; -- Decrement index
            end if;
        end if;
    end if;
end process;
```

Figura 20: Logica de gestionare a indexului în `MemoryUnit.vhd`, utilizând butoanele de control.

În fragmentul de cod din Figura 20, procesul sincronizat cu `clk` răspunde la semnalele de la butoane pentru a modifica indexul curent:

- `BtnCenter`: Resetează indexul la 0, revenind la prima pereche de vectori din memorie.
- `BtnUp`: Incrementează indexul, navigând spre următoarea pereche de vectori, dacă nu s-a atins limita superioară (`MEM_DEPTH-1`).
- `BtnDown`: Decrementează indexul, navigând spre perechea anterioară de vectori, dacă nu s-a atins limita inferioară (0).

Această logică asigură o navigare fluidă și controlată prin setul de date stocate, prevenind depășirile de limite și menținând integritatea accesului la memorie.

Atribuirea Ieșirilor Vectorilor și a Indexului După actualizarea indexului, vectorii `A` și `B` sunt direcționați din memoria corespunzătoare elementului curent. Indexul este, de asemenea, convertit într-un format binar de 10 biți și atribuit semnalului `index_out`, care poate fi utilizat pentru afișaje sau alte scopuri de monitorizare.

```vhdl
A <= memory(index).vecA; -- Current vector A output
B <= memory(index).vecB; -- Current vector B output
index_out <= std_logic_vector(to_unsigned(index, 10)); -- Binary index output
```

Figura 21: Atribuirea ieșirilor vectorilor `A` și `B`, precum și a indexului curent în `MemoryUnit.vhd`.

În fragmentul de cod din Figura 21, vectorii `A` și `B` sunt actualizați în funcție de indexul curent, extrăgând perechea de vectori corespunzătoare din memorie. Indexul curent este convertit într-un format binar adecvat și atribuit semnalului `index_out`, care poate fi utilizat pentru afișaje pe ecran sau pentru alte operațiuni de monitorizare.

Fluxul Operațional al `MemoryUnit`

1. **Inițializarea Memoriei**: Vectorii `A` și `B` sunt stocați în memorie în perechi predefinite.
2. **Gestionarea Indexului**: Utilizatorul utilizează butoanele `BtnUp`, `BtnDown` și `BtnCenter` pentru a naviga prin perechile de vectori.
3. **Actualizarea Ieșirilor**: Vectorii `A` și `B` sunt actualizați în funcție de indexul curent, iar indexul este convertit și transmis prin semnalul `index_out`.

Prin această structură, `MemoryUnit` asigură o gestionare eficientă a datelor vectoriale necesare pentru operațiile SIMD, facilitând accesul rapid și controlat la diferitele perechi de vectori stocate.

5.4.2 BitSelector

Modulul `BitSelector` joacă un rol esențial în gestionarea și afișarea rezultatelor operațiilor SIMD. Acesta permite selectarea unei porțiuni specifice a vectorului de 64 de biți rezultat (`Result`) și direcționarea acesteia către afișajul pe 7 segmente (`SSD`). Funcționalitatea principală a modulului constă în alegerea între partea inferioară (lower 32 de biți) sau superioară (upper 32 de biți) a rezultatului, în funcție de semnalul de control `SelectBit`.

Structura `BitSelector` Modulul `BitSelector` primește vectorul de 64 de biți `Result` și un semnal de selecție `SelectBit`. În funcție de valoarea lui `SelectBit`, modulul alege fie partea inferioară, fie partea superioară a vectorului `Result`, direcționând astfel biții selectați către ieșirea `SelectedBits`. De asemenea, semnalul `SelectBit` este indicat prin intermediul unui LED (`IndicatorLED`) pentru a oferi feedback vizual utilizatorului.

```
1  architecture Behavioral of BitSelector is
2  begin
3      -- Selectăm partea relevantă a vectorului în funcție de SelectBit
4      SelectedBits <= Result(31 downto 0) when SelectBit = '0' else Result(63 downto
       ↪  32);
5
6      -- Indicăm starea lui SelectBit pe LED
7      IndicatorLED <= SelectBit;
8  end Behavioral;
```

Figura 22: Fragment de cod din `BitSelector.vhd`, ilustrând logica de selecție a bitilor și indicarea stării prin LED.

În fragmentul de cod de mai sus, se evidențiază următoarele componente cheie:

- **Selecția Bitilor:** Instrucțiunea `when-else` este utilizată pentru a alege între `Result(31 downto 0)` și `Result(63 downto 32)` pe baza valorii semnalului `SelectBit`. Această logică permite alternarea rapidă între cele două părți ale rezultatului fără a necesita componente suplimentare sau timpi de comutare.
- **Indicator LED:** Semnalul `SelectBit` este direct atribuit ieșirii `IndicatorLED`, oferind astfel un feedback vizual utilizatorului despre care parte a rezultatului este afișată în prezent. Acest lucru este util pentru navigarea și verificarea rezultatelor operațiilor SIMD în timp real.

Logica de Seleție a Bitilor Logica internă a modulului `BitSelector` este simplă și eficientă. Utilizând o instrucțiune `when-else`, modulul decide care parte a vectorului `Result` va fi transmisă la ieșirea `SelectedBits`:

- `SelectBit = '0'`: Se selectează partea inferioară (`Result(31 downto 0)`).
- `SelectBit = '1'`: Se selectează partea superioară (`Result(63 downto 32)`).

Această abordare permite alternarea rapidă și flexibilă între cele două porțiuni ale rezultatului, facilitând afișarea eficientă a datelor pe afișajul pe 7 segmente.

Indicator LED Indicatorul LED oferă un feedback vizual asupra stării selecției. Atunci când `SelectBit` este setat la '0', LED-ul indică faptul că partea inferioară a rezultatului este activă. În schimb, când `SelectBit` este setat la '1', LED-ul indică activarea părții superioare. Această caracteristică îmbunătățește interacțiunea utilizatorului cu sistemul, permițându-i să înțeleagă rapid ce parte a rezultatului este afișată.

5.4.3 SSD (Afișaj 7-segmente)

Modulul `SSD` este responsabil pentru afișarea rezultatelor operațiilor SIMD pe un afișaj pe 7 segmente. Acesta convertește valorile binare din vectorul `digits` într-un format compatibil cu afișajul, gestionând multiplexarea și decodificarea cifrelor pentru afișarea corectă a acestora. Modulul utilizează un contor intern pentru a alterna între diferitele cifre afișate, asigurând o actualizare rapidă și eficientă a afișajului.

Structura `SSD` Modulul `SSD` primește un vector de 32 de biți (`digits`) care reprezintă patru cifre hexazecimale de afișat. Utilizând un contor (`cnt`), modulul multiplexează afișarea fiecărei cifre pe cele opt anode (`an`) ale afișajului pe 7 segmente (`cat`). Astfel, fiecare cifră este afișată rapid și în mod alternant, creând iluzia unui afișaj continuu.

```
signal cnt : STD_LOGIC_VECTOR(16 downto 0) := (others => '0'); -- 17-bit counter
signal sel : STD_LOGIC_VECTOR(2 downto 0);                     -- 3-bit selector

counter: process(clk)
begin
    if rising_edge(clk) then
        cnt <= cnt + 1; -- Increment counter every clock cycle
    end if;
end process;

sel <= cnt(16 downto 14); -- Select digit based on MSBs of counter
```

Figura 23: Fragment de cod din `SSD.vhd`, ilustrând logica de multiplexare a cifrelor prin contorul intern.

Logica de Multiplexare În fragmentul de cod din Figura 23, semnalul `cnt` este utilizat ca contor intern sincronizat cu semnalul de ceas (`clk`). La fiecare ciclu de ceas, contorul este incrementat, iar biții superiori (`cnt(16 downto 14)`) sunt utilizați pentru a selecta cifra curentă care trebuie afișată.

```vhdl
muxCat: process(sel, digits)
begin
    case sel is
        when "000" => digit <= digits(3 downto 0);   -- First digit
        when "001" => digit <= digits(7 downto 4);   -- Second digit
        -- ...
        when "111" => digit <= digits(31 downto 28); -- Eighth digit
        when others => digit <= (others => 'X');
    end case;
end process;
```

Figura 24: Fragment de cod din `SSD.vhd`, ilustrând selecția cifrei curente bazată pe semnalul `sel`.

Selecția Cifrei În fragmentul de cod din Figura 24, procesul `muxCat` utilizează un `case` bazat pe semnalul `sel` pentru a alege ce subset de 4 biți din vectorul `digits` va fi afișat. Fiecare caz corespunde unei cifre specifice, extrăgând un subset de 4 biți din `digits` și atribuindu-l semnalului `digit`.

```vhdl
muxAn: process(sel)
begin
    case sel is
        when "000" => an <= "11111110";
        when "001" => an <= "11111101";
        -- ...
        when "111" => an <= "01111111";
        when others => an <= (others => 'X');
    end case;
end process;
```

Figura 25: Fragment de cod din `SSD.vhd`, ilustrând activarea anodei corespunzătoare cifrei selectate.

Activarea Anodului În fragmentul de cod din Figura 25, procesul `muxAn` activează unul dintre cele opt anode (`an`) ale afișajului pe 7 segmente, în funcție de valoarea semnalului `sel`. Fiecare anod reprezintă o cifră specifică, permițând afișarea rapidă a cifrelor într-un ciclu continuu.

```vhdl
with digit select
    cat <= "1000000" when "0000",  -- 0
           "1111001" when "0001",  -- 1
           "0100100" when "0010",  -- 2
           "0000000" when "1000",  -- 8
           "0001110" when "1111",  -- F
           (others => 'X') when others; -- Default case
```

Figura 26: Fragment de cod din `SSD.vhd`, ilustrând decodificarea cifrei pentru afișajul pe 7 segmente.

Decodificarea Cifrei pentru Afișajul pe 7 Segmente În fragmentul de cod din Figura 26, instrucțiunea `with ... select` convertește valoarea binară a cifrei (`digit`) în codul corespunzător pentru afișajul pe 7 segmente (`cat`). Fiecare valoare a lui `digit` este asociată cu un șir

specific de segmente active pentru a reprezenta corect cifra respectivă. De exemplu, valoarea binară `"0000"` este decodificată pentru a afișa cifra 0, iar `"1111"` este decodificată pentru a afișa cifra F.

Funcționarea `SSD` 1. **Contorul de Multiplexare**: La fiecare ciclu de ceas, contorul `cnt` este incrementat, iar biții superiori (`sel`) determină care cifră este selectată pentru afișare. 2. **Selecția Cifrei**: În funcție de valoarea lui `sel`, procesul `muxCat` extrage un subset de 4 biți din `digits` și îl atribuie semnalului `digit`. 3. **Activarea Anodului**: Procesul `muxAn` activează unul dintre cele opt anode ale afișajului, corespunzând cifrei selectate. 4. **Decodificarea pentru Afișaj**: Instrucțiunea `with ... select` convertește valoarea binară a cifrei în codul specific pentru afișajul pe 7 segmente, activând segmentele corespunzătoare pentru a afișa cifra corectă.

Fluxul Operațional al `SSD`

1. **Generarea Produselor Parțiale**: Semnalul `cnt` este incrementat la fiecare ciclu de ceas, determinând selecția cifrei curente.
2. **Selecția Cifrei**: Procesul `muxCat` extrage subsetul corespunzător de biți din `digits` pentru cifra care urmează să fie afișată.
3. **Activarea Anodului**: Procesul `muxAn` activează anodul specific cifrei selectate.
4. **Decodificarea Cifrei**: Valorile binare ale cifrei sunt convertite în coduri specifice pentru afișajul pe 7 segmente, activând segmentele necesare pentru a afișa corect cifra.

5.4.4 MPG (Debounce Butoane)

Modulul `MPG` este esențial pentru gestionarea semnalelor de la butoanele de control (`btn`), asigurând o funcționare fiabilă prin eliminarea efectelor de *bounce*. Debouncing-ul este o tehnică utilizată pentru a preveni multiplele semnale de activare generate atunci când un buton este apăsat sau eliberat, rezultând astfel în citiri stabile și precise ale intrărilor.

Structura `MPG` Modulul `MPG` implementează un mecanism de debouncing utilizând un contor intern și trei registre de tip `flip-flop` (`Q1`, `Q2`, `Q3`) pentru a filtra zgomotele și fluctuațiile rapide ale semnalului de la buton.

Sunt evidențiate următoarele componente cheie:

- **Contorul Intern (`cnt_int`):**
  - Este definit ca un vector de 18 biți, inițializat la zero.
  - Acest contor este incrementat la fiecare ciclu de ceas (`clk`) în cadrul procesului sincronizat.
  - Rolul său este de a crea o perioadă de întârziere suficientă pentru a permite stabilizarea semnalului de la buton după o apăsare.
- **Registrii de Tip `Flip-Flop` (`Q1`, `Q2`, `Q3`):**
  - Acești regiști sunt utilizați pentru a filtra semnalul de intrare (`btn`) și a elimina fluctuațiile rapide (*bounce*).
  - Semnalul `Q1` captează starea butonului după ce contorul ajunge la valoarea maximă (`cnt_int` = "11111111111111111").
  - Semnalele `Q2` și `Q3` formează o secvență de întârziere, asigurând că semnalul de ieșire (`enable`) este activat doar atunci când butonul a fost stabil apăsat pentru o perioadă predefinită.
- **Logica de Generare a Semnalului `enable`:**
  - `enable` este definit ca `Q2 AND (NOT Q3)`.

- Această logică detectează o tranziție de la o stare scăzută la una înaltă a semnalului filtrat, indicând o apăsare stabilă a butonului.

- **Procesul de Sincronizare:**
  - Toate procesele sunt sincronizate cu semnalul de ceas (`clk`) și se declanșează la frontul ascendent al acestuia (`rising_edge(clk)`).
  - Acest lucru asigură că actualizările semnalelor interne și a contorului sunt realizate într-un mod predictibil și controlat.

Funcționarea `MPG` 1. Incrementarea Contorului Intern: - La fiecare ciclu de ceas, contorul `cnt_int` este incrementat, avansând spre valoarea maximă ("1111111111111111"). - Odată ce contorul ajunge la această valoare, semnalul de intrare (`btn`) este capturat în `Q1`.

2. Filtrarea Semnalelor de Intrare: - Semnalele `Q1`, `Q2` și `Q3` formează o secvență de `flip-flop`-uri care întârzie și stabilizează semnalul capturat. - Aceasta asigură că orice fluctuații rapide ale semnalului de la buton sunt eliminate înainte de a fi utilizate în logica de control.

3. Generarea Semnalului `enable`: - Semnalul `enable` este activat doar atunci când `Q2` este '1' și `Q3` este '0', indicând o tranziție stabilă a semnalului de la buton. - Acest semnal este apoi utilizat pentru a controla diverse componente din sistem, asigurând că acțiunile sunt declanșate doar după o apăsare confirmată a butonului.

5.5 Integrarea Completă (test_env)

Secțiunea `test_env` reprezintă nivelul superior al sistemului, integrând toate componentele periferice și unitatea centrală `MMX_Unit`. Scopul acestui modul este de a coordona interacțiunea între butoane, comutatoare, memorie, operații SIMD și afișajul pe 7 segmente, asigurând o funcționare fluidă și sincronizată a întregului sistem.

Structura `test_env` Modulul `test_env` instanțiază și conectează toate componentele principale ale sistemului, gestionând semnalele de control și fluxul de date între acestea. Fragmentul de cod de mai jos ilustrează instanțierea componentelor și maparea semnalelor interne și externe.

- **Declararea Semnalelor Interne:**
  - `BtnUp_enable`, `BtnDown_enable`, `BtnCenter_enable`: Semnale de ieșire de la modulele `MPG`, folosite pentru debouncing-ul butoanelor de control.
  - `A`, `B`: Vectorii de 64 de biți proveniți din `MemoryUnit`, folosiți ca operanzi pentru `MMX_Unit`.
  - `index_out`: Indică poziția curentă din memorie, utilizată pentru afișarea pe LED-uri.
  - `Result`: Rezultatul operațiilor SIMD generate de `MMX_Unit`.
  - `SelectedBits`: Partea selectată a rezultatului (`lower` sau `upper`) pentru afișare.
  - `IndicatorLED`: LED care indică partea selectată a rezultatului.
  - `OpCode`, `Size`: Semnale de control extrase din comutatoarele (`sw`), utilizate pentru a determina operația SIMD și dimensiunea datelor.
- **Instanțierea Componentelor MPG:**
  - `MPG_Up`, `MPG_Down`, `MPG_Center`: Modulele `MPG` gestionează debouncing-ul butoanelor `btn(1)`, `btn(3)`, și `btn(0)` respectiv. Aceste module asigură că semnalele de control sunt stabile și precise înainte de a fi utilizate în sistem.
- **Instanțierea** `MemoryUnit`**:**
  - Modulul `MemoryUnit` gestionează stocarea și accesarea perechilor de vectori `A` și `B`. Semnalele `BtnUp_enable`, `BtnDown_enable`, și `BtnCenter_enable` controlează navigarea prin memorie.

- **Instanțierea `MMX_Unit`:**
  - `MMX`: Unitatea centrală care execută operațiile SIMD pe vectorii `A` și `B`, bazându-se pe semnalele de control `OpCode` și `Size`. Rezultatul operațiilor este trimis către semnalul `Result`.
- **Instanțierea `BitSelector`:**
  - `Selector`: Modulul `BitSelector` selectează fie partea inferioară (`lower`) fie partea superioară (`upper`) a rezultatului `Result` pe baza semnalului `SelectBit` (`sw(15)`). Rezultatul selectat este trimis către `SelectedBits` pentru afișare.
- **Instanțierea `SSD` (Afișaj 7-segmente):**
  - `SSD_Display`: Modulul `SSD` convertește valorile binare din `SelectedBits` într-un format compatibil cu afișajul pe 7 segmente, gestionând multiplexarea și decodificarea cifrelor.
- **Maparea LED-urilor:**
  - LED-urile de la `led(14 downto 5)` indică poziția curentă din memorie (`index_out`).
  - LED-urile `led(4 downto 3)` indică dimensiunea operației (`Size`).
  - LED-urile `led(2 downto 0)` indică operația selectată (`OpCode`).
  - LED-ul `led(15)` indică starea selectorului `SelectBit` prin semnalul `IndicatorLED`.

Fluxul de Date și Control 1. Intrările Utilizatorului: - Butoanele de control (`btn`) permit utilizatorului să navigheze prin seturile de date stocate în `MemoryUnit`. - Comutatoarele (`sw`) permit selectarea operației SIMD (`OpCode`) și dimensiunea datelor (`Size`), precum și alegerea părții de afișat a rezultatului (`SelectBit`).

2. Procesarea Datelor: - `MemoryUnit` furnizează vectorii `A` și `B` în funcție de indexul curent, controlat prin butoane. - `MMX_Unit` execută operațiile SIMD pe vectorii `A` și `B`, generând rezultatul `Result`.

3. Selectarea și Afișarea Rezultatului: - `BitSelector` selectează partea relevantă a rezultatului (`SelectedBits`) în funcție de `SelectBit`. - `SSD` convertește `SelectedBits` într-un format vizual și afișează cifrele pe afișajul pe 7 segmente. - LED-urile oferă feedback vizual asupra stării curente a sistemului, inclusiv poziția din memorie, dimensiunea operației, operația selectată și partea de afișat a rezultatului.

6 Testare și Validare

6.1 Introducere

Testarea și validarea reprezintă etape cruciale în dezvoltarea oricărui sistem hardware, asigurându-se că acesta funcționează conform specificațiilor și cerințelor stabilite. În contextul unității MMX, aceste procese sunt esențiale pentru a garanta corectitudinea operațiilor SIMD (Single Instruction, Multiple Data) implementate și pentru a asigura performanța și fiabilitatea sistemului final.

Scopul acestui capitol este de a descrie metodologia adoptată pentru testarea și validarea unității MMX, utilizând testbench-uri dedicate și simulări detaliate pentru fiecare componentă a sistemului. Vom prezenta modul în care fiecare modul a fost testat individual și integrat în sistemul complet, evidențiind rezultatele obținute și analiza acestora pentru a confirma funcționalitatea corectă și eficientă a designului hardware.

Importanța testării și validării nu poate fi subestimată, deoarece aceste etape asigură identificarea și corectarea erorilor înainte de implementarea finală, reducând astfel riscul de defecte în producție și îmbunătățind calitatea produsului final. Prin efectuarea unei testări riguroase, se poate asigura că unitatea MMX îndeplinește cerințele de performanță și fiabilitate necesare pentru aplicații reale.

Structura capitolului este organizată după cum urmează:

- **Metodologie de Testare**: Prezentarea strategiilor și instrumentelor utilizate în procesul de testare.
- **Testbench-uri pentru Module**: Detalierea testbench-urilor dezvoltate pentru fiecare componentă a unității MMX, inclusiv scenariile de testare și codurile VHDL.
- **Rezultate Simulare**: Prezentarea și analiza rezultatelor simulărilor pentru diferitele teste efectuate.
- **Validarea Funcționalității**: Compararea rezultatelor simulărilor cu specificațiile inițiale și identificarea eventualelor abateri sau erori.
- **Evaluarea Performanței**: Analiza performanței sistemului în termeni de timpi de execuție și utilizare a resurselor hardware.
- **Concluzii și Recomandări**: Sumarizarea principalelor descoperiri și propunerea de îmbunătățiri sau pași următori pentru dezvoltarea ulterioară.

Prin parcurgerea acestui capitol, vom demonstra că unitatea MMX a fost testată și validată eficient, asigurând o implementare solidă și fiabilă a operațiilor SIMD esențiale pentru performanța sistemului.

6.2 Metodologie de Testare

Testarea și validarea sunt etape esențiale în dezvoltarea unității MMX, asigurând că toate componentele funcționează corect atât individual, cât și în cadrul sistemului integrat. Metodologia adoptată pentru testarea unității MMX se bazează pe o abordare structurată, care include testarea unităților individuale, testarea integrată și validarea funcționalității generale. În continuare, vom detalia strategiile, instrumentele și criteriile utilizate în procesul de testare.

Strategia de Testare Abordarea adoptată pentru testarea unității MMX se poate împărți în următoarele etape principale:

- **Testarea Unităților Individuale:** Fiecare componentă a unității MMX, cum ar fi `PADD`, `PSUB`, `PADDS`, `PADDUS`, `PSUBS`, și `PMUL`, este testată independent pentru a verifica

funcționalitatea corectă conform specificațiilor. Acest lucru se realizează prin utilizarea testbench-urilor dedicate, care aplică diverse scenarii de testare și verifică rezultatele generate.

- **Testarea Integrată:** După validarea funcționalității unităților individuale, componentele sunt integrate în cadrul `MMX_Unit` și testate împreună pentru a asigura compatibilitatea și corectitudinea interacțiunilor între ele. Această etapă verifică dacă datele sunt procesate corect de către sistemul integrat și dacă fluxul de control funcționează conform așteptărilor.
- **Validarea Funcționalității Generale:** În această etapă, sistemul complet este testat pentru a confirma că unitatea MMX îndeplinește toate cerințele funcționale și de performanță. Testele includ scenarii complexe care combină mai multe operații SIMD și verifică rezultatele finale pentru a asigura corectitudinea și eficiența sistemului.

Instrumente Utilizate Pentru efectuarea testării și validării unității MMX, au fost utilizate următoarele instrumente și resurse:

- **Software de Simulare VHDL:** ModelSim și Vivado sunt utilizate pentru simularea și verificarea comportamentului codului VHDL. Aceste instrumente permit rularea testbench-urilor și observarea rezultatelor în timp real, facilitând identificarea și corectarea erorilor.
- **Testbench-uri VHDL:** Codurile testbench-urilor dezvoltate pentru fiecare componentă a unității MMX (de exemplu, `PADD_TB.vhd`, `PSUB_TB.vhd`, `PADDS_TB.vhd`, `PADDUS_TB.vhd`, `PSUBS_TB.vhd`, `PMUL_TB.vhd`) sunt utilizate pentru a aplica diverse scenarii de testare și pentru a verifica rezultatele generate de fiecare operație SIMD.
- **Imagini de Simulare:** Rezultatele simulărilor sunt documentate prin capturi de ecran și diagrame care ilustrează comportamentul sistemului în diverse scenarii de testare. Aceste imagini sunt utilizate pentru a susține analiza rezultatelor și pentru a demonstra conformitatea cu specificațiile.

Criterii de Acceptare Fiecare test efectuat este evaluat în funcție de următoarele criterii de acceptare:

- **Corectitudinea Rezultatelor:** Rezultatele generate de operațiile SIMD trebuie să corespundă așteptărilor stabilite în specificații pentru fiecare scenariu de testare.
- **Consistența Comportamentului:** Sistemul trebuie să funcționeze în mod consistent în toate scenariile de testare, fără a prezenta erori sau comportamente neașteptate.
- **Performanța Sistemului:** Timpii de execuție și utilizarea resurselor hardware trebuie să se încadreze în limitele specificate, asigurând o performanță optimă a unității MMX.
- **Robustețea:** Sistemul trebuie să fie robust în fața diferitelor combinații de input-uri și să gestioneze corect condițiile de overflow sau saturare, conform cerințelor.

Planul de Testare Planul de testare a unității MMX include următoarele etape:

1. **Dezvoltarea Testbench-urilor:** Crearea testbench-urilor VHDL pentru fiecare componentă a unității MMX, care să acopere toate scenariile de testare necesare.
2. **Execuția Testelor Unitare:** Rularea testelor individuale pentru fiecare componentă și verificarea rezultatelor generate față de așteptări.
3. **Analiza Rezultatelor:** Analizarea rezultatelor simulărilor pentru a identifica eventualele erori sau abateri de la specificații.
4. **Corectarea Erorilor:** Modificarea codului VHDL și ajustarea logicii de funcționare pentru a remedia erorile identificate în timpul testării.

5. **Testarea Integrată:** După validarea componentelor individuale, se efectuează testarea integrată a sistemului `MMX_Unit` pentru a asigura compatibilitatea și funcționarea corectă a întregului sistem.
6. **Validarea Funcționalității Generale:** Realizarea unor teste comprehensive care să valideze funcționalitatea completă a unității MMX în scenarii complexe de operare.

Documentarea Procesului de Testare Toate testele efectuate sunt documentate detaliat, incluzând descrierea fiecărui scenariu de testare, codurile testbench-urilor utilizate și rezultatele obținute. Această documentație este esențială pentru a demonstra conformitatea unității MMX cu specificațiile tehnice și pentru a facilita eventuale revizuiri sau extinderi viitoare ale sistemului.

6.2.1 Testbench pentru PADD

Descrierea Modulului Modulul `PADD` este responsabil pentru efectuarea operațiilor de adunare pe vectori de 64 de biți, suportând diferite dimensiuni ale datelor (`Size` = "00" pentru byte, "01" pentru word, și "10" pentru doubleword). Acesta gestionează wrap-around-ul pe diferite niveluri de granularitate, asigurând corectitudinea operațiilor de adunare în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
8	0x0101010101010101	0x0101010101010101	0x0202020202020202	0x0202020202020202
8	0xffffffffffffff	0x0101010101010101	0x0000000000000000	0x0000000000000000
8	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
8	0x0102030405060708	0x0101010101010101	0x0203040506070809	0x0203040506070809
8	0xf00faa55cc33ff00	0x0f0ff55a33cc00ff	0xff1e9fafffffffff	0xff1e9fafffffffff
16	0x0002000300040005	0x0006000700080009	0x0008000a000c000e	0x0008000a000c000e
16	0xffff0000ffff0000	0x0001000100010001	0x0000000100000001	0x0000000100000001
16	0x0002000300040005	0x0001000100010001	0x0003000400050006	0x0003000400050006
16	0x00f000aa00cc00ff	0x000f0001000f0000	0x00ff00ab00db00ff	0x00ff00ab00db00ff
32	0xffffffff00000000	0xffffffff00000000	0xfffffffe00000000	0xfffffffe00000000
32	0xffffffffffffff	0x0000000100000001	0x0000000000000000	0x0000000000000000
32	0x0000000100000002	0x0000000100000001	0x0000000200000003	0x0000000200000003
32	0x0000f0010000aa02	0x0000000100000f01	0x0000f0020000b903	0x0000f0020000b903
32	0x0101010101010101	0x0101010101010101	0x0202020202020202	0x0202020202020202
32	0xffffffffffffff	0xffffffffffffff	0xfffffffeffffffe	0xfffffffeffffffe

6.2.2 Testbench pentru PADDS

Descrierea Modulului Modulul `PADDS` este responsabil pentru efectuarea operațiilor de adunare saturată pe vectori de 64 de biți, suportând diferite dimensiuni ale datelor (`Size` = '0' pentru 8-bit și `Size` = '1' pentru 16-bit). Acesta gestionează saturarea în cazurile în care rezultatul adunării depășește capacitatea reprezentării pe dimensiunea specificată, asigurând astfel integritatea datelor în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
8	0x0102030405060708	0x0101010101010101	0x0203040506070809	0x0203040506070809
8	0x7f7f7f7f7f7f7f7f	0x0101010101010101	0x7f7f7f7f7f7f7f7f	0x7f7f7f7f7f7f7f7f
8	0x8080808080808080	0x0101010101010101	0x8181818181818181	0x8181818181818181
8	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
8	0xf0f0f0f0f0f0f0f0	0x0f0f0f0f0f0f0f0f	0xffffffffffffff	0xffffffffffffff
16	0x0002000300040005	0x0006000700080009	0x0008000a000c000e	0x0008000a000c000e
16	0x7fff7fff7fff7fff	0x0001000100010001	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff
16	0x8000800080008000	0x0001000100010001	0x8001800180018001	0x8001800180018001
16	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
16	0x00f000aa00cc00ff	0x000f0001000f0000	0x00ff00ab00db00ff	0x00ff00ab00db00ff

6.2.3 Testbench pentru PADDUS

Descrierea Modulului Modulul `PADDUS` este responsabil pentru efectuarea operațiilor de adunare saturată nesemnată pe vectori de 64 de biți, suportând diferite dimensiuni ale datelor (`Size` = '0' pentru 8-bit și `Size` = '1' pentru 16-bit). Acesta gestionează saturarea atunci când rezultatul adunării depășește capacitatea de reprezentare nesemnată a dimensiunii specificate, asigurând astfel integritatea datelor în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
8	0x0102030405060708	0x0101010101010101	0x0203040506070809	0x0203040506070809
8	0x7f7f7f7f7f7f7f7f	0x0101010101010101	0x8080808080808080	0x8080808080808080
8	0xff7f7f7f7f7f7f7f	0x0101010101010101	0xff80808080808080	0xff80808080808080
8	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
8	0xf0f0f0f0f0f0f0f0	0x0f0f0f0f0f0f0f0f	0xffffffffffffff	0xffffffffffffff
16	0x0002000300040005	0x0006000700080009	0x0008000a000c000e	0x0008000a000c000e
16	0x7fff7fff7fff7fff	0x0001000100010001	0x8000800080008000	0x8000800080008000
16	0xffff8000ffff8000	0x0001000100010001	0xffff8001ffff8001	0xffff8001ffff8001
16	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
16	0x00f000aa00cc00ff	0x000f0001000f0000	0x00ff00ab00db00ff	0x00ff00ab00db00ff

6.2.4 Testbench pentru PSUB

Descrierea Modulului Modulul `PSUB` este responsabil pentru efectuarea operațiilor de scădere pe vectori de 64 de biți, suportând diferite dimensiuni ale datelor (`Size` = "00" pentru 8-bit, "01" pentru 16-bit și "10" pentru 32-bit). Acesta gestionează wrap-around-ul pe diferite niveluri de granularitate, asigurând corectitudinea operațiilor de scădere în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
8	0x0102030405060708	0x0101010101010101	0x0001020304050607	0x0001020304050607
8	0x0000000000000000	0x0101010101010101	0xffffffffffffff	0xffffffffffffff
8	0xffffffffffffff	0xffffffffffffff	0x0000000000000000	0x0000000000000000
8	0xf0f1f2f3f4f5f6f7	0x0f0e0d0c0b0a0908	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
8	0x8080808080808080	0x7f7f7f7f7f7f7f7f	0x0101010101010101	0x0101010101010101
16	0x0005000400030002	0x0001000200010001	0x0004000200020001	0x0004000200020001
16	0x0000000000000000	0x0001000200030004	0xfffffffefffdfffc	0xfffffffefffdfffc

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
16	0xffffffffffffff	0xffffffffffffff	0x0000000000000000	0x0000000000000000
16	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff	0x0000000000000000	0x0000000000000000
16	0x8000000080000000	0x0001000100010001	0x7fffffff7fffffff	0x7fffffff7fffffff
16	0x00f000aa00cc00ff	0x000f0001000f0000	0x00e100a900bd00ff	0x00e100a900bd00ff
32	0x0000000200000000	0x0000000100000001	0x00000001ffffffff	0x00000001ffffffff
32	0x0000000000000000	0xffffffffffffffff	0x0000000100000001	0x0000000100000001
32	0xffffffffffffffff	0xffffffffffffffff	0x0000000000000000	0x0000000000000000
32	0x0000f0020000aa03	0x0000000100000f01	0x0000f00100009b02	0x0000f00100009b02
32	0xffffffff80000000	0xffffffff7fffffff	0x0000000000000001	0x0000000000000001
32	0xffffffff0000aa02	0x0000000100000f01	0xffffffe00009b01	0xffffffe00009b01
32	0xffffffff00000000	0x0000000100000001	0xffffffefffffffff	0xffffffefffffffff
32	0x0000000100000002	0x0000000100000001	0x0000000000000001	0x0000000000000001

6.2.5 Testbench pentru PSUBS

Descrierea Modulului Modulul `PSUBS` este responsabil pentru efectuarea operațiilor de scădere saturată semnată pe vectori de 64 de biți, suportând diferite dimensiuni ale datelor (`Size` = '0' pentru 8-bit și `Size` = '1' pentru 16-bit). Acesta gestionează saturarea atunci când rezultatul scăderii depășește capacitatea de reprezentare semnată a dimensiunii specificate, asigurând astfel integritatea datelor în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

Bit Width	A	B	Rezultatul Așteptat	Rezultatul Calculat
8	0x0102030405060708	0x0101010101010101	0x0001020304050607	0x0001020304050607
8	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
8	0xff7fff7fff7fff7f	0x0101010101010101	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
8	0x8000800080008000	0x0101010101010101	0x80ff80ff80ff80ff	0x80ff80ff80ff80ff
8	0xf0f0f0f0f0f0f0f0	0x0f0f0f0f0f0f0f0f	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
16	0xf00faa55cc33ff00	0x0f0ff55a33cc00ff	0xe100b4fb9867fe01	0xe100b4fb9867fe01
16	0xffffffffffffff	0xffffffffffffff	0x0000000000000000	0x0000000000000000
16	0xf0f1f2f3f4f5f6f7	0x0f0e0d0c0b0a0908	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
16	0x0001000100010001	0x0001000100010001	0x0000000000000000	0x0000000000000000
16	0x0005000400030002	0x0001000200010001	0x0004000200020001	0x0004000200020001
16	0x7fffffff7fffffff	0x0001000100010001	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
16	0x0000000000000000	0x0001000200030004	0xfffffffefffdfffc	0xfffffffefffdfffc
16	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff	0x0000000000000000	0x0000000000000000
16	0x8000000080000000	0x0001000100010001	0x8000ffff8000ffff	0x8000ffff8000ffff
16	0x0000f0020000aa03	0x0000000100000f01	0x0000f00100009b02	0x0000f00100009b02
16	0x0000000200000000	0x0000000100000001	0x000000010000ffff	0x000000010000ffff
16	0xffffffff80000000	0xffffffff7fffffff	0x0000000080000001	0x0000000080000001
16	0xffffffff0000aa02	0x0000000100000f01	0xffffffe00009b01	0xffffffe00009b01

6.2.6 Testbench pentru PMUL

Descrierea Modulului Modulul `PMUL` este responsabil pentru efectuarea operațiilor de multiplicare pe vectori de 64 de biți, suportând două moduri de operare: `PMULLW` (multiplicare pe partea inferioară a cuvintelor) și `PMULHW` (multiplicare pe partea superioară a cuvintelor). Acest modul gestionează corect multiplicarea în funcție de semn și dimensiunea datelor, asigurând integritatea și precizia rezultatelor în contextul SIMD (Single Instruction, Multiple Data).

Rezultate Simulare

A	B	Mode	Rezultatul Așteptat	Rezultatul Calculat
0x0001000100010001	0x0001000100010001	0	0x0001000100010001	0x0001000100010001
0x0001000100010001	0x0001000100010001	1	0x0000000000000000	0x0000000000000000
0x0000000000000000	0x0001000200030004	0	0x0000000000000000	0x0000000000000000
0x0000000000000000	0x0001000200030004	1	0x0000000000000000	0x0000000000000000
0x7fff7fff7fff7fff	0x0001000100010001	0	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff
0x7fff7fff7fff7fff	0x0001000100010001	1	0x0000000000000000	0x0000000000000000
0x7fff7fff7fff7fff	0x7fff7fff7fff7fff	0	0x0001000100010001	0x0001000100010001
0x7fff7fff7fff7fff	0x7fff7fff7fff7fff	1	0x3fff3fff3fff3fff	0x3fff3fff3fff3fff
0x8000800080008000	0x0001000100010001	0	0x8000800080008000	0x8000800080008000
0x8000800080008000	0x0001000100010001	1	0xffffffffffffffff	0xffffffffffffffff
0xffff0000ffff0000	0x0001000100010001	0	0xffff0000ffff0000	0xffff0000ffff0000
0xffff0000ffff0000	0x0001000100010001	1	0xffff0000ffff0000	0xffff0000ffff0000
0x7fff80007fff8000	0x7fff80007fff8000	0	0x0001000000010000	0x0001000000010000
0x7fff80007fff8000	0x7fff80007fff8000	1	0x3fff40003fff4000	0x3fff40003fff4000
0x0000f0020000aa03	0x0000000100000f01	0	0x0000f0020000d703	0x0000f0020000d703
0x0000f0020000aa03	0x0000000100000f01	1	0x0000ffff0000faf5	0x0000ffff0000faf5
0xffffffff80000000	0xffffffff7fffffff	0	0x0001000180000000	0x0001000180000000
0xffffffff80000000	0xffffffff7fffffff	1	0x00000000c0000000	0x00000000c0000000

6.3 Testbench pentru MMX Unit

6.3.1 Descrierea Modulului

Modulul `MMX_Unit` este proiectat pentru a executa operații aritmetice multiple pe vectori de 64 de biți, utilizând tehnologia SIMD (Single Instruction, Multiple Data). Acesta suportă diverse operații precum adunare, scădere, și multiplicare, pe diferite dimensiuni de date: 8 biți, 16 biți și 32 biți. Modulul gestionează corect operațiile pe cuvinte inferioare și superioare, asigurând precizia și integritatea rezultatelor, chiar și în cazuri de overflow sau subflux.

- **8 biți:**
  - Date semnate: $[-2^7, 2^7 - 1] = [-128, 127]$
  - Date nesemnate: $[0, 2^8 - 1] = [0, 255]$
- **16 biți:**
  - Date semnate: $[-2^{15}, 2^{15} - 1] = [-32,768, 32,767]$
  - Date nesemnate: $[0, 2^{16} - 1] = [0, 65,535]$
- **32 biți:**
  - Date semnate: $[-2^{31}, 2^{31} - 1] = [-2,147,483,648, 2,147,483,647]$
  - Date nesemnate: $[0, 2^{32} - 1] = [0, 4,294,967,295]$

6.3.2 Rezultate Simulare

Pentru a verifica funcționalitatea modulului `MMX_Unit`, a fost creat un testbench VHDL (`MMX_Unit_TB.vhd`) care efectuează o serie de teste pe diverse scenarii de operare. Rezultatele acestor teste sunt prezentate în tabelul de mai jos.

Nr.	A	B	Size	Operație	Rezultatul Așteptat	Computed
0	0x0102030405060708	0x0101010101010101	00 (8-bit)	PADD(0)	0x0203040506070809	0x0203040506070809
			00 (8-bit)	PADDS(1)	0x0203040506070809	0x0203040506070809
			00 (8-bit)	PADDUS(2)	0x0203040506070809	0x0203040506070809
			00 (8-bit)	PSUB(3)	0x0001020304050607	0x0001020304050607
			00 (8-bit)	PSUBS(4)	0x0001020304050607	0x0001020304050607
			01 (16-bit)	PADD(0)	0x0203040506070809	0x0203040506070809
			01 (16-bit)	PADDS(1)	0x0203040506070809	0x0203040506070809
			01 (16-bit)	PADDUS(2)	0x0203040506070809	0x0203040506070809
			01 (16-bit)	PSUB(3)	0x0001020304050607	0x0001020304050607

Nr.	A	B	Size	Operație	Rezultatul Așteptat	Computed
			01 (16-bit)	PSUBS(4)	0x0001020304050607	0x0001020304050607
			01 (16-bit)	PMULLW(5)	0x030207040b060f08	0x030207040b060f08
			01 (16-bit)	PMULHW(6)	0x0001000300050007	0x0001000300050007
			10 (32-bit)	PADD(0)	0x0203040506070809	0x0203040506070809
			10 (32-bit)	PSUB(3)	0x0001020304050607	0x0001020304050607
1	0x0000000000000000	0x0000000000000000	00 (8-bit)	PADD(0)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PADDS(1)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PADDUS(2)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PADD(0)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PADDS(1)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PADDUS(2)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULLW(5)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PADD(0)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
2	0xff7fff7fff7fff7f	0x0101010101010101	00 (8-bit)	**PADD(0)**	0x0080008000800080	0x0080008000800080
			00 (8-bit)	**PADDS(1)**	0x007f007f007f007f	0x007f007f007f007f
			00 (8-bit)	**PADDUS(2)**	0xff80ff80ff80ff80	0xff80ff80ff80ff80
			00 (8-bit)	PSUB(3)	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
			00 (8-bit)	PSUBS(4)	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
			01 (16-bit)	PADD(0)	0x0080008000800080	0x0080008000800080
			01 (16-bit)	PADDS(1)	0x0080008000800080	0x0080008000800080
			01 (16-bit)	**PADDUS(2)**	0xffffffffffffffff	0xffffffffffffffff
			01 (16-bit)	PSUB(3)	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
			01 (16-bit)	PSUBS(4)	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
			01 (16-bit)	PMULLW(5)	0x7e7f7e7f7e7f7e7f	0x7e7f7e7f7e7f7e7f
			01 (16-bit)	PMULHW(6)	0xffffffffffffffff	0xffffffffffffffff
			10 (32-bit)	PADD(0)	0x0081008000810080	0x0081008000810080
			10 (32-bit)	PSUB(3)	0xfe7efe7efe7efe7e	0xfe7efe7efe7efe7e
3	0x8000800080008000	0x0101010101010101	00 (8-bit)	PADD(0)	0x8101810181018101	0x8101810181018101
			00 (8-bit)	PADDS(1)	0x8101810181018101	0x8101810181018101
			00 (8-bit)	PADDUS(2)	0x8101810181018101	0x8101810181018101
			00 (8-bit)	**PSUB(3)**	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff
			00 (8-bit)	**PSUBS(4)**	0x80ff80ff80ff80ff	0x80ff80ff80ff80ff
			01 (16-bit)	PADD(0)	0x8101810181018101	0x8101810181018101
			01 (16-bit)	PADDS(1)	0x8101810181018101	0x8101810181018101
			01 (16-bit)	PADDUS(2)	0x8101810181018101	0x8101810181018101
			01 (16-bit)	PSUB(3)	0x7eff7eff7eff7eff	0x7eff7eff7eff7eff
			01 (16-bit)	**PSUBS(4)**	0x8000800080008000	0x8000800080008000
			01 (16-bit)	PMULLW(5)	0x8000800080008000	0x8000800080008000
			01 (16-bit)	PMULHW(6)	0xff7fff7fff7fff7f	0xff7fff7fff7fff7f
			10 (32-bit)	PADD(0)	0x8101810181018101	0x8101810181018101
			10 (32-bit)	PSUB(3)	0x7eff7eff7eff7eff	0x7eff7eff7eff7eff
4	0xf0f0f0f0f0f0f0f0	0x0f0f0f0f0f0f0f0f	00 (8-bit)	PADD(0)	0xffffffffffffffff	0xffffffffffffffff
			00 (8-bit)	PADDS(1)	0xffffffffffffffff	0xffffffffffffffff
			00 (8-bit)	PADDUS(2)	0xffffffffffffffff	0xffffffffffffffff
			00 (8-bit)	PSUB(3)	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
			00 (8-bit)	PSUBS(4)	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
			01 (16-bit)	PADD(0)	0xffffffffffffffff	0xffffffffffffffff
			01 (16-bit)	PADDS(1)	0xffffffffffffffff	0xffffffffffffffff
			01 (16-bit)	PADDUS(2)	0xffffffffffffffff	0xffffffffffffffff
			01 (16-bit)	PSUB(3)	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
			01 (16-bit)	PSUBS(4)	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
			01 (16-bit)	PMULLW(5)	0x2e102e102e102e10	0x2e102e102e102e10
			01 (16-bit)	PMULHW(6)	0xff1dff1dff1dff1d	0xff1dff1dff1dff1d
			10 (32-bit)	PADD(0)	0xffffffffffffffff	0xffffffffffffffff
			10 (32-bit)	PSUB(3)	0xe1e1e1e1e1e1e1e1	0xe1e1e1e1e1e1e1e1
5	0xf00faa55cc33ff00	0x0f0ff55a33cc00ff	00 (8-bit)	**PADD(0)**	0xff1e9faffffffff	0xff1e9faffffffff
			00 (8-bit)	PADDS(1)	0xff1e9f7fffffffff	0xff1e9f7fffffffff
			00 (8-bit)	PADDUS(2)	0xff1effaffffffff	0xff1effaffffffff
			00 (8-bit)	PSUB(3)	0xe100b5fb9967ff01	0xe100b5fb9967ff01
			00 (8-bit)	PSUBS(4)	0xe100b5fb9967ff01	0xe100b5fb9967ff01
			01 (16-bit)	PADD(0)	0xff1e9faffffffff	0xff1e9faffffffff
			01 (16-bit)	PADDS(1)	0xff1e9faffffffff	0xff1e9faffffffff

Nr.	A	B	Size	Operație	Rezultatul Așteptat	Computed
			01 (16-bit)	PADDUS(2)	0xff1efffffffff	0xff1efffffffff
			01 (16-bit)	PSUB(3)	0xe100b4fb9867fe01	0xe100b4fb9867fe01
			01 (16-bit)	PSUBS(4)	0xe100b4fb9867fe01	0xe100b4fb9867fe01
			01 (16-bit)	PMULLW(5)	0xf1e13ae2e1a40100	0xf1e13ae2e1a40100
			01 (16-bit)	PMULHW(6)	0xff0f0390f584ffff	0xff0f0390f584ffff
			10 (32-bit)	PADD(0)	0xff1f9faffffffff	0xff1f9faffffffff
			10 (32-bit)	PSUB(3)	0xe0ffb4fb9867fe01	0xe0ffb4fb9867fe01
6	0xffffffffffffff	0xffffffffffffff	00 (8-bit)	**PADD(0)**	0xfefefefefefefefe	0xfefefefefefefefe
			00 (8-bit)	PADDS(1)	0xfefefefefefefefe	0xfefefefefefefefe
			00 (8-bit)	**PADDUS(2)**	0xffffffffffffff	0xffffffffffffff
			00 (8-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	**PADD(0)**	0xfffefffefffefffe	0xfffefffefffefffe
			01 (16-bit)	PADDS(1)	0xfffefffefffefffe	0xfffefffefffefffe
			01 (16-bit)	**PADDUS(2)**	0xffffffffffffff	0xffffffffffffff
			01 (16-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULLW(5)	0x0001000100010001	0x0001000100010001
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	**PADD(0)**	0xfffffffefffffffe	0xfffffffefffffffe
			10 (32-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
7	0xf0f1f2f3f4f5f6f7	0x0f0e0d0c0b0a0908	00 (8-bit)	PADD(0)	0xffffffffffffff	0xffffffffffffff
			00 (8-bit)	PADDS(1)	0xffffffffffffff	0xffffffffffffff
			00 (8-bit)	PADDUS(2)	0xffffffffffffff	0xffffffffffffff
			00 (8-bit)	PSUB(3)	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
			00 (8-bit)	PSUBS(4)	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
			01 (16-bit)	PADD(0)	0xffffffffffffff	0xffffffffffffff
			01 (16-bit)	PADDS(1)	0xffffffffffffff	0xffffffffffffff
			01 (16-bit)	PADDUS(2)	0xffffffffffffff	0xffffffffffffff
			01 (16-bit)	PSUB(3)	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
			01 (16-bit)	PSUBS(4)	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
			01 (16-bit)	PMULLW(5)	0x4c2eba64189266b8	0x4c2eba64189266b8
			01 (16-bit)	PMULHW(6)	0xff1dff55ff86ffae	0xff1dff55ff86ffae
			10 (32-bit)	PADD(0)	0xffffffffffffff	0xffffffffffffff
			10 (32-bit)	PSUB(3)	0xe1e3e5e7e9ebedef	0xe1e3e5e7e9ebedef
8	0x0001000100010001	0x0001000100010001	00 (8-bit)	PADD(0)	0x0002000200020002	0x0002000200020002
			00 (8-bit)	PADDS(1)	0x0002000200020002	0x0002000200020002
			00 (8-bit)	PADDUS(2)	0x0002000200020002	0x0002000200020002
			00 (8-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PADD(0)	0x0002000200020002	0x0002000200020002
			01 (16-bit)	PADDS(1)	0x0002000200020002	0x0002000200020002
			01 (16-bit)	PADDUS(2)	0x0002000200020002	0x0002000200020002
			01 (16-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULLW(5)	0x0001000100010001	0x0001000100010001
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PADD(0)	0x0002000200020002	0x0002000200020002
			10 (32-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
9	0x0005000400030002	0x0001000200010001	00 (8-bit)	PADD(0)	0x0006000600040003	0x0006000600040003
			00 (8-bit)	PADDS(1)	0x0006000600040003	0x0006000600040003
			00 (8-bit)	PADDUS(2)	0x0006000600040003	0x0006000600040003
			00 (8-bit)	PSUB(3)	0x0004000200020001	0x0004000200020001
			00 (8-bit)	PSUBS(4)	0x0004000200020001	0x0004000200020001
			01 (16-bit)	PADD(0)	0x0006000600040003	0x0006000600040003
			01 (16-bit)	PADDS(1)	0x0006000600040003	0x0006000600040003
			01 (16-bit)	PADDUS(2)	0x0006000600040003	0x0006000600040003
			01 (16-bit)	PSUB(3)	0x0004000200020001	0x0004000200020001
			01 (16-bit)	PSUBS(4)	0x0004000200020001	0x0004000200020001
			01 (16-bit)	PMULLW(5)	0x0005000800030002	0x0005000800030002
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PADD(0)	0x0006000600040003	0x0006000600040003
			10 (32-bit)	PSUB(3)	0x0004000200020001	0x0004000200020001
10	0x7fffffff7fffffff	0x0001000100010001	00 (8-bit)	PADD(0)	0x7f00ff007f00ff00	0x7f00ff007f00ff00
			00 (8-bit)	PADDS(1)	0x7f00ff007f00ff00	0x7f00ff007f00ff00
			00 (8-bit)	**PADDUS(2)**	0x7fffffff7fffffff	0x7fffffff7fffffff
			00 (8-bit)	PSUB(3)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
			00 (8-bit)	PSUBS(4)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe

Nr.	A	B	Size	Operație	Rezultatul Așteptat	Computed
			01 (16-bit)	PADD(0)	0x8000000080000000	0x8000000080000000
			01 (16-bit)	**PADDS(1)**	0x7fff00007fff0000	0x7fff00007fff0000
			01 (16-bit)	PADDUS(2)	0x8000ffff8000ffff	0x8000ffff8000ffff
			01 (16-bit)	PSUB(3)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
			01 (16-bit)	PSUBS(4)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
			01 (16-bit)	PMULLW(5)	0x7ffffff7ffffff	0x7ffffff7ffffff
			01 (16-bit)	PMULHW(6)	0x0000ffff0000ffff	0x0000ffff0000ffff
			10 (32-bit)	PADD(0)	0x8001000080010000	0x8001000080010000
			10 (32-bit)	PSUB(3)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
			00 (8-bit)	PADD(0)	0x0001000200030004	0x0001000200030004
			00 (8-bit)	PADDS(1)	0x0001000200030004	0x0001000200030004
			00 (8-bit)	PADDUS(2)	0x0001000200030004	0x0001000200030004
			00 (8-bit)	**PSUB(3)**	0x00ff00fe00fd00fc	0x00ff00fe00fd00fc
			00 (8-bit)	PSUBS(4)	0x00ff00fe00fd00fc	0x00ff00fe00fd00fc
			01 (16-bit)	PADD(0)	0x0001000200030004	0x0001000200030004
11	0x0000000000000000	0x0001000200030004	01 (16-bit)	PADDS(1)	0x0001000200030004	0x0001000200030004
			01 (16-bit)	PADDUS(2)	0x0001000200030004	0x0001000200030004
			01 (16-bit)	**PSUB(3)**	0xffffffefffdfffc	0xffffffefffdfffc
			01 (16-bit)	PSUBS(4)	0xffffffefffdfffc	0xffffffefffdfffc
			01 (16-bit)	PMULLW(5)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PADD(0)	0x0001000200030004	0x0001000200030004
			10 (32-bit)	**PSUB(3)**	0xfffefffefffcfffc	0xfffefffefffcfffc
			00 (8-bit)	PADD(0)	0xfefefefefefefefe	0xfefefefefefefefe
			00 (8-bit)	**PADDS(1)**	0x7ffe7ffe7ffe7ffe	0x7ffe7ffe7ffe7ffe
			00 (8-bit)	PADDUS(2)	0xfefffefffefffeff	0xfefffefffefffeff
			00 (8-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PADD(0)	0xfffefffefffefffe	0xfffefffefffefffe
12	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff	01 (16-bit)	**PADDS(1)**	0x7fff7fff7fff7fff	0x7fff7fff7fff7fff
			01 (16-bit)	PADDUS(2)	0xfffefffefffefffe	0xfffefffefffefffe
			01 (16-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PSUBS(4)	0x0000000000000000	0x0000000000000000
			01 (16-bit)	PMULLW(5)	0x0001000100010001	0x0001000100010001
			01 (16-bit)	PMULHW(6)	0x3fff3fff3fff3fff	0x3fff3fff3fff3fff
			10 (32-bit)	PADD(0)	0xfffefffefffefffe	0xfffefffefffefffe
			10 (32-bit)	PSUB(3)	0x0000000000000000	0x0000000000000000
			00 (8-bit)	PADD(0)	0x8001000180010001	0x8001000180010001
			00 (8-bit)	PADDS(1)	0x8001000180010001	0x8001000180010001
			00 (8-bit)	PADDUS(2)	0x8001000180010001	0x8001000180010001
			00 (8-bit)	PSUB(3)	0x80ff00ff80ff00ff	0x80ff00ff80ff00ff
			00 (8-bit)	PSUBS(4)	0x80ff00ff80ff00ff	0x80ff00ff80ff00ff
			01 (16-bit)	PADD(0)	0x8001000180010001	0x8001000180010001
13	0x8000000080000000	0x0001000100010001	01 (16-bit)	PADDS(1)	0x8001000180010001	0x8001000180010001
			01 (16-bit)	PADDUS(2)	0x8001000180010001	0x8001000180010001
			01 (16-bit)	PSUB(3)	0x7ffffff7ffffff	0x7ffffff7ffffff
			01 (16-bit)	**PSUBS(4)**	0x8000ffff8000ffff	0x8000ffff8000ffff
			01 (16-bit)	PMULLW(5)	0x8000000080000000	0x8000000080000000
			01 (16-bit)	PMULHW(6)	0xffff0000ffff0000	0xffff0000ffff0000
			10 (32-bit)	PADD(0)	0x8001000180010001	0x8001000180010001
			10 (32-bit)	PSUB(3)	0x7ffefffe7ffefffe	0x7ffefffe7ffefffe
			00 (8-bit)	PADD(0)	0x0000f0030000b904	0x0000f0030000b904
			00 (8-bit)	PADDS(1)	0x0000f0030000b904	0x0000f0030000b904
			00 (8-bit)	PADDUS(2)	0x0000f0030000b904	0x0000f0030000b904
			00 (8-bit)	PSUB(3)	0x0000f00100009b02	0x0000f00100009b02
			00 (8-bit)	PSUBS(4)	0x0000f00100009b02	0x0000f00100009b02
			01 (16-bit)	PADD(0)	0x0000f0030000b904	0x0000f0030000b904
14	0x0000f0020000aa03	0x0000000100000f01	01 (16-bit)	PADDS(1)	0x0000f0030000b904	0x0000f0030000b904
			01 (16-bit)	PADDUS(2)	0x0000f0030000b904	0x0000f0030000b904
			01 (16-bit)	PSUB(3)	0x0000f00100009b02	0x0000f00100009b02
			01 (16-bit)	PSUBS(4)	0x0000f00100009b02	0x0000f00100009b02
			01 (16-bit)	PMULLW(5)	0x0000f0020000d703	0x0000f0020000d703
			01 (16-bit)	PMULHW(6)	0x0000ffff0000faf5	0x0000ffff0000faf5
			10 (32-bit)	PADD(0)	0x0000f0030000b904	0x0000f0030000b904
			10 (32-bit)	PSUB(3)	0x0000f00100009b02	0x0000f00100009b02
15	0x0000000200000000	0x0000000100000001	00 (8-bit)	PADD(0)	0x0000000300000001	0x0000000300000001
			00 (8-bit)	PADDS(1)	0x0000000300000001	0x0000000300000001
			00 (8-bit)	PADDUS(2)	0x0000000300000001	0x0000000300000001

Nr.	A	B	Size	Operație	Rezultatul Așteptat	Computed
			00 (8-bit)	**PSUB(3)**	0x00000001000000ff	0x00000001000000ff
			00 (8-bit)	PSUBS(4)	0x00000001000000ff	0x00000001000000ff
			01 (16-bit)	PADD(0)	0x0000000300000001	0x0000000300000001
			01 (16-bit)	PADDS(1)	0x0000000300000001	0x0000000300000001
			01 (16-bit)	PADDUS(2)	0x0000000300000001	0x0000000300000001
			01 (16-bit)	**PSUB(3)**	0x000000010000ffff	0x000000010000ffff
			01 (16-bit)	PSUBS(4)	0x000000010000ffff	0x000000010000ffff
			01 (16-bit)	PMULLW(5)	0x0000000200000000	0x0000000200000000
			01 (16-bit)	PMULHW(6)	0x0000000000000000	0x0000000000000000
			10 (32-bit)	PADD(0)	0x0000000300000001	0x0000000300000001
			10 (32-bit)	**PSUB(3)**	0x00000001ffffff	0x00000001ffffff
16	0xffffffff80000000	0xffffffff7fffffff	00 (8-bit)	PADD(0)	0xfefefefeffffffff	0xfefefefeffffffff
			00 (8-bit)	PADDS(1)	0xfefefefeffffffff	0xfefefefeffffffff
			00 (8-bit)	PADDUS(2)	0xffffffffffffff	0xffffffffffffff
			00 (8-bit)	PSUB(3)	0x0000000001010101	0x0000000001010101
			00 (8-bit)	PSUBS(4)	0x0000000080010101	0x0000000080010101
			01 (16-bit)	PADD(0)	0xfffeffffffffffff	0xfffeffffffffffff
			01 (16-bit)	PADDS(1)	0xfffeffffffffffff	0xfffeffffffffffff
			01 (16-bit)	PADDUS(2)	0xffffffffffffff	0xffffffffffffff
			01 (16-bit)	PSUB(3)	0x0000000000010001	0x0000000000010001
			01 (16-bit)	PSUBS(4)	0x0000000080000001	0x0000000080000001
			01 (16-bit)	PMULLW(5)	0x0001000180000000	0x0001000180000000
			01 (16-bit)	PMULHW(6)	0x00000000c0000000	0x00000000c0000000
			10 (32-bit)	PADD(0)	0xffffffefffffffff	0xffffffefffffffff
			10 (32-bit)	PSUB(3)	0x0000000000000001	0x0000000000000001
17	0xffffffff0000aa02	0x0000000100000f01	00 (8-bit)	**PADD(0)**	0xfffff000000b903	0xfffff000000b903
			00 (8-bit)	PADDS(1)	0xfffff000000b903	0xfffff000000b903
			00 (8-bit)	**PADDUS(2)**	0xffffffff0000b903	0xffffffff0000b903
			00 (8-bit)	PSUB(3)	0xffffffe00009b01	0xffffffe00009b01
			00 (8-bit)	PSUBS(4)	0xffffffe00009b01	0xffffffe00009b01
			01 (16-bit)	**PADD(0)**	0xffff00000000b903	0xffff00000000b903
			01 (16-bit)	PADDS(1)	0xffff00000000b903	0xffff00000000b903
			01 (16-bit)	**PADDUS(2)**	0xffffffff0000b903	0xffffffff0000b903
			01 (16-bit)	PSUB(3)	0xffffffe00009b01	0xffffffe00009b01
			01 (16-bit)	PSUBS(4)	0xffffffe00009b01	0xffffffe00009b01
			01 (16-bit)	PMULLW(5)	0x0000ffff0000c802	0x0000ffff0000c802
			01 (16-bit)	PMULHW(6)	0x0000ffff0000faf5	0x0000ffff0000faf5
			10 (32-bit)	**PADD(0)**	0x000000000000b903	0x000000000000b903
			10 (32-bit)	PSUB(3)	0xffffffe00009b01	0xffffffe00009b01

7 Concluzie

În cadrul acestui document, am prezentat dezvoltarea, implementarea, testarea și validarea unității MMX, o componentă aritmetică destinată efectuării operațiilor SIMD (Single Instruction, Multiple Data) pe vectori de 64 de biți. Scopul principal a fost de a crea o unitate eficientă și modulară, capabilă să gestioneze diverse operații aritmetice, inclusiv adunare, scădere și înmulțire, cu suport pentru mecanisme de saturare și wrap-around pe diferite dimensiuni de date (8-bit, 16-bit și 32-bit).

Dezvoltarea și Implementarea: Am structurat unitatea MMX în mai multe submodule, fiecare responsabil pentru o operație specifică, cum ar fi `PADD`, `PADDS`, `PADDUS`, `PSUB`, `PSUBS`, și `PMUL`. Această abordare modulară a permis testarea și optimizarea independentă a fiecărui submodul, facilitând astfel mentenanța și extinderea sistemului. Modulul central, `MMX_Unit`, a orchestrat fluxul de date și control, asigurând o integrare coerentă și eficientă a tuturor componentelor.

Testarea și Validarea: Am dezvoltat testbench-uri dedicate pentru fiecare submodul, care au fost utilizate pentru a verifica funcționalitatea corectă în diverse scenarii de testare. Simulările efectuate au demonstrat conformitatea rezultatelor cu așteptările specificate, evidențiind capacitatea unității MMX de a gestiona corect operațiile SIMD, inclusiv situațiile de saturare și wrap-around. Rezultatele simulărilor au confirmat robustetea și fiabilitatea implementării, asigurându-se că unitatea îndeplinește cerințele funcționale și de performanță stabilite.

Performanța și Resursele: Evaluarea performanței unității MMX a arătat că aceasta funcționează eficient în termeni de timpi de execuție și utilizare a resurselor hardware. Utilizarea unor circuite reutilizabile și a unei arhitecturi sincronizate prin semnalul de ceas (`clk`) a contribuit la optimizarea consumului de resurse și la reducerea complexității gestionării operațiilor multiple simultane. Scalabilitatea designului permite adăugarea de noi operații SIMD fără a necesita modificări semnificative ale structurii de bază.

Concluzii Finale: Unitatea MMX dezvoltată și prezentată în acest document reprezintă o soluție hardware robustă și eficientă pentru efectuarea operațiilor SIMD, cu suport pentru diferite dimensiuni și mecanisme de control al rezultatelor. Prin abordarea modulară și utilizarea unor metode de testare riguroase, am reușit să asigurăm corectitudinea și performanța sistemului, pregătindu-l pentru integrarea în aplicații complexe ce necesită procesare paralelă de date.

Recomandări pentru Viitor: Pentru îmbunătățirea ulterioară a unității MMX, se recomandă explorarea următoarelor direcții:

- **Extinderea Funcționalității**: Adăugarea de noi operații SIMD, cum ar fi operații logice și de manipulare a bitilor, pentru a crește versatilitatea unității.
- **Optimizarea Performanței**: Investigarea unor metode de optimizare suplimentară a timpurilor de execuție și a consumului de resurse, prin tehnici avansate de pipelining și paralele.
- **Validare pe Hardware Real**: Implementarea unității MMX pe o platformă hardware reală pentru a verifica comportamentul în condiții de operare reale și a identifica eventuale optimizări suplimentare.
- **Automatizarea Testării**: Dezvoltarea unor suite automate de testare pentru a facilita verificarea continuă a funcționalității unității în cazul modificărilor ulterioare ale designului.

În concluzie, unitatea MMX reprezintă un pas important în dezvoltarea unor soluții hardware eficiente pentru procesarea paralelă a datelor, oferind o bază solidă pentru viitoare extinderi și optimizări în domeniul procesării SIMD.

Bibliografie

1. Intel Corporation. "Intel Architecture MMX™ Technology." Documentație oficială, 1996. *Ardent Tool*.
2. Oracle Corporation. "MMX Instructions - x86 Assembly Language Reference Manual." Documentație oficială, 2014. *Oracle Docs*.
3. Wikibooks. "x86 Assembly/MMX." Ghid online, 2019. *Wikibooks*.
4. Freeman, C. "Intel MMX Pipelined Processor and Assembler." Proiect open-source pe GitHub, 2019. *GitHub*.
5. Wikipedia. "MMX (instruction set)." Articol enciclopedic, 2020. *Wikipedia*.
6. Plantation Productions. "The MMX Instruction Set." Art of Assembly Language, 1st Edition, 2000. *O'Reilly Media*.
7. Softpixel. "MMX Instruction Set." Ghid online, 2005. *Softpixel*.
8. University of Maryland, Baltimore County. "THE INTEL MICROPROCESSORS." Material didactic, 2010. *UMBC User Pages*.
9. Toronto Metropolitan University. "Intel MMX™ Technology Overview." Prezentare, 2000. *Toronto Metropolitan University*.
10. University of Tennessee, Knoxville. "MATRIX-VECTOR MULTIPLICATION USING RECONFIGURABLE SYSTEM." Proiect de cercetare, 2003. *University of Tennessee EECS*.