Universitatea
Transilvania
din Brașov
FACULTATEA DE INGINERIE ELECTRICĂ
ȘI ȘTIINȚA CALCULATOARELOR

Extensie a setului de instrucțiuni RISC-V

Conducător științific:
Conf. Ciobanu Cătălin Bogdan

Absolvent:
Pușcașu Alexandru

Brașov, 2025

Departamentul de Electronică și Calculatoare
Programul de studii: Sisteme electronice și de comunicații integrate

PUȘCAȘU Alexandru

Extensie a setului de instrucțiuni RISC-V

Conducător științific:
Conf. CIOBANU Cătălin Bogdan

Brașov, 2025

Cuprins

FIGURI

TABELE

CODURI SURSĂ

AI - Artificial Intelligence;
AMBA - Advanced Microcontroller Bus Architecture;
AMX - Advanced Matrix eXtension;
APB - Advanced Peripheral Bus;
ARM - Advanced RISC Machine;
ASCII - American Standard Code for Information Interchange;
AVX - Advaced Vector Extension;
AXI - Advanced eXtension Interface;
BRAM - Block Random Access Memory;
CV-X-IF - CoreV-eXtension-Interface;
DSP - Digital Signal Processor;
FF - Flip-Flop;
FPGA - Field Programmable Gate Array;
IoT - Internet of Things;
IP - Intellectual Property;
ISA - Instruction Set Architecture;
LUT - Look-Up Table;
PCIe - Peripheral Component Interconnect Express;
PLL - Phase Lock Loop;
RAM - Random Access Memory;
RISC - Reduced Instruction Set Computer;
RISC-V - Reduced Instruction Set Computer - Five;
RTL - Register Transfer Level;
SDP - Simple Dual Port;
SIMD - Single Instruction Multiple Data;
SOC - System On a Chip;
TDP - True Dual Port;
TPU - Tensor Processor Unit;
VTA - Versatile Tensor Accelerator;
SVE - Scalabele Vector Extension;
UART - Universal Asynchronous Receiver-Transmitter;
URAM - Ultra Random Access Memory;
USB - Universal Serial Bus;

1 INTRODUCERE

Această lucrare analiză extinderea setului de instrucțiuni RISC-V și implementarea unui accelerator care pentru instrucțiunile propuse. Acceleratorul este conectat direct la un procesor RISC-V printr-o interfață dedicată și definită pentru această arhitectură. Instrucțiunile propuse sunt pentru operații matriciale și implementarea hardware folosește memorii și topologii hardware dedicate.

Instrucțiunile definite sunt de tip SIMD, o instrucțiune operează cu mai multe date simultan, comparativ cu o instrucțiune normală. Prin această metodă o parte din operațiile de control sunt executate în hardware, ceea ce oferă o îmbunătățire a timpului de rulare. Acest proiect face parte din proiectul european de cercetare ISOLDE [1]–[3].

1.1 ACCELERATOARE PENTRU OPERAȚII MATRICIALE

Am identificat mai multe soluții de acceleratoare pentru operații matriciale. Soluțiile identificate sunt de la mai multe echipe. Una dintre soluții este de la Google, și este numită TPU [4], [5]. O altă soluție este proiectată de Apache și se numește VTA [6]. Altă soluție din surse publice, dezvoltat de PULP Project (ETH Zurich și Universitatea din Bologna), este acceleratorul Red mule [7].

Acceleratoarele identificate anterior nu sunt conectate direct cu procesorul. Acceleratorul VTA poate fi integrat într-un SOC [6]. Acceleratorul TPU comunică cu sistemul prin PCIe sau USB [4], [5]. Acceleratorul Red mule poate fi și el integrat în SOC prin interfața APB [7].

Legat de topologiile interne toate acceleratoarele folosesc matrice sistolice pentru înmulțirile de matrice și convoluție. Pe lângă logica aritmetică, există memorii tampon - pentru stocarea rezultatelor intermediare. Intern acceleratoarele au unități care se ocupă de operațiile cu memoria: generare de adrese, aducerea datelor, stocarea matricelor și stocarea rezultatelor intermediare. Suplimentar toate acceleratoarele au și regiștrii de configurație și control [4]–[7]. Din aceste acceleratoare, doar acceleratorul Red mule a fost proiectat pentru o singură operație (înmulțirea de matrice), restul de acceleratoare suportă și alte operații între matrice.

Legat de modul de programare, soluțiile de la Google și Apache au propriul lor limbaj de programare. Extern se compilează programul pentru ele și în timpul rulării se configurează adresa din memorie pentru acel program [4]–[6]. Datorită numărului redus de operații, Red mule, are nevoie doar de adresele datelor și dimensiunile matricelor.

Acceleratoare integrate în procesor avem de la companiile Intel și ARM. Intel oferă o extensie pentru operațiile matriciale numită AMX [8]. Această implementare are instrucțiuni dedicate în setul de instrucțiuni x86. Ea este compusă din 8 regiștrii matrice de $16linii \times 64bytes$. Abordarea lor este

de a aduce submatrici în acești regiștri și după să folosească circuite dedicate pentru procesarea lor. Fiind proiectat pentru Inteligența Artificială, operează cu date de întregi pe 8 biți (INT8) și numere reale pe 16 biți (BF16) [8].

Scalable Matrix Extension (SME) dezvoltat de ARM adaugă suport pentru operațiile matriciale bazându-se pe hardware existent în procesor. A fost introdus ca parte a arhitecturi Armv9-A și implică accelerarea de operații matriciale, pentru Inteligența Artificială. Operară cu date de tip număr real pe 16 biți (BF16 și FP16) și numere întregi pe 8 biți (INT8). Interpretează regiștrii vectoriali existenți ca și matrici, similar cu AMX. Optimizarea vine prin eliminarea instrucțiunilor de control în hardware, ceea ce îmbunătățește timpul de rulare [9].

Tipurile de date suportate de acceleratoare sunt diverse. Soluția de la Apache merge doar pe numere întregi pe 8 biți [6]. Acceleratorul TPU, în primele iterații a operat și el doar cu numere întregi pe 8 biți. Generațiile următoare au introdus și suportul pentru numerele cu virgulă mobilă pe 16 sau 32 de biți [4], [5]. Red mule are suport doar pentru numerele în virgulă mobilă pe 16 și 8 biți [7].

O altă soluție de accelerator pentru operațiile matriciale este Tensix de la Tenstorent [10]. Soluția lor este o rețea de elemente de procesoare interconectate. Un element de procesare este format din cinci procesoare RISC-V și acceleratoare pentru operații matriciale și vectoriale. Un element de procesare are două unități care se ocupă cu comunicarea între elemente, iar restul de trei procesoare pre/post-procesează datele și controlează acceleratoarele. Acceleratorul matricial este o matrice sistolică de 64×64 elemente iar unitatea vectorială procesează în paralel 64 de date. Tipurile de date suportate sunt numere reale între 2 și 32 de biți, și numere reale pe 8 biți. Modul de programare este bazat pe compilatorul de RISC-V și suplimentar este adăugată infrastructura Tenstorent [10].

1.2 Setul de instrucțiuni RISC-V

Setul de instrucțiuni RISC-V propus în 2010 de UC Berkeley [11] este un set de instrucțiuni (ISA) cu sursă liberă. El respectă principiile RISC. Este a cincea versiune de set de instrucțiuni propusă de UC Berkeley. Aceste seturi de instrucțiuni fiind gândite în principal pentru activități didactice. Fiind cu sursă liberă oricine poate să implementeze un procesor care implementează setul de instrucțiuni sau să modifice setul de instrucțiuni fără taxe sau alte implicații legale.

RISC-V a depășit nivelul educațional și a fost acceptat la nivel industrial. Din 2015 s-a fondat *RISC-V Foundation* care se ocupă cu menținerea setului de instrucțiuni și a extensiilor acestuia. Această organizație cuprinde 4500 de membrii din peste 70 de țări [12]. Standardele definite de RISC-V sun accesibile pentru toată lumea în mod egal, fără taxe sau costuri aferente și fără a ține partea unui partener.

Popularitatea RISC-V se datorează instrucțiunilor reduse, dar care pot rezolva numeroase probleme în multiple domenii. Datorită accesului liber la setul de instrucțiuni sunt posibile numeroase soluții, ceea ce accelerează timpul de lansarea al unui produs. Accesul liber permite o mai buna colaborare și inovație in industrie. Standardele oferite de *RISC-V Foundation* asigură o uniformitate software, cu toate că sunt posibile multiple implementări, standardul asigură că toate îndeplinesc aceleași funcționalități. Un alt avantaja îl reprezintă faptul că RISC-V a fost gândit pentru a fi extins pentru aplicații dedicate [13].

Prin domeniile de aplicație a le procesoarelor RISC-V se numără [13]:

- IoT: pot oferii consumul redus de energie necesar unor astfel de aplicații.
- Telefoane inteligente: prin modificări ale implementării se pot obține performanțele computaționale necesare sau se poate extinde setul de instrucțiuni pentru sarcinii dedicate.
- Industria auto, centre de date: procesoarele pot rezolva probleme computaționale complexe și prin implementarea unor extensii se permite dezvoltarea unor soluții simple, sigure și flexibile.

Pentru a extinde capabilitățile unui procesor există două opțiuni: realizarea unui accelerator dedicat sau definirea și implementarea de noi instrucțiuni în setul de instrucțiuni. Aceste extinderii ale capacității procesorului sunt proiectate pentru deservi unor domenii de aplicații specifice.

Acest concept de extensii există și la RISC-V. În cazul acestui set de instrucțiuni, există patru seturi de instrucțiuni de baza: un set pentru procesoare folosite în domeniul sistemelor încorporate și trei seturi generale care operară cu date pe 32, 64 sau 128 de biți. La aceste instrucțiuni de bază de adaugă 16 extensii deja definite și un interval de instrucțiuni pe care le poate definii utilizatorul pentru propriile lui aplicații.

Extensiile definite acoperă diverse domenii de aplicații. Dacă instrucțiunile RISC-V sunt pe 32 de biți, există o extensie, **C**, de la comprimate, care permite instrucțiuni pe 16 biți. Există extensii pentru numere în virgulă mobilă, trei posibilități, în funcție de lungimea numărului. Alte extensii simplifică și permit realizarea de mașini virtuale și translația de instrucțiuni. Cum era și normal există și instrucțiuni pentru procesoare vectoriale și procesoare de pachete.

Pentru a simplifica extensia setului de instrucțiuni încă de la început a fost definit un interval de valori pentru codul instrucțiuni, care să nu fie în conflict cu alte extensii standard. Intervalul este $0b0001011 - 0b0101011$, cea ce se traduce prin 32 de valori. Dar cum mai există și câmpul *funct3*, care mai oferă 3 biți pentru specifica subtipul instrucțiuni, rezultă că RISC-V permite definirea a minim 256 de instrucțiuni. În funcție de codarea folosită se pot utiliza și suplimentar alți 7 biți pentru instrucțiune.

2 CUNOȘTINȚE SPECIFICE

În acest capitol urmărim să prezentăm cunoștințe folosite în acest proiect. Aspectele acoperite în cele ce urmează nu le considerăm comune, iar pentru o mai bună înțelegere dorim să le clarificăm.

2.1 CV-X-IF

Extinderea setului de instrucțiuni de cele mai multe ori implică modificarea procesorului, deoarece nu există un mod dinamic de descărcare a instrucțiunilor invalide, neimplementate în acel procesor. O soluție de extindere a setului de instrucțiuni a fost propusă de Altera/Intel la procesorul Nios-II.

Soluția propusă de Nios-II era reprezentată de o interfață de extensie, care permitea utilizatorului să implementeze propriile instrucțiuni într-un circuit dedicat. Acea interfață permitea acces la instrucțiune, operanzi și regiștrii (scrierea și citirea lor).

Asemănător cu soluția de la Nios-II, pentru procesoarele RISC-V există o interfață de extensie pentru instrucțiunile invalide. Această interfață se numește CoreV-eXtension-Interface (CV-X-IF). Interfața este formată din 5 canale:

1. **Compressed** - pentru decodificarea instrucțiunilor comprimate
2. **Issue** - pentru identificarea unei componente care să execute instrucțiunea invalidă
3. **Commit** - pentru validarea execuției, interfața permite și execuția speculativă a instrucțiunilor, prin acest canal se validează instrucțiunea respectivă
4. **Register** - pentru a citii date de la regiștrii
5. **Result** - pentru transmiterea rezultatelor în registrul destinație, dacă este cazul, și confirmarea finalizării instrucțiuni

Când procesorul identifică o instrucțiune invalidă o va trimite pe această interfață. Mai întâi o trimite pe canalul Comprossed și primește ca răspuns instrucțiunea expandată, daca este cazul. Ulterior o transmite pe interfața de Issue si așteaptă să afle dacă este acceptată de o altă componentă. Cum instrucțiunile pot fi speculative prin canalul Commit se anunță dacă instrucțiunea trebuie executată sau nu. Pe canalul Register un accelerator are acces la regiștrii procesorului, nu are acces la orice registru și doar la aceia care sunt specificații în instrucțiune. Pe canalul Result se răspunde procesorului că instrucțiunea si-a terminat execuția, daca au apărut erori sau excepții și se pot scrie rezultatul în registrul rezultat din instrucțiune.

Pentru a avea acces la regiștrii prin această interfață noile instrucțiuni trebuie să respecte codarea instrucțiunilor RISC-V. Deși pare restrictivă această soluție oferă protecție registrelor, dar și o viteză crescută pentru schimbul de date.

2.2 Memorie Polimorfică

O memorie clasică ar fi fost restrictivă pentru un accelerator pentru operații matriciale. O memorie clasică permite accesul mai multor date de pe aceeași linie, prin creșterea lățimii datelor. Dar această soluție nu este preferată pentru accesul pe coloană din cauza că datele nu sunt continue și vor fi necesare mai multe accese la memorie și module care să calculeze adresele de salt.

Pentru această problemă am identificat un tip de memorie numit memorie polimorfică, *PolyMem* [14] și folosim schemele de memorie definite original în *Polymorphic Register File* [15], [16].

Acest tip de memorie este compus din mai multe module de memorie clasică, dar suplimentar, pe baza tipului de acces la memoria polimorfică, sunt calculate adresele de acces pentru fiecare modul de memorie independent. Metodele de acces sunt: Dreptunghi (ReO), Linie (ReRo), Coloană (RoCo), Linie-Coloană (RoCo) și Transpusa (ReTr). Primele trei scheme permit accesul doar modului de acces al cărui nume îl poartă. Tipul RoCo permite accesul pe linie, pe coloană și o sub matrice. Tipul ReTr permite accesul matricei în mod transpus.

În Figura 1 este prezentat un exemplu de arhitectură internă a memorie polimorfice. Figura respectivă prezintă opt module de memorie. Pe baza adresei bidimensionale și modului de acces sunt calculate adresele fiecărei memorii și datele sunt rearanjate. La ieșire datele sunt puse în ordinea corectă și pot fi accesate de utilizator. Latența minimă este de un tact de ceas. Datorită nivelelor de rearanjare și calculare a adreselor calea critică va crește și pentru aceasta se preferă folosirea unor bistabili la ieșire să nu afecteze designul proiectantului.

Acest tip de memorie permite accesul la mai multe date în paralel, direct proporțional cu numărul de module de memorie folosite. Capacitatea memoriei este de tip bidimensional și permite stocarea a $N \times M$ cuvinte, lățimea unui cuvânt este egală cu lățimea datelor din modulele de memorie.

Un avantaj al memoriei polimorfice folosite este posibilitatea de a avea mai multe interfețe de citire și scriere independente. Acest lucru ne permite să scăpăm de hazardul structural și să putem să citim ambii operanzi în același timp și să scriem și rezultatul simultan.

O interfața pentru memorie polimorfică cuprinde adresa 2D de la care începe să se acceseze datele și modul de acces. Numărul de date este definit la momentul implementării.

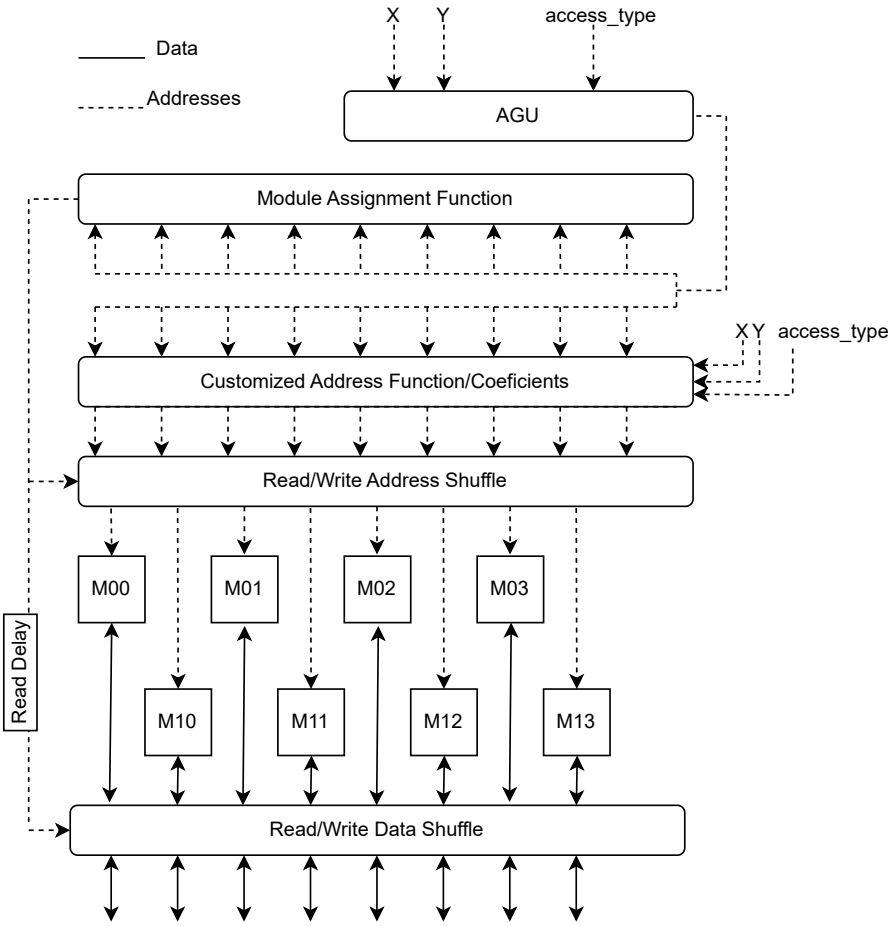

Figura 1: Arhitectura internă a memorie polimorfice, prelucrare după [15]

2.3 MATRICE SISTOLICĂ

O matrice sistolică este compusă din elemente de procesare (EP) conectate între ele pe două dimensiuni. Pentru înmulțirea de matrice sunt necesare două matrice la intrare și rezultatul va fi o matrice. Pentru a calcula elementul de la poziția *i, j* trebuie calculat produsul scalar dintre linia *i* de la prima matrice și coloana *j* de la a doua matrice. Bazat pe arhitectura ei, matricea sistolica are două fluxuri de procesare: pe linii și pe coloane. Datele primei matrice sunt procesate în ordinea linie-cu-line, la a doua matrice sunt procesate în ordinea coloana-cu-coloana [11].

Figura 2 prezintă fluxul de date în structura matricei sistolice. La intrare sunt două matrice de 2×2 și matricea sistolică este de 2×2. Matricea *A* este procesată linie-cu-linie pe liniile matricei sistolice. Matricea *B* este procesată coloană-cu-coloană și parcurge coloanele matricei sistolice. Matricea sistolică necesită o sincronizare strictă, pentru a avea date valide în toate PE-urile. Pentru a ajuta sincronizarea, anumite elemente cu valoare nulă sunt inserate în fluxul de procesare.

Un EP are două funcționalități: i) realizarea de operații de înmulțire și adunare și ii) ca element de

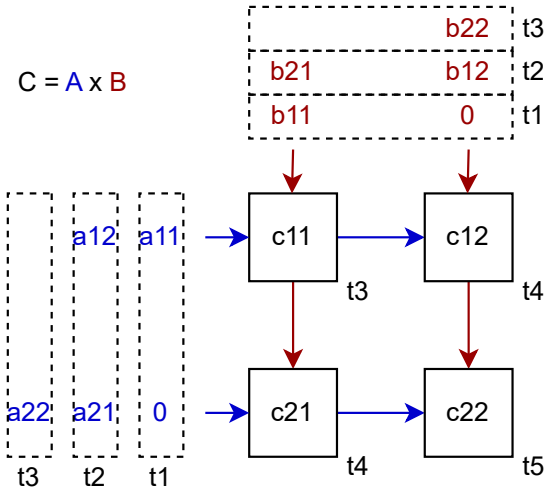

Figura 2: Exemplu de matrice sistolică, exemplificarea sincronizării pentru înmulțirea de matrice

procesare, stochează datele și le trimite la următorul element de procesare [17]. Acest element are un rol pasiv, ca unitate aritmetică, și un rol activ, ca element din linia de procesare. Un exemplu de element de procesare este în Figura 3.

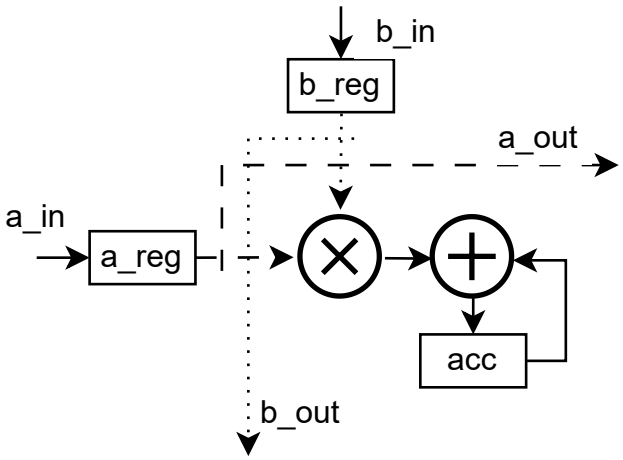

Figura 3: Exemplu de element de procesare

2.4 Extensii SIMD/Vectoriale

Extensiile SIMD sunt o parte importanta a instrucțiunilor unui procesor deoarece permit procesarea mai multor date simultan folosind o singură instrucțiune [11]. O denumire alternativă pentru astfel de procesoare este denumirea de procesor vectorial. Aceste extensii îmbunătățesc timpul de rulare și reduc consumul de energie [17], [18]. Acest paralelism la nivel de date ajută la reducerea hazardurilor de control, acest tip de hazard apare când procesorul are de verificat o instrucțiunile cu salt condiționat [11].

În procesoarele actuale aceste date sunt procesate în paralel prin multiplicarea unităților aritmetice. Un procesor clasice poate procesa doar o singură dată pe instrucțiune [11]. Acceleratoarele vectoriale se bazează pe paralelismul la nivel de date și astfel îmbunătățesc timpul de rulare. Prin procesarea mai multor date în paralel se reduce și numărul total de instrucțiuni executate de un program. O comparație între procesoarele vectoriale și cele scalare ne arată că un procesor scalar are de executat multiple instrucțiuni de calculare a adreselor și de verificarea a condițiilor, pentru iterarea datelor [17].

Procesoarele vectoriale, precum cele scalare au și ele regiștrii pentru stocarea datelor de lucru. Un astfel de registru vectorial stochează mai multe elemente. Unele valori comune sunt de 128, 256 sau 512 biți lățime. În funcție de tipul de dată se poate stoca un număr variabil de date, de exemplu pentru un registru de 128 de biți se pot stoca 4 numere de 32 de biți sau 8 numere de 16 biți [11].

Operații comune pentru procesoarele vectoriale sunt operații precum adunare, scădere, înmulțire și împărțire element cu element. Prin combinarea acestor operațiilor și cu algoritmi speciali se pot realiza și operații mai complexe pe matrice, precum înmulțirea de matrice și convoluția.

Procesoarele vectoriale necesită înțelegerea platformei hardware, înțelegerea algoritmului și a metodelor de paralelizare de date. La nivel de programare, utilizatorul are nevoie să cunoscută lățimea registrelor vectoriale și numărul lor, acest lucru are un impact asupra volumului de date ce pot fi încărcate în regiștrii și procesate în paralel. Utilizarea procesoarelor vectoriale poate necesita un număr ridicat de schimbări în codul sursă. Acest lucru este necesar datorită volumului de date procesate în paralele, dar și reorganizarea acceselor la memorie pentru a îmbunătății timpii cu operațiile la memorie.

Printre soluțiile comerciale de procesoare vectoriale se numără NEON de la ARM, RVV pentru procesoarele RISC-V sau AVX de la Intel. Extensia Neon dispune de regiștrii vectoriali de 128 de biți și suportă numere întregi și numere reale. O nouă extensie vectorială propusă de ARM, SVE (Scalabele Vector Extension) permite regiștrii de până la 2048 [19], [20].

Extensia vectorială pentru RISC-V (RVV) definește suplimentar 32 de regiștrii vectoriali. Lungimea lor este între 128 și 512 biți. Standardul actual este versiunea 1.0, lansată în 2021. Suportă date de tip număr întreg și numere reale, cu precizie simplă sau dublă [19], [21].

Intel a introdus în 2011 extensia vectorială AVX (Advaced Vector Extension). Această extensie a introdus regiștrii de 256 de biți. Această extensie a primit o actualizare în 2016 și a modificat regiștrii vectoriali la 512 biți [19], [22].

3 SPECIFICAȚII

În acest capitol o să descriem specificațiile definite la începutul proiectului. Aceste specificații au avut un impact asupra deciziilor de design pe care le-am luat pe parcurs.

Specificațiile funcționale pentru accelerator sunt:

1. Acceleratorul va opera cu matrice și va oferii suport pentru operații matriciale.
2. Acceleratorul să suporte tipuri de date de tip întreg, cu și fără semn, pe 8, 16 sau 32 de biți.
3. Acceleratorul să suporte următoarele operații matriciale: adunare, scădere, înmulțire de matrice, înmulțire și împărțire element cu element.
4. Acceleratorul să suporte operații care să modifice toate elementele din matrice în același fel (scalarea matricei).
5. Acceleratorul să suporte regiștrii matriciali definiți software de utilizator în timpul rulării.
6. Acceleratorul să ofere utilizatorului o abstractizare a matricei și operațiilor. Utilizatorul va definii matricele (lungime, lățime și tipul de dată) iar printr-o singură instrucțiune va indica acceleratorului operația dorită.
7. Acceleratorul să dispună de suport pentru compilator, care să simplifice dezvoltarea de programe.
8. Acceleratorul să dispună de o memorie internă.
9. Acceleratorul să se conecteze prin interfețe standard la procesorul RISC-V și restul sistemului.
10. Acceleratorul trebuie să realizez independent operații la memorie.
11. Acceleratorul trebuie să aibă acces la același spațiu de adrese ca procesorul.

Aceste specificații funcționale au fost verificate în primă faza prin simulare a designului. Ulterior am verifica prin rularea de teste pe un design real; procesorul împreună cu acceleratorul, conectate la memorie și un teste scris în C pentru testarea corectitudinii acestor operații.

Acceleratul nostru trebuie să fie optimizat pentru operațiile cu matrice și trebuie să fie conectat la un procesor RISC-V. Interfața de conectare la procesor va fi interfața standard CV-X-IF. Interfața la memorie va fi ARM AMBA AXI. Împreună cu procesorul, acceleratorul va fi conectat la aceeași magistrală cu procesorul pentru a permite spațiul comun de adrese.

Aceste specificații vor avea un impact și la instrucțiunile definite. Trebuie să oferim instrucțiunile necesare pentru toate operațiilor definite pe toate tipurile de date pe care urmărim să le suportăm.

Tabelul 1 prezintă specificațiile non funcționale pe care le urmărim. Printre optimizările urmărite de noi se numără realizarea unui design sintetizabil pentru FPGA, și validarea se va realiza print-un proces în Vivado. Pentru asigurarea calității designului urmărim să atingem cel puțin 100MHz pe un FPGA. O metrică foarte importantă pentru un accelerator este reducerea timpului de rulare și reducerea numărului de instrucțiuni executate de program, aceste proprietăți vor fi măsurate pe sistemul final. Acceleratorul nostru va avea mai mulți parametrii, prin urmare dorim ca acești parametrii să ajute la îmbunătățirea performantelor.

Nr. crt.	Optimizare	Metrică
1	Acceleratorul să fie sintetizabil	Sintetizarea designului pe un FPGA Xilinx VCU 128; trebuie să trecem cu succes etapele din Vivado.
2	Îmbunătățirea timpului de rulare	Îmbunătățire de două ori a timpului de rulare în comparație cu un procesor RISC-V simplu
3	Reducerea numărului de instrucțiuni	Reducerea cu cel puțin 30% a instrucțiuni executate, în comparație cu un procesor RISC-V simplu
4	Frecvență	Minim 100MHz frecvență acceleratorului pe un FPGA
5	Scalabilitate	Îmbunătățirea timpului de rulare proporțional cu numărul de unități aritmetice interne

Tabel 1: Specificații nefuncționale

Acceleratorul va fi realizat în SystemVerilog, printr-o decizie personală. Pentru monitorizarea progresului și publicare sub licență cu surse libere se va folosii platforma GitHub.

4 SET DE INSTRUCȚIUNI

Cum acceleratorul dezvoltat este unul cuplat la procesor, pentru al controla avem nevoie de instrucțiuni dedicate. În acest capitol prezentăm instrucțiunile noi definite. Pentru îmbunătății experiență de programare am adăugat noile instrucțiuni și în compilatorul pentru RISC-V.

4.1 INSTRUCȚIUNILE

Datorită interfeței de extensie instrucțiunile noi introduse trebuie să respecte tipurile de codare definite în RISC-V. Acceptăm această soluție deoarece urmărim să schimbăm date între accelerator și procesor prin intermediul registrelor.

RISC-V fiind gândit pentru extinderea setului instrucțiuni, a prevăzut un interval de valori pentru codul instrucțiuni. Aceste valori pot fi folosite fără a exista riscul să între în conflict cu alte instrucțiuni standardizate. Am reușit să codificăm instrucțiunile folosind un singur cod de instrucție. Scopul instrucțiuni fiind identificat prin alte câmpuri.

Instrucțiunile definite, până la momentul actual, se împart în trei categorii:

1. pentru definirea registrelor software
2. pentru operații între matriciale
3. pentru operații la memoria principală

Pentru codarea instrucțiunilor am folosit două tipuri de codări. Am folosit formatul *R*, care este folosit pentru operații aritmetice. Acest format definește doi regiștrii sursă, operanzii, un registru destinație și 10 biți pentru scopul instrucțiuni. Al doilea format folosit este cel de tip *I*, folosit pentru instrucțiunile pentru operații la memorie. Cu acest format avem un registru sursă, adresa de baza, un registru destinație, un număr întreg cu semn pe 12 biți și 3 biți pentru scopul instrucțiuni.

Cum procesoarele RISC-V au maxim 32 de regiștrii, acesta este și numărul de regiștrii software pe care îi avem. Tipurile de date pe care le suportăm sunt numere întregi, cu și fără semn pe 8, 16 sau 32 de biți. Instrucțiunile RISC-V au două elemente comune, câmpul cu valoare de instrucțiune și un câmp de 3 biți pentru subtipul instrucțiuni.

Pentru a definii un registru software avem nevoie să știm tipul de dată, lungime, lățimea, coordonatele în memoria bidimensională și numărul registrului. Prin urmare reiese nevoie de divizare în mai multe instrucțiuni. Numărul maxim de regiștrii pe care îi putem accesa într-o tranzacție este de maxim doi. Soluția aleasă este să folosim două instrucțiuni codate tip *R*. Prima instrucțiune, denumită *v2ddef<tip>*, definește lungimea și lățimea, și tipul de date (această informați este codată pe cei 7

biți rămași liberi). A doua instrucțiune, denumită *v2dloc* definește locația în memoria bidimensionala. În aceste cazuri regiștrii sursă vin de la procesor, iar registrul destinație este în accelerator.

Instrucțiunile pentru operațiile matriciale sunt codificate folosind tot tipul *R*. Acestea sunt de două tipuri: operații între matrice, cu sufixul *.vv*, sau operații între o matrice și un scalar, cu sufixul *.vs*. În cazul *.vv* ambii operanzi reprezintă regiștrii din accelerator. În cazul *.vs* primul operand este o matrice din accelerator, iar al doilea operand este un scalar, care vine de la procesor. Rezultatul este un registru matricial definit software.

Pentru instrucțiunile de acces la memoria principală, am folosit formatul *I*. Operandul care vine de la procesor este adunat cu incrementul oferit prin cei 12 biți, astfel obținându-se adresa de la care trebuie să se realizeze citirea sau scrierea. Cum acceleratorul știe tipul de dată și dimensiunile, calculează în mod automat numărul de byți necesari. Acceleratorul presupune că datele sunt stocate în memorie în modul limbajului de programare C, linie după line. Și mai presupune că vectorul a fost liniarizat, liniile sunt una după alta, nu s-au folosit matrice de referințe.

În Tabelul 2 sunt prezentate toate instrucțiunile definite. Tot în acel tabel am prezentat sursa operanzilor. Pe lângă sursa operanzilor am prezentat și valorile care ne permit decodificarea instrucțiunilor.

Așa cum am prezentat anterior câmpul *funct3* este comun tuturor formatelor de instrucțiune RISC-V, acest câmp ne permite să identificăm tipul instrucțiunii. Pentru fiecare tip de instrucțiune am asociat un număr unic. Singura excepție fiind instrucțiunile pentru operațiile la memorie, deoarece aceste instrucțiuni, de tip *I*, nu au alt câmp pentru decodificare, ce 12 biți rămași sunt pentru un număr întreg.

Instrucțiunile de tip *R* mai ai un câmp *funct7*, 7 biți, care permit identificarea subcategoriei. Acest câmp ne permite să identificăm tipul de dată al registrului, dar și modul de plasare în memoria bidimensională. Tot acest câmp ajută se identificăm operația aritmetică pe care o avem de realizat.

Operațiile matriciale suportate sunt adunarea, scăderea, împărțirea și înmulțirea. Ca și extensie a acestor operații există împărțirea și înmulțirea element-cu-element. Alte operații suportate sunt cele între o matrice și un scalar. Aceste operații permit modificarea fiecărui element din matrice cu un scalar, un singur număr.

Nr. crt.	Mnemonică	Sursă operator 1	Sursă operator 2	funct3	funct7	Descriere
1	v2ddef8	Procesor	Procesor	0	0	Instrucțiunile pentru definerea tipurlui de date, a lungimii și lățimii
2	v2ddefu8	Procesor	Procesor	0	1	
3	v2ddef16	Procesor	Procesor	0	2	
4	v2ddefu16	Procesor	Procesor	0	3	
5	v2ddef32	Procesor	Procesor	0	4	
6	v2ddefu32	Procesor	Procesor	0	5	
7	v2dloc.rect	Procesor	Procesor	1	0	Instrucțiunile pentru plasarea registrului în memoria bidemensională apecificarea tipului de matrice
8	v2dloc.row	Procesor	Procesor	1	1	
9	v2dloc.col	Procesor	Procesor	1	2	
10	v2dloc.trect	Procesor	Procesor	1	5	
11	v2dld	Procesor	-	2	-	Instrucțiunile pentru operațiile cu memoria principală
12	v2dst	Procesor	-	3	-	
13	v2dadd.vv	Accelerator	Accelerator	4	1	Operație de adunarea între două matrici
14	v2dsub.vv	Accelerator	Accelerator	4	2	Operația de diferență între între două matrici
16	v2ddiv.vv	Accelerator	Accelerator	4	8	Împărțire element cu element a datelor din două matrici
17	v2dmul.vv	Accelerator	Accelerator	4	16	Operația de înmulțire a două matrici
18	v2dsmul.vv	Accelerator	Accelerator	4	48	Înmulțirea element cu element a datelor din două matrici
19	v2dadd.vs	Accelerator	Procesor	5	1	Adunarea tuturor elementelor din matrice cu un scalar
20	v2dsub.vs	Accelerator	Procesor	5	2	Scăderea tuturor elementelor din matrice cu un scalar
21	v2ddiv.vs	Accelerator	Procesor	5	8	Împărțirea tuturor elementelor din matrice cu un scalar
22	v2dmul.vs	Accelerator	Procesor	5	16	Înmulțirea tuturor elementelor din matrice cu un scalar

Tabel 2: Lista completă a instrucțiunilor definite

4.2 SUPORT DE COMPILATOR

Acceleratorul are și suport de compilator. Am modificat *riscv-gcc-toolchain* [23] și am adăugat instrucțiunile definite de noi. Mai concret suportul este doar la nivel de asamblor, nu putem să optimizăm direct fragmente de cod. Utilizatorul poate să folosească *inline assembly* și să apeleze noile instrucțiuni.

Pentru început am modifica proiectul *riscv-opcodes* [24]. Am realizat un *fork* de la acest proiect și am adăugat noile instrucțiuni, modificările sunt disponibile pe GitHub-ul personal [25]. Am ales acest proiect deoarece verifică instrucțiunile să nu fie în conflict între ele. Suplimentar mai generează și fragmentele de cod ce pot fi folosite pentru a modifica elemente din mediul de dezvoltare RISC-

V. Pentru cazul nostru, cel mai important rămân condițiile de potrivire a instrucțiunilor noi definite. Aceste condiții le-am folosit în compilator.

```
# define 2d register
v2ddef8   md rs1 rs2 31..25=0 14..12=0 6..0=0x2B
v2ddefu8  md rs1 rs2 31..25=1 14..12=0 6..0=0x2B
v2ddef16  md rs1 rs2 31..25=2 14..12=0 6..0=0x2B
v2ddefu16 md rs1 rs2 31..25=3 14..12=0 6..0=0x2B
v2ddef32  md rs1 rs2 31..25=4 14..12=0 6..0=0x2B
v2ddefu32 md rs1 rs2 31..25=5 14..12=0 6..0=0x2B

# place register in poly mem, also define shape
v2dloc.rect  md rs1 rs2 31..25=0 14..12=1 6..0=0x2B
v2dloc.row   md rs1 rs2 31..25=1 14..12=1 6..0=0x2B
v2dloc.col   md rs1 rs2 31..25=2 14..12=1 6..0=0x2B
v2dloc.trect md rs1 rs2 31..25=5 14..12=1 6..0=0x2B

# set base address 2d register
v2dld     md rs1       imm12 14..12=2 6..0=0x2B
v2dst     md rs1       imm12 14..12=3 6..0=0x2B

# vv opearations
v2dadd.vv  md ms1 ms2 31..25=1  14..12=4 6..0=0x2B
v2dsub.vv  md ms1 ms2 31..25=2  14..12=4 6..0=0x2B
v2dcnv.vv  md ms1 ms2 31..25=4  14..12=4 6..0=0x2B
v2ddiv.vv  md ms1 ms2 31..25=8  14..12=4 6..0=0x2B
v2dmul.vv  md ms1 ms2 31..25=16 14..12=4 6..0=0x2B
v2dsmul.vv md ms1 ms2 31..25=48 14..12=4 6..0=0x2B

# vs operations
v2dadd.vs  md ms1 rs2 31..25=1  14..12=5 6..0=0x2B
v2dsub.vs  md ms1 rs2 31..25=2  14..12=5 6..0=0x2B
v2ddiv.vs  md ms1 rs2 31..25=8  14..12=5 6..0=0x2B
v2dmul.vs  md ms1 rs2 31..25=16 14..12=5 6..0=0x2B
```

Secvență de Cod 1: Fișierul de definire a extensiei RISC-V noi introduse.

În Secvența de Cod 1 am prezentat definirea instrucțiunilor în formatul *riscv-opcodes*. Formatul prevede definirea pe intervale de biți a valorilor. Pe lângă câmpurile cu valori fixe, pentru decodificarea instrucțiunii, există și câmpuri pentru operanzi. Ele sunt definite într-un fișier separat și definesc doar lungimea lor în biți. În instrucțiunile definite există două tipuri de regiștrii. Primul tip sunt regiștrii procesorului, acestea sunt definiți de *rs1* și *rs2*. Regiștrii interni ai acceleratorului încep cu litera **m**, de la matrice, și sunt reprezentați de *md*, *ms1* și *ms2*.

Ca și în cazul definiri instrucțiunilor am început prin a face *fork* de la proiectul oficial de compilator RISC-V, modificările sunt disponibile pe GitHub [26]. De la fișierele generate de *riscv-opcodes* am copiat definirile instrucțiunilor noastre și le-am adăugat în fișierele compilatorului. Au mai fost

necesare și alte modificări, precum legarea instrucțiunilor noi unei extensii RISC-V. Pentru simplitate am ales să le lăsăm extensiei *RV32I*. În acest fel instrucțiunile pot fi folosite de cele mai simple procesoare.

Prin oferirea suportului de compilator am simplificat utilizarea acceleratorului, oferind o experiență simplificată de dezvoltare. Folosirea de instrumente de dezvoltare comune permit integrarea în soluțiile software existente.

```
#ifndef MATRIX_ACCELERATOR_H
#define MATRIX_ACCELERATOR_H

#define MA_DEFINE_int8_t(ID, W, H) ({ \
        asm volatile \
        ( \
            "v2ddef8 x" _STR(ID) " , %[x], %[y]\n\t" \
            : \
            : [x] "r" (W), [y] "r" (H) \
        ); \
    })

#define MA_DEFINE_uint8_t(ID, W, H) ({ \
        asm volatile \
        ( \
            "v2ddefu8 x" _STR(ID) " , %[x], %[y]\n\t" \
            : \
            : [x] "r" (W), [y] "r" (H) \
        ); \
    })

#define MA_DEFINE_int16_t(ID, W, H) ({ \
        asm volatile \
        ( \
            "v2ddef16 x" _STR(ID) " , %[x], %[y]\n\t" \
            : \
            : [x] "r" (W), [y] "r" (H) \
        ); \
    })

#define MA_DEFINE_uint16_t(ID, W, H) ({ \
        asm volatile \
        ( \
            "v2ddef16 x" _STR(ID) " , %[x], %[y]\n\t" \
            : \
            : [x] "r" (W), [y] "r" (H) \
        ); \
    })

#define MA_DEFINE_int32_t(ID, W, H) ({ \
```

```
41          asm volatile \
42          ( \
43              "v2ddef32 x" _STR(ID) " , %[x], %[y]\n\t" \
44              : \
45              : [x] "r" (W), [y] "r" (H) \
46          ); \
47      })
48
49  #define MA_DEFINE_uint32_t(ID, W, H) ({ \
50          asm volatile \
51          ( \
52              "v2ddefu32 x" _STR(ID) " , %[x], %[y]\n\t" \
53              : \
54              : [x] "r" (W), [y] "r" (H) \
55          ); \
56      })
57
58  #define MA_LOC_RECT(ID, X, Y) ({ \
59      asm volatile \
60      ( \
61          "v2dloc.rect x" _STR(ID) " , %[x], %[y]\n\t" \
62          : \
63          : [x] "r" (X), [y] "r" (Y) \
64      ); \
65  })
66
67  #define MA_LOAD_REGISTER(ID, PTR) ({ \
68          asm volatile \
69          ( \
70              "v2dld x" _STR(ID) ", %[x]\n\t" \
71              : \
72              : [x] "o" (PTR) \
73          ); \
74      })
75
76  #define MA_STORE_REGISTER(ID, PTR) ({ \
77          asm volatile \
78          ( \
79              "v2dst x" _STR(ID) ", %[x]\n\t" \
80              : \
81              : [x] "o" (PTR) \
82          ); \
83      })
84
85  #define MA_VV_ADD(RID, S1ID, S2ID) asm volatile( "v2dadd.vv x" _STR(RID) ", x"
      _STR(S1ID) ", x" _STR(S2ID) );
86  #define MA_VV_SUB(RID, S1ID, S2ID) asm volatile( "v2dsub.vv x" _STR(RID) ", x"
      _STR(S1ID) ", x" _STR(S2ID) );
```

```
87 #define MA_VV_CNV(RID, S1ID, S2ID) asm volatile( "v2dcnv.vv x" _STR(RID) ", x"
       _STR(S1ID) ", x" _STR(S2ID) );
88 #define MA_VV_DIV(RID, S1ID, S2ID) asm volatile( "v2ddiv.vv x" _STR(RID) ", x"
       _STR(S1ID) ", x" _STR(S2ID) );
89 #define MA_VV_MULT(RID, S1ID, S2ID) asm volatile( "v2dmul.vv x" _STR(RID) ", x"
       _STR(S1ID) ", x" _STR(S2ID) );
90 #define MA_VV_SMULT(RID, S1ID, S2ID) asm volatile( "v2dsmul.vv x" _STR(RID) ", x"
       _STR(S1ID) ", x" _STR(S2ID) );
91
92 #define MA_VS_ADD(RID, S1ID, S2) ({ \
93         asm volatile \
94         ( \
95             "v2dadd.vs x" _STR(RID) ", x" _STR(S1ID) ", %[x]\n\t" \
96             : \
97             : [x] "r" (S2) \
98         ); \
99     })
100
101 #define MA_VS_SUB(RID, S1ID, S2) ({ \
102         asm volatile \
103         ( \
104             "v2dsub.vs x" _STR(RID) ", x" _STR(S1ID) ", %[x]\n\t" \
105             : \
106             : [x] "r" (S2) \
107         ); \
108     })
109
110 #define MA_VS_DIV(RID, S1ID, S2) ({ \
111         asm volatile \
112         ( \
113             "v2ddiv.vs x" _STR(RID) ", x" _STR(S1ID) ", %[x]\n\t" \
114             : \
115             : [x] "r" (S2) \
116         ); \
117     })
118
119 #define MA_VS_MULT(RID, S1ID, S2) ({ \
120         asm volatile \
121         ( \
122             "v2dmul.vs x" _STR(RID) ", x" _STR(S1ID) ", %[x]\n\t" \
123             : \
124             : [x] "r" (S2) \
125         ); \
126     })
127
128 #endif // MATRIX_ACCELERATOR_H
```

Secvență de Cod 2: Bibliotecă pentru accelerator.

Pentru a îmbunătății utilizarea și integrarea acceleratorului am implementat și o biblioteca pentru el. Implementarea este în Secvența de Cod 2. Este realizată pentru C și folosește *macros*. Oferă programatorului o abstractizare peste instrucțiunile de asamblor. Iar în spate se expandează prin instrucțiuni de *inline assembly*. Această bibliotecă a fost folosită în testele de performanță realizate de noi pentru studiul acceleratorului.

4.3 EXEMPLU

Pentru a exemplifica procesul de utilizare al acceleratorului ne vom folosii de un exemplu. Secvența de cod 3 prezintă un exemplu de programare în RISC-V, se omite scrierea datelor în memoria principală.

În exemplul se prezintă adunarea a două matrice de 4×4 cu date de tip întregi pe 32 de biți. În acest caz avem nevoie de trei matrice, două pentru operanzi și una pentru rezultat. Pentru transferul parametrilor avem nevoie ca valorile să fie în regiștrii procesorului. Acest lucru este realizat de primele două instrucțiuni care încarcă numărul *4* în doi regiștrii.

Pentru orice registru trebuie să definim tipul de dată, lungimea, lățimea și plasarea în memoria bidimensională. Pentru aceasta sunt necesare două instrucțiuni, aceste instrucțiuni se pot vedea la liniile: 4, 5, 8, 9, 12 și 13. Ordinea nu este relevantă, putem să definim mai întâi locația și ulterior tipul și dimensiunile sau invers. Instrucțiunile de tip *v2dloc.** definesc locația, iar instrucțiunile de tip *v2ddef** definesc tipul de dată împreună cu dimensiunile.

Pentru putea executa operația de adunare, trebuie să încărcăm datele în memoria internă. Pentru aceasta se folosește operația *v2dld*, liniile: 6 și 10. În acest moment matricele sunt definite și matricele de intrare au date valide.

La linia 15 se execută adunarea celor două matrice. După această instrucțiune rezultatul este în memoria internă. Cu ultima instrucțiune, ce de *store*, datele sunt scrise în memoria principală. Acest rezultat nu este șters din memoria acceleratorului și poate fi folosită ulterior.

```
1 addi x5, x0, 4 ; set width and height to 4
2 addi x6, x0, 4 ; load value in 2 registers
3
4 v2dloc.rect m0, x0, x0 ; place register m0 at 0, 0 in 2d memory and define as
    rectangle
5 v2ddef32 m0, x5, x6 ; set register m0 width and height
6 v2dld m0, x0, 0 ; load register m0 data from main memory to local memory
7
8 v2dloc.rect m1, x5, x6 ; place register m1 at 4, 4 in 2d memory and define as
    rectangle
9 v2ddef32 m1, x5, x6 ; set register m1 width and height
10 v2dld m1, x0, 64 ; load register m1 data from main memory to local memory
```

```
11
12 v2dloc.rect m2, x0, x6 ; place register m2 at 0, 4 in 2d memory and define as
       rectangle
13 v2ddef32 m2, x5, x6 ; set register m1 width and height
14
15 v2dadd.vv m2, m0, m1 ; m2 <- m1 + m0
16
17 v2dst m2, x0, 128 ; store register m2 from local memory to main memory
```

Secvență de Cod 3: Exemplu de utilizarea al accleratorului, în RISC-V.

5 ARHITECTURĂ

Acceleratorul este compus din cinci componente principale: *Decode Unit*, *Control Unit*, *DMA Unit*, *Vectorial Unit*, *Matrix Unit* și *Internal Memory*. În Figura 4 este prezentată arhitectura internă a acceleratorului. În secțiunile următoare vom prezenta scopul fiecărei componente și arhitectura internă a componentei.

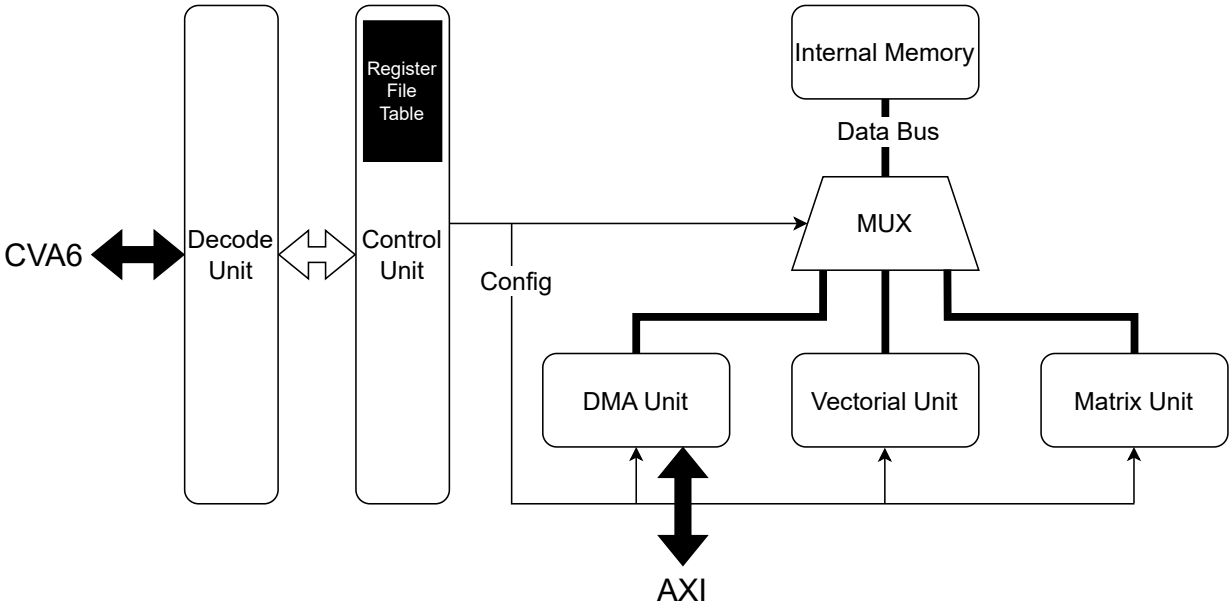

Figura 4: Arhitectura internă a acceleratorului

În Figura 4 sunt prezentate și două magistrale interne. Magistrala *Config* oferă informațiile despre regiștrii implicați în acea operație. Informațiile oferite sunt lungimea, lățimea, tipul de dată și locația în memoria bidimensională. Tot prin această interfață se alege și ce unitate va executa operația matricială cerută. Magistrala *Data Bus* oferă acces la memoria internă. Prin multiplexare se acorda acces doar unei singure componente. Prin interfața de date se trimit adrese și se primesc datele.

Acceleratorul are un parametru care pot fi configurat de utilizator în timpul implementării. Acest parametru este numărul de date care pot fi accesate în paralel. Importanța acestui parametru este dată de faptul că va impune și utilizarea mai multor unități aritmetice pentru a utiliza toate datele de care dispunem.

5.1 Decode Unit

Această unitate se ocupă de comunicația cu procesorul RISC-V și asigură protocolul CV-X-IF. Această unitate validează instrucțiunile pentru accelerator, verifică dacă instrucțiunea invalidă pentru procesor este validă pentru accelerator. Pe lângă validarea instrucțiunilor mai are ca scop și aducerea datelor din regiștrii procesorului. Pe baza tipului de instrucțiune va cere acei regiștrii, dacă este cazul, de la procesor.

Această componentă se numește *Decode Unit* deoarece execută și decodificarea instrucțiuni. Dacă la intrare are o instrucțiune în format RISC-V, în această unitate ea este decodificată și convertită în semnale simple, care sunt folosite intern. Tot această unitate se ocupă și cu calcularea adresei, adresarea este în mod indexat, pe lângă adresa din registru se transmite și un imediat. În această etapă adresa este calculată.

Când toate datele necesare instrucțiuni au fost colectate, sunt transmise la *Control Unit*. Aceasta se va ocupa cu executarea lor și va semnalizarea când operația s-a terminat. La acest eveniment unitatea de control are de semnalizat finalizarea execuției către procesor. Până în momentul în care instrucțiunea nu este terminată acceleratorul nu va accepta noi instrucțiuni.

5.2 Control Unit

Această componentă este componenta principală de control. Ea deține toate informațiile despre regiștrii matriciali definiți software. În *Register File Table* sunt stocate informațiile despre regiștrii: lungime, lățime, tipul de dată și coordonatele în memoria bidimensionala. Suplimentar sunt stocați și biți care semnalizează dacă registrul a fost definit și se află în memorie. Aceste informații sunt actualizate pe baza operațiilor executate.

Informațiile despre regiștrii nu sunt accesibile unităților de execuție. În momentul în care o operație trebuie executată, doar datele despre regiștrii implicați în acea operație vor fi partajate, în mod de citire. Modificarea acestei tabele este posibilă doar de către *Control Unit*.

O altă funcție pe care o are de îndeplinit este aceea de a alege unitatea optimă pentru executarea operației. Dacă este o operație la memorie va fi aleasă *DMA Unit*. Pentru operațiile de înmulțire de matrice și convoluții se va alege unitatea *Matrix Unit*.

Deși o componentă simplistă asigură funcționalități de control și sincronizare pentru accelerator.

5.3 DMA Unit

Scopul acestei unități este de a executa operații cu memoria principală. Această unitate aduce date și le scrie în memoria internă sau scrie datele din memoria interna în memoria principală.

După cum se poate observa și în Figura 6 această componentă are o arhitectură internă simplă. O componentă de tip DMA comunică cu memoria principală. O altă componentă se ocupă cu generarea adreselor în memoria internă.

Adresa din memoria principala este oferită printr-o instrucțiune. Adresa este de tip indexat, dar *Decode Unit* se ocupă cu calcularea ei. Pe baza informațiilor despre un registru (lungime, lățime și tipul de dată) putem să calculăm numărul de octeții implicați în operația cu memoria principală.

Componenta care generează adresele ține cont de numărul de date care pot fi scrise în paralele, astfel incrementând corespunzător adresele. Acceleratorul a fost gândit pentru a lucra cu limbajul de programare C. Modul de accesare a metricilor și scriere în memoria internă este *Row Major*. În acest mod matriciale sunt stocate în memorie consecutiv linie după linie. Pentru a putea fi folosite în accelerator matricele trebuie liniarizate, trebuie declarat un vector unidimensional cu lungimea egală cu *lungime* $\times$ *lățime*.

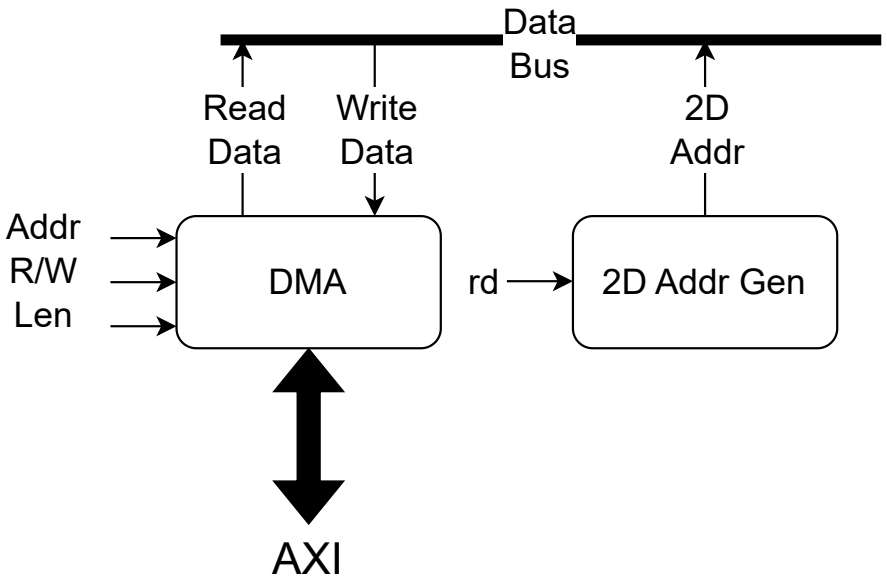

Figura 5: Arhitectura internă a unității DMA

Componenta de tip *DMA* a fost implementată de noi. În Figura 6 este prezentata structura internă. Modulul a fost proiecta pentru a folosii funcționalitățile oferite de AXI, în special *Burst*. În AXI putem să cerem maxim 4KB de date într-un transfer. Acesta este motivul pentru care am ales cozi de această dimensiune.

Elementele *Tranzaction Sequnecer* asigură împărțirea unei tranzacții în mai multe de 4KB. Modulul respectă protocolul AXI și oferă și aliniară tranzacțiilor în paginile de 4KB.

Cozile de date sunt legate direct la canalele de date, iar mai departe datele sunt citite sau scrise printr-un protocol de tip *Valid-Ready*.

Se poate observa și simetria între partea de citire și cea de scriere. Cele două operații sunt identice, doar diferă sensul datelor.

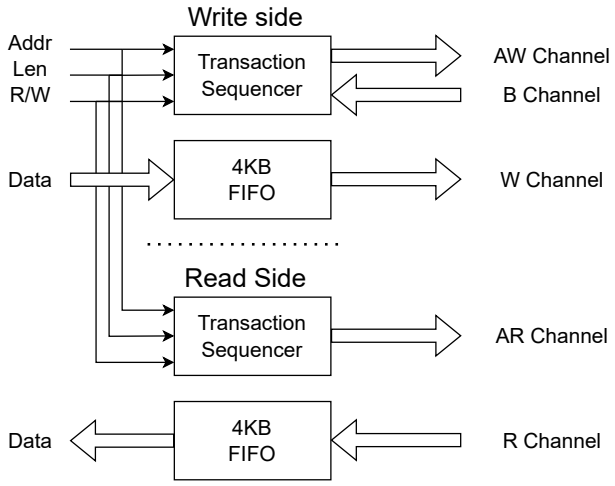

Figura 6: Arhitectura internă a controlerului DMA

5.4 VECTORIAL UNIT

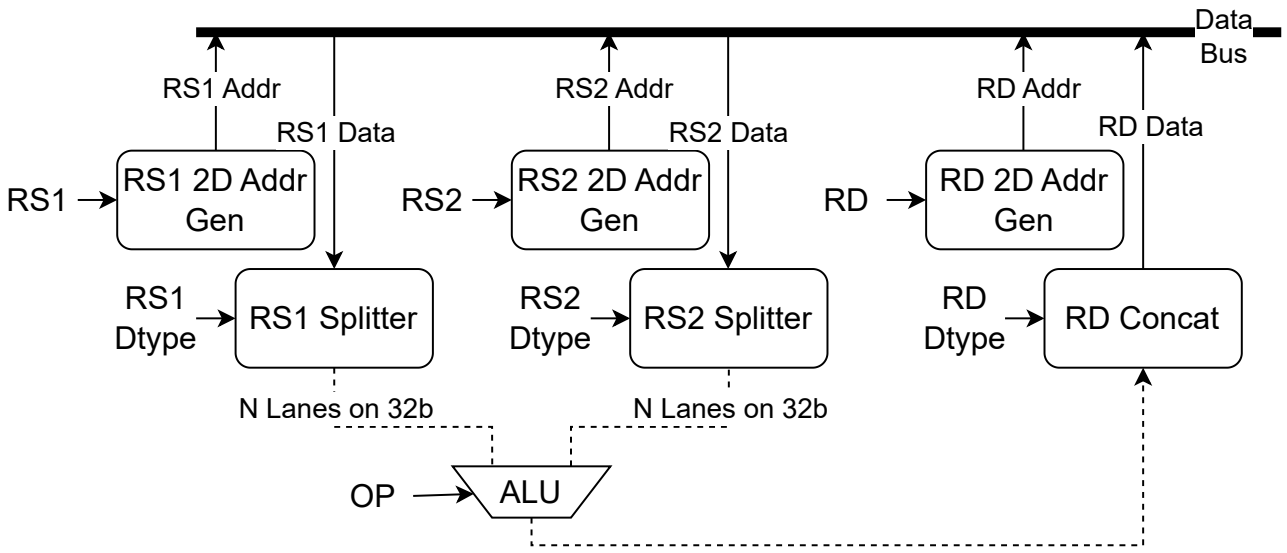

Figura 7: Arhitectura internă a unității vectoriale

Scopul acestei unități aritmetice este acela de a executa operații matriciale care pot fi asimilate ca și operații vectoriale. Operațiile aritmetice pe care le poate executa sunt: adunarea, scăderea, înmulțirea element-cu-element și împărțire element-cu-element. Tot această unitate va executa și operațiile cu un scalar, adică aplicarea unei operații de scalare fiecărui element din matrice cu aceeași valoare.

În Figura 7 este prezentată structura internă a *Vectorial Unit*. Structura internă cuprinde generatoare de adrese, elemente de divizat/concatenat datele și unități aritmetice.

Tipurile de date suportate sunt 8, 16 și 32 de biți. Suportând trei tipuri diferite de date, dar care sunt și proporționale am ales să stocăm mai multe numere într-un element care poate stoca 32 de biți. Dacă avem numere pe 8 biți și memoria are elemente pe 32 de biți, vom stoca într-un element patru numere de 8 biți. Am ales această metodă pentru a utiliza eficient memoria, evitând a stoca date inutile. Această metodă de stocare are și avantaje la operațiile cu memoria, datele sunt aduse și stocate direct, fără alte procesări în acele operații.

Datele fiind concatenate, la operații aritmetice ele trebuie divizate, acesta este scopul modulelor de tip *Splitter*. Aceste unități primesc tipul de dată și un element de la memorie. Intern aritmetica se va desfășura pe 32 de biți. Unitățile care împart datele vor avea grijă să separe un element în cazul în care tipul de dată are mai puțini biți. Pentru a fi corecte pentru unitățile aritmetice datele sunt extinse la 32 de biți. Deoarece datele sunt cu semn sau fără semn, vom ține cont de acest aspect în cazul extinderii bitului de semn.

Modulul de tip *Concat* are scopul de a concatena rezultatele pentru a putea fi scrise corect în memoria internă. Dacă rezultatul este pe un număr redus de biți, rezultatul va fi trunchiat la numărul acela de biți. Datele sunt concatenate în ordinea corectă pentru a putea fi compatibile cu limbajul de programare C și pentru a putea fi folosite ulterior.

Important de observat este că pentru fiecare operand tipul de date este configurat independent. Numărul de operații aritmetice va fi același indiferent de tipurile de date implicate în operație. Tipul de dată influențează doar numărul de operații la memoria internă. Acest mod de lucru independent permite o mai mare flexibilitate pentru utilizatorul final și mărește spectrul de aplicații. Singura condiție pentru această unitate este ca numărul de elemente să fie aceleași pentru cele trei matrice implicate.

Deoarece avem acces la mai multe date în paralel am optat pentru a avea același număr de unități aritmetice. Astfel creștem numărul de date procesate pe ciclu de ceas.

5.5 Matrix Unit

Matrix Unit are ca scop efectuare de operații de înmulțire de matrice și convoluție. În versiunea actuală convoluția nu este implementată. Figura 8 prezintă arhitectura internă a acestei unități. Se poate observa similaritatea cu *Vectorial Unit*. Principiul de operare este același, sunt aduse datele, divizate, se execută operațiile și la final sun concatenate și scrise în memoria internă.

Pentru executarea operațiilor matriciale am optata pentru folosirea metricilor sistolice. Această topologie permite efectuarea unui număr ridicat de operații pe ciclu de ceas. La fiecare tact de ceas,

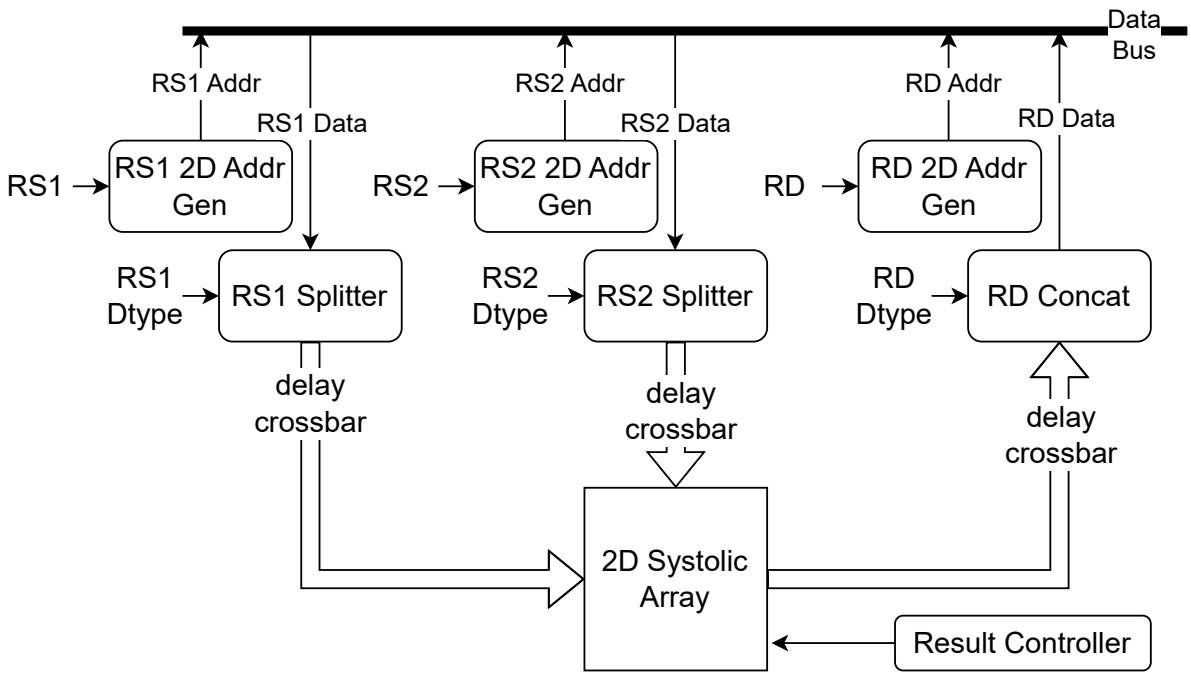

Figura 8: Arhitectura internă a unității matriciale

un element din matricea sistolică calculează înmulțirea a doi operanzi și adunarea cu rezultatul intermediar din acel element.

Și în această unitate aritmetica internă este pe 32 de biți. De acea în continuare este nevoie de modulele de *Splitter* și *Concat*. Iar dimensiunea matricei sistolice este direct proporțională cu numărul de date pe care le putem citii în paralel. Dimensiunea pe fiecare direcție este egală cu acest număr de date disponibile pe tact de ceas.

Matricea sistolice necesită o sincronizare atentă. Pentru aceasta la intrarea în matrice datele sunt întârziate corespunzător. Elementul de tip *Delay Crossbar* asigură întârzierea cu un tact de ceas față de data linia anterioară. Este implementat prin bistabil de deplasare de lățime 32 de biți. În cazul rezultatelor realizează o sincronizarea, datele să fie la momentul potrivit la intrarea în modulul de concatenare.

Legat de modul de acces al datelor, memoria polimorfică folosită ne ajută, atât la aducerea datelor și scrierea lor. Pentru primul operator la fiecare operație la memorie sunt aduse date de pe aceiași coloană. Primul operator trebuie să ofere la fiecare ciclu de ceas date valide pe linie, de acea am ales acest mod de parcurgere. Pentru al doilea operator datele sunt aduse de pe aceiași linie, la fiecare tact avem nevoie de elementele de pe colană. O coloană include mai multe datele de pe mai multe rânduri, iar un rând include date de pe mai multe coloane. Rezultate sunt scrise în mod coloană. Rezultatele sunt dependente de primul operator, aceasta este implementarea noastră.

Modulul de *Result Controller* are ca scop calcularea momentului când este rezultatul valid în

fiecare element din matricea sistolică, pe parcurs se vor regăsii rezultate intermediare. La acel moment de timp, rezultatul va fi colectat și acel element trebuie resetat, trebuie scris zero în memoria acumulatorului.

Matricea sistolică calculează rezultatele în rafale, la fiecare rafală o sub-matrice egală cu dimensiunea matricei sistolice este calculată. De acest model de scriere se ocupă unitatea care calculează adresele, ea dispune de toate informațiile pentru a calcula adresa corectă pentru fiecare calup de date.

5.6 Internal Memory

Memoria internă este bazată pe o memorie polimorfică bidimensională. Principalul avantaj oferit de acest tip de memorie este acela de a putea accesa date multiple pe aceiași linie sau aceiași coloană. Aceste modele de acces a datelor ne permit simplificarea efortului de implementare dar și reduc necesitatea altor zone de memorie tampon pentru rearanjarea datelor.

Numărul de date accesibile în paralel este configurabil la momentul implementării. Acest parametru are impact și asupra celorlalte componente, lățimea magistralei crește cu acest parametru și numărul de unități aritmetice crește proporțional.

Internal Memory este un adaptor peste modulul de memorie polimorfică. Singurul scop este acela de a trece de la magistrala internă definită de noi la semnalele folosite de acest modul.

Memoria polimorfică este configurată cu două interfețe de citire și una de scriere. Astfel în același timp putem să citim cei doi operatori simultan și putem să scriem un rezultat obținut anterior.

Configurația folosită de noi este aceea de memorie bidimensională pătratică de 1024×1024 elemente. Lungimea unui element este de 32 de biți, egal cu lungimea celui mai lung tip de dată suportat. Astfel această memorie dispune de 4MB de date utile. Intern memoria polimorfică multiplică memoriile pentru fiecare interfață de citire.

6 EXPERIMENTE

Pentru accelerator am urmărit să îl analizăm în trei direcții:

1. optimizarea obținută comparativ cu un procesor RISC-V.
2. resursele utilizate pe un FPGA.
3. frecvența maximă pe care o putem atinge pe un FPGA.

Pentru extragerea acestor informații a trebuit să ne realizăm un sistem de testare. Sistemul definit de noi este folosit atât în simulatorul RTL cât și în uneltele pentru FPGA. Sistemul de test este prezentat în Figura 9.

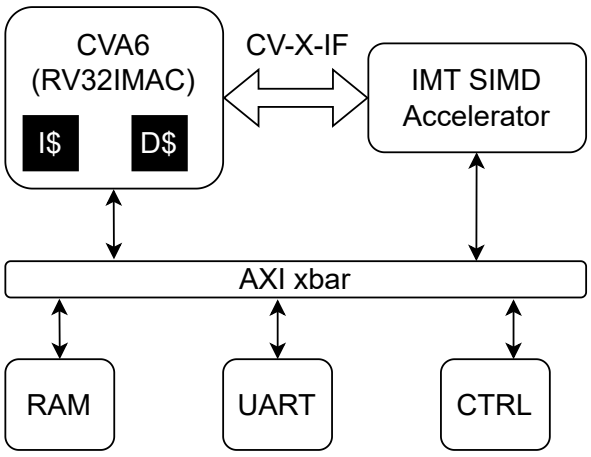

Figura 9: Arhitectura sistemului de test

Sistemul de testare include un procesor RISC-V. Procesorul ales este unul din surse libere și se numește CVA6 și este susținut de *Open Hardware Group* [27]. Configurația pe care am ales-o este RV32I cu extensiile *MAC_zicsr*. Procesorul este pe 32 de biți. Pe lângă instrucțiunile de bază are implementate în hardware și operațiile de înmulțire și împărțire. Iar extensia *zicsr* ne oferă accesul la regiștrii de control și stare. Procesorul are și cache pentru instrucțiuni și date. Configurația aleasă este un cache de 4KB pentru instrucțiuni. Cache-ul pentru date este de 8KB și este de tip *write-through*, datele scrise în cache sunt scrise automat și în memoria principală. Un rezumat al configurației procesorului se găsește în Tabelul 3.

Conectat la procesor este acceleratorul dezvoltat de noi. Conexiunea este realizată prin interfața CV-X-IF. Procesorul și acceleratorul sunt conectate la aceeași magistrală și au același spațiu de adrese. Componenta RAM din sistem are un dublu rol, atât RAM cât și ROM. La începutul execuției, programul va fi scris în memoria RAM.

Procesor RISC-V	CVA6
Extensii	IMAC_zicsr
L1 I$	4KB
L2 I$	8KB

Tabel 3: Configurația procesorului CVA6

Componenta UART este folosită pentru a putea avea o consolă pentru sistem. În acest fel putem să obținem informații de la sistem, cu o complexitate redusă. Această componentă în funcție de simulare sau FPGA are două implementări diferite. Pentru FPGA am folosit un IP de UART de la Xilinx. Pentru simulare am implementat o versiune a IP-ului de la Xilinx de simulare, regiștrii sunt aceiași dar pentru afișarea caracterului folosim funcțiile de afișare din Verilog. Pentru a simplifica utilizarea acestei componente, am modificat biblioteca *printf*, mai exact funcția *_put_char* care se ocupă cu afișarea unui caracter și va trimite acel caracter către UART.

Componenta CTRL este pentru control. În modul de simulare va genera un eveniment care va oprii simularea. Tot această componentă încapsulează și un numărător pe 64 de biți, pe care îl vom folosii pentru măsurarea performanțelor. La începutul secțiuni de cod pe care vrem să o monitorizăm pornim numărătorul, iar la final îl oprim și colectăm rezultatul.

6.1 Simulare RTL

Prin simulare RTL urmărim să extragem informații legate de optimizarea operațiilor matriciale comparativ cu un procesor clasic. Testele pe care le urmărim sunt pe diverse tipuri de operații matriciale, tipuri de date, dimensiuni de matrice și numărul de date accesibile în paralel.

Tipurile de date testate sunt numere întregi pe 8, 16 și 32 de biți. Operațiile matriciale testate sunt adunarea (*Addition*), scăderea (*Subtraction*), înmulțire de matrice (*Dot Product*) și înmulțirea element-cu-element (*Cross Product*). Modul de testare este același pentru fiecare test.

Testul începe prin inițializarea matricelor operand cu valori aleatorii. După aceia aceste matrice sunt definite și încărcate în memoria internă a acceleratorului. Operația aritmetică este executată, după care rezultatul este scris în memoria principală. Rezultatul este calculat și pe procesor folosind algoritmi simpli pentru acea operația. La final memoria cache este invalidată, pentru a avea acces la valorile calculate de accelerator, în acest fel comparăm rezultatul obținut de accelerator și rezultatul în software. Această verificare realizează și o verificare funcțională a acceleratorului.

Pentru a analiza optimizarea, măsurăm timpul operației aritmetice pe accelerator și pe procesorul RISC-V. Timpul pe accelerator ia în calcul definirea matricelor, operațiile cu memoria principală și operația aritmetică. Măsurăm toate operațiile pentru a evalua timpul resimțit de utilizator când

folosește acceleratorul. Timpul pentru operația aritmetică este măsurat și el pentru a avea un punct de referință. Operația aritmetică este măsurată înainte de invalidarea cache-ului, astfel prevenim cazul nefavorabil în care nu avem date în memoria cache.

În scenariul de simulare considerăm memoria RAM una rapidă, asemănătoare cu un nivel 2 de cache. În acest mode putem să evaluăm performanța maximă pe care o putem obține.

6.2 TESTE FPGA

Pentru testele pe FPGA vom folosii sistemul VCU128 de la compania Xilinx, un sumar al resurselor este în Tabelul 4. Pentru scenariul acesta, deoarece dispunem de HBM, vom folosii acest tip de memorie pe post de memorie RAM. În această configurație putem măsura și impactul acestor memorii de performanță ridicată asupra unui sistem.

System Logic Cells (K)	2852
HBM DRAM (GB)	8
DSP Slices	9024
Block RAM (Mb)	70.9
Ultra RAM (Mb)	270

Tabel 4: Resurse VCU128, sursă [28]

FPGA-ul ales are și memorii Ultra RAM (URAM). Acest tip de memorie comparativ cu unul clasic de tip Block RAM (BRAM) are o densitate mai mare și a fost proiectată să ruleze la frecvențe mai mari. În testele noastre vom analiza și impactul tipului de memoria asupra frecvenței maxime pe care o putem obține.

Pe FPGA în urma etapei de sinteză cu Vivado 2024.1 vom extrage resursele utilizate, atât de sistemul final, dar cât și de accelerator. Frecvența de sinteză este setată la 100MHz. Pe baza acestei valorii și a timpului pe calea critică vom putea calcula frecvența maximă. Formula pe care o vom utiliza este $F_{max} = 1000/(10 - Worst\_Slack)$.

7 REZULTATE

În acest capitol vom prezenta rezultate în urma testelor și vom oferii o interpretarea a acestora. Capitolul are două secțiuni, unul în care analizăm rezultatele de la implementare fizică. În al doilea analizăm rezultatele de performanță, din perspectiva timpului de rulare.

7.1 IMPLEMENTARE FIZICĂ

Pentru FPGA trebuie să urmărim să ne adaptăm circuitul cât mai mult pe resursele existente. În cazul FPGA-ului folosit memoriile existente (BRAM și URAM) sunt native cu interfața True Dual Port (TDP) [29]. O memorie TDP are două porturi pentru aceeași memorie. Pentru fiecare interfață se poate scrie sau citii independent de cealaltă.

Designul de la care am pornit al memoriei polimorfice implica memorii Simple Dual Port (SDP), două porturi dar pe unul doar se scrie iar pe al doilea doar se citește. Dar acestă memorie se poate implementa și cu memorii TDP, deoarece avem de citit maxim două date simultan. Prima parte se va concentra pe acest design neoptimizat pentru FPGA. În a doua parte vom analiza design-ul optimizat pentru FPGA și vom prezenta și modificările care au fost realizate.

7.1.1 Memorii Simple Dual Port

Acest test a urmărit analiza resurselor utilizate pe un FPGA și frecvența maximă pe care o putem obține. Pentru început am analizat impactul pe care îl are numărul de date accesibile în paralel. Acest parametru va avea impact asupra resurselor din accelerator. Cele două unități aritmetice depind direct proporțional de acest parametru.

În Tabelul 5 sunt prezentate rezultatele de utilizare obținute în urma etapei de sinteză. În acest tabel am inclus atât resursele utilizate de un sistem funcțional care integrează acest accelerator, și resursele folosite doar de accelerator. Sistemul folosit este cel prezentat în Figura 9. FPGA-ul folosit dispune de două tipuri de memorie internă: BRAM și URAM, astfel noi am testat și acest scenariu, folosind un ambele tipuri de memorie pentru memoria internă a acceleratorului.

Am variat numărul de date în paralelele e la 4 la 32, la fiecare test numărul dublându-se. Se observă că numărul de Digital Signal Processor (DSP), aproximativ, se cvadruplează de un test la altul. Acest lucru este cauzat de matrice sistolică, care depinde pe fiecare direcție de numărul de date în paralele. Prin urmare, dacă dublăm acest număr pe fiecare latură, în final matricea se va cvadrupla. Dublând numărul de date în paralele se dublează și numărul de unități aritmetice în unitatea vectorială. Numărul de DSP-uri are și o componentă constantă, realizată de DSP-urile din procesor și

Lanes	URAM	Demonstrator					Accelerator				
		LUT [K]	FF [K]	BRAM	URAM	DSP	LUT [K]	FF [K]	BRAM	URAM	DSP
4	DA	47.8	33.9	38	512	212	17.0	13.9	4	512	208
8	DA	65.4	54.5	42	512	800	34.5	34.5	8	512	796
16	DA	161.7	141.2	50	512	3128	130.9	121.3	16	512	3124
32	DA	1091.6	481.2	64	512	8642	951.0	461.2	30	512	8640
4	NU	55.8	33.8	1894	0	212	25.1	13.8	1860	0	208
8	NU	73.4	54.5	1898	0	800	42.0	34.6	1864	0	796
16	NU	164.2	141.3	1904	0	3128	132.9	121.3	1870	0	3124
32	NU	1096.4	481.6	1920	0	8642	955.8	461.6	1884	0	8640

Tabel 5: Resursele utilizate pe Xilinx VCU128 de accelerator și demonstrator.

unitățile care calculează adresele.

Numărul de bistabili (fli-flop - FF) se dublează cu dublarea numărului de date paralele. Acest lucru este cauzat de arhitectura de tip *pipeline* folosită în numeroase componente. Elemente de stocare din calea de date se dublează și ele. Doar elementele din calea de control nu se modifică, deoarece ele depind de dimensiunea memoriei interne. Dependent de datele în paralel este și numărul de Look Up Tabels (LUT). Și acest număr se dublează, cauzat de căile de date.

Memoriile rămân aproape constante. În cazul URAM aceasta rămân constante fiind folosite doar în memoria internă a acceleratorului, care nu se schimba. Și memoriile BRAM folosite în memoria internă nu se modifică. Numărul de memorii BRAM are o componentă constantă și una variabilă. Constant în utilizarea lor este memoria cache folosită de procesor și memoria internă, când este cazul. Pe partea de memorii variabilă se află niște cozi interne folosite de accelerator, care sunt dependente de numărul datelor paralele.

Dacă comparăm LUT-urile între implementările cu URAM și cele BRAM observăm o diferență de aproximativ 20% între ele. Dar și numărul de URAM utilizate este mai redus, decât cel de BRAM, pentru aceeași scop. Acest lucru se cauzează diferenței constructive între cele două tipuri de memorie. Memoriile URAM au o capacitate mai mare, per bloc, decât BRAM. O primitivă de URAM are 72 de biți lățime și 4K adrese, iar o memorie BRAM are 36 de biți lățime și 1K adrese. O memorie URAM este de o capacitate de 8 ori mai mare. Numărul scăzut de LUT-uri se datorează faptului că memoriile URAM au fost proiectate să fie înlănțuite mai ușor, astfel mai multe module sunt grupate apropiat pentru a reduce logica de interconectare. În contrast memoriile BRAM sunt distribuite pe FPGA și consumă mai multă logică pentru interconectare.

Un alt test a fost să analizăm frecvența maximă pe car o putem obține. În Tabelul 6 am analizat frecvența maximă dacă variem numărul de date în paralel și impactul pe care îl are tipul de memorie folosit. În același tabel am inclus datele despre sistem și accelerator.

Componenta care limitează frecvența maximă a sistemului este procesorul RISC-V folosit. Pe de

Lanes	URAM	Demonstrator	Accelerator
		F Max [MHz]	F Max [MHz]
4	Da	133.65	196.54
8	Da	131.60	197.78
16	Da	135.28	194.97
32	Da	66.63	66.63
4	Nu	143.23	196.54
8	Nu	147.28	156.49
16	Nu	144.68	156.49
32	Nu	66.63	66.63

Tabel 6: Frecvența maximă în diverse configurații pentru demonstrator și accelerator

altă parte acceleratorul reușește să obțină o frecventă de peste 150 de MHz. În cazul în care avem mai mult de 32 de date paralele, frecvență scade considerabil. Din analiza rapoartelor de timp se observă cum complexitatea caii de control al matricei sistolice afectează performanța. Ce este important de observat, acceleratorul beneficiază de folosirea de URAM. În acest caz obținem frecvențe peste 190MHz.

Lanes	URAM	LUT	FF	BRAM	URAM	F Max [MHz]	BW [GB/s]
4	Da	2426	72	0	512	202.06	3.23
8	Da	2554	94	0	512	199.32	6.38
16	Da	20976	148	0	512	194.97	12.48
32	Da	38937	553	0	512	162.28	20.77
4	Nu	1338	48	1856	0	215.66	3.45
8	Nu	2240	93	1856	0	214.68	6.87
16	Nu	18736	187	1856	0	211.06	13.51
32	Nu	39686	723	1856	0	194.17	24.85

Tabel 7: Rezultate de sinteză memorie polimorfică, pe VCU128

Am analizat și memoria internă, am urmărit resursele utilizate și frecvența maximă pe care o putem obține de la ea. În Tabelul 7 am prezentat aceste rezultate, cazurile analizate sunt același ca pentru accelerator. Se poate observa că acceleratorul este mai lent decât memoria, cauza principală sunt căile de date.

Se poate observa în Figurile 10 și 11 frecvența maximă pe care o poate obține acceleratorul este sub cea a memorie polimorfice. Iar demonstratorul final are frecvența sub cea a acceleratorului. În cazul demonstratorului, pe baza analizelor rapoartelor, se observă că procesorul ales, CVA6, este cel care limitează frecvență maximă a sistemului, se poate observa palierul în ambele figuri. Când numărul de date în paralel crește la 32, din cauza caii de control pentru matricea sistolică, acceleratorul pierde din frecvența maximă pe care o poate obține.

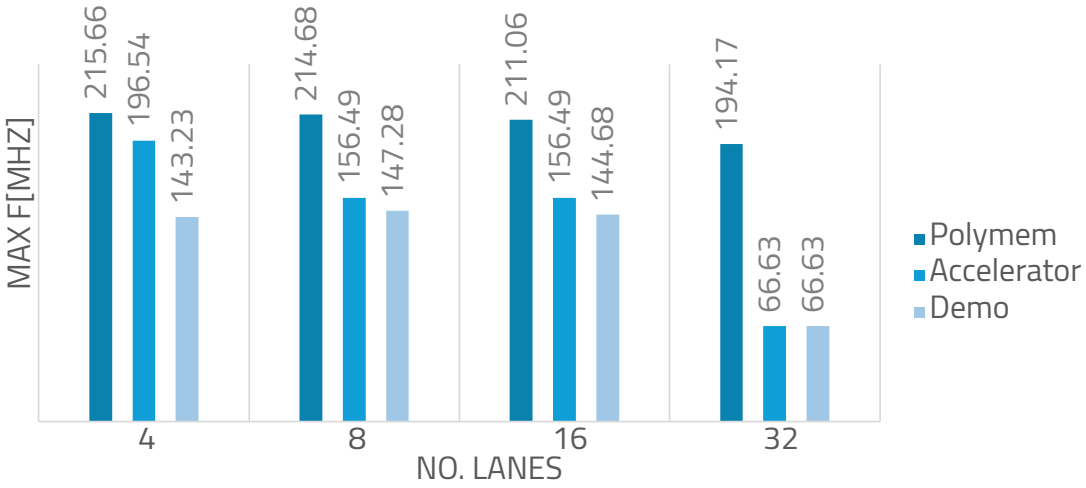

Figura 10: Comparație a frecvenței maxime, folosind BRAM

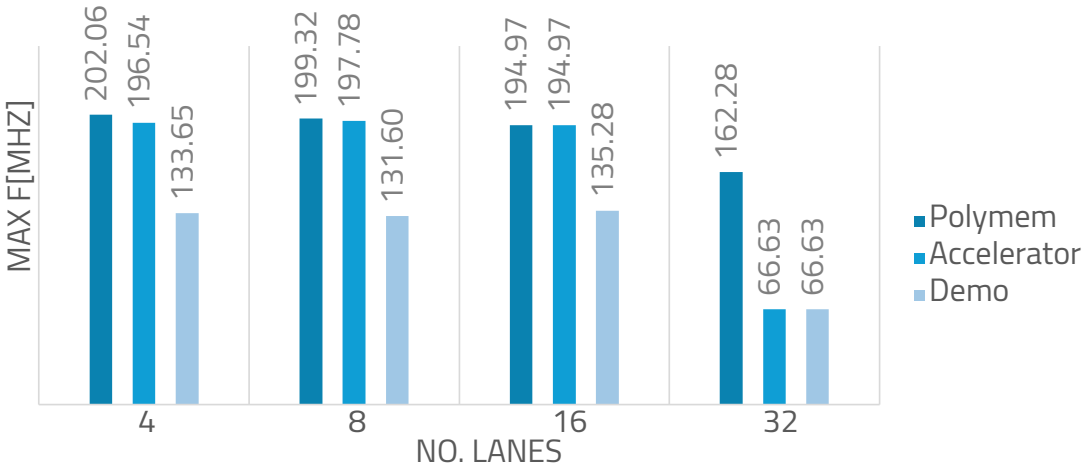

Figura 11: Comparație a frecvenței maxime, folosind URAM

În Figura 12 am comparat frecvența maximă pe care o poate obține acceleratorul folosind cele două tipuri de memorie posibile pe FPGA: URAM și BRAM. La 32 de date în paralel tipul de memorie nu are impact asupra acestei metrici. Se poate observa ce memoria URAM ajută la obținerea unor frecvențe mai mari. Principalul motiv este densitatea mai mare de stocare a memoriilor URAM, când avem memorii de capacitate ridicată calea de date devine mai scurtă. Tot celule URAM sunt grupate pe FPGA, comparativ cu memoriile BRAM care sunt distribuite pe FPGA, ceea ce duce la timpi de propagare ridicați pe căile de interconectare. Pe baza acestor avantaje, memoriile URAM ne ajută să obținem un accelerator care poate să livreze performanțe mai ridicate.

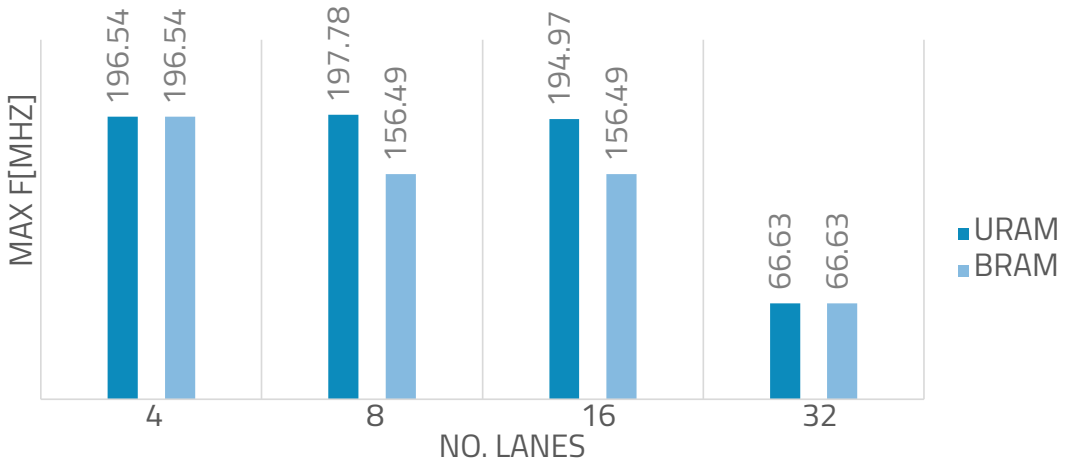

Figura 12: Comparație a frecvenței maxime a acceleratorului, pe baza tipului de memorie folosită

7.1.2 Memorii True Dual Port

Memoriile TDP reprezintă o soluție bună în cazul acceleratorului deoarece oferă posibilitatea de citire a două date simultan. Pentru operația de scrie oferă un port de scriere suplimentar. Folosirea acestei memorii a necesitat modificări în design. În urma testelor funcționale design-ul a trecut testele cu aceste modificări. Testele de performanță nu au fost afectate deoarece modificările au vizat doar memoria și am urmărit să păstrăm același comportament, nu a fost nevoie să modificăm nimic în restul acceleratorului.

Cele trei operații (două de citire și una de scriere) nu se pot întâmpla în același perioadă. Am optat ca memoria să meargă la o frecvență de două ori mai mare decât cea sistemului. În această manieră o jumătate din perioada sistemului executăm citirea și a doua jumătate executăm scrierea. Această ordine nu este arbitrară.

Adresele pentru operațiile de citire și scriere sunt diferite deci la fiecare perioadă datele de ieșire se schimbă, dar, pentru sistem, ele trebuie să rămână stabile două perioade, ca se sincronizeze cu frecvența sistemului. La ieșirea din modulul de memorie polimorfică și după rearanjarea datelor, am

poziționat bistabile cu semnal de activare din două în două perioade și sincronizat cu operația de citire.

Pentru accesul la memorie am optat să alternăm operațiile și operanzii pentru scriere și citire. Într-o perioadă ajung datele și cererile de citire și perioada următoare cele de scriere. Pentru funcționarea corectă a memoriilor am avut nevoie și de o sincronizarea atentă.

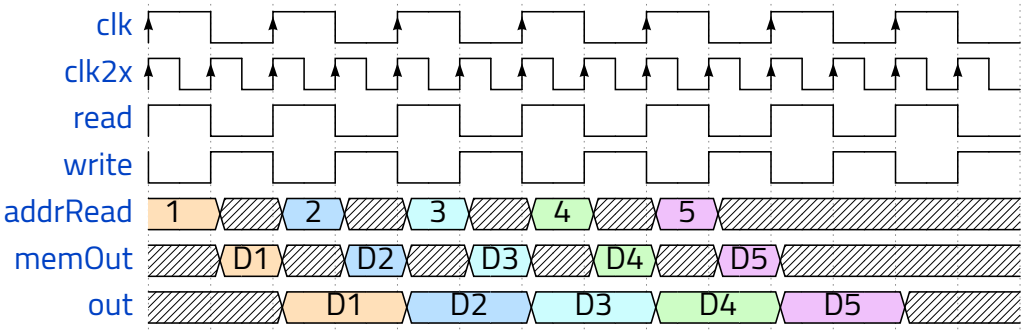

Figura 13: Diagramă de timp pentru sincronizarea memorie TDP cu sistemul

În Figura 13 am prezentat o diagrama de timp care reprezintă sincronizarea folosită între memorie și sistem. Ceasul memorie și ceasul sistemului sunt sincrone, iar reportul de perioadă este de 2:1, ceasul memorie este de două ori mai rapid. Memoriile folosite au bistabile la ieșire, deci orice rezultat de la citire apare după o perioadă.

Figura 13 are ceasul sistemului reprezentat de *clk* și ceasul memorie de *clk2x*. Semnalele *read* și *write* indică perioada în care se execută operațiile de citire și scriere. Acest două semnale sunt în antifază, iar operația de citire se întâmpla în prima perioadă din ceasul sistemului. Am ales această metodă pentru a sincroniza citirea din memorie cu cea din sistem.

Figura 13 prezintă adresa de citire (*addrRead*) și datele la ieșirea memoriei (*memOut*). În cazul datelor la ieșirea memorie, hașurile reprezintă date care nu sunt utile pentru această operație. Iar datele din *out* reprezintă formule de undă de care acceleratorul ar nevoie pentru a funcționa, datele disponibile o perioadă de ceas mai târziu și stabile pentru două perioade. Acest comportament s-a obținut folosind semnalul de scriere pe post de semnal de activare al bistabilelor de ieșire. Acest semnal întârzie datele cu încă o perioadă și le ține stabile pentru două perioade ale memoriei.

În Tabelul 8 am prezentat rezultatele de sinteză pentru design-ul care folosește memorii TDP. Pe lângă resursele utilizate am calculat și frecvențele maxime pe care le pot obține acceleratorul și memoria polimorfică.

Prin această optimizare pentru FPGA rezultatul era cel preconizat, o înjumătățire a modulelor de memorie. Numărul de memorii URAM se reduce de la 512 la 256, exact jumătate. În cazul memoriilor BRAM numărul se reduce de la 1856 la 928. Datorită bistabilelor inserate la ieșirea blocului de memorie, numărul e bistabile este mai mare. Creșterea este egală cu numărul de date accesibile în

Lanes	URAM	Accelerator					Memorie polimorfică					
		LUT [K]	FF [K]	BRAM	URAM	Max F [MHz]	LUT [K]	FF	BRAM	URAM	Max F [MHz]	Max Bandwidth [GB/s]
4	DA	17.6	14.1	4	256	144.91	1.2	72	0	256	215.66	3.45
8	DA	35.1	35.1	8	256	144.89	4.5	91	0	256	212.86	6.81
16	DA	127.8	122.2	8	256	144.91	23.7	155	0	256	214.68	13.74
32	DA	961.5	463.3	28	256	66.91	36.9	675	0	256	219.97	28.16
4	NU	19.9	14.1	932	0	144.91	3.7	48	928	0	265.39	4.25
8	NU	37.7	35.1	936	0	144.89	6.2	91	928	0	264.97	8.48
16	NU	126.8	122.2	1038	0	144.91	23.1	155	928	0	467.73	29.93
32	NU	962.3	463.4	1052	0	66.91	36.9	675	1024	0	308.55	39.49

Tabel 8: Utilizare FPGA și frecvență maximă design cu memorii TDP

paralel.

Frecvența maximă a acceleratorului nu mai este influențată de tipul de memorie. Cauza este prezența bistabilelor la ieșirea din memoria polimorfică, acest lucru rupe calea critică. Interesant este că numărul de date în paralele nu are un impact asupra frecvenței maxime. Singurul caz particular este pentru 32 de date paralele, cauza rămânând calea de control a matricei sistolice.

Memoria polimorfică este influențată pozitiv de memoria TDP. În cazul cu URAM frecvența maximă rămâne în jurul frecvenței de 210MHz și nu are o dependință directă cu numărul de date în paralele. Cu acest tip de memorie lățimea de bandă maximă crește față de cazul SDP. O îmbunătățire semnificativa se observă la memoriile BRAM unde frecvența maximă este mult peste cea din cazul SDP. Iar lățimea de bandă maximă este cu aprox. 30% mai mare.

7.2 Performanțe

Pentru analiza de performanță am început prin analiza impactului pe care îl are tipul de dată și dimensiunea matricei asupra timpului de rulare. În Tabelul 9 am prezentata rezultatele acestui test. Configurația aleasă este de 8 date accesibile în paralel. Testele au fost rulate pe matrice pătratice de dimensiuni egale.

În tabel sunt prezentate operația aritmetică executată, tipul de dată folosit și dimensiunea matricei pătratice. Rezultatele colectate sunt numărul de perioade de ceas pe care operația aritmetică a consumat-o pe un procesor RISC-V și numărul de perioade de ceas necesare pentru accelerator să execute aceeași operație, acest număr include și operațiile de configurare a acceleratorului dar și operațiile la memorie. Iar pe ultima coloană am calculat optimizarea, raportul între timpul pe accelerator și timpul pe procesor, acest rezultat este adimensional.

Scenariul în care date au fost colectate în Tabelul 9 este unul în care am considerat o memorie principală cu o latență foarte redusă. Acest scenariu este asemănător cu cazul în care avem un nivel doi de cache. În acest caz rezultatul de optimizare este maximul care se poate obține.

Acceleratorul are două unități aritmetice: una pentru înmulțirii de matrice și una pentru celelalte

Test	Tip de dată	Dim.	Accelera-tor	Procesor	Accelerare
Addition	int8_t	32	474	84765	178.83 ×
Subtraction	int8_t	32	470	84678	180.17 ×
Dot Product	int8_t	32	514	3556085	6918.45 ×
Cross Product	int8_t	32	463	85694	185.08 ×
Addition	int16_t	32	666	90841	136.40 ×
Subtraction	int16_t	32	655	91626	139.89 ×
Dot Product	int16_t	32	818	3885403	4749.88 ×
Cross Product	int16_t	32	662	92692	140.02 ×
Addition	int32_t	32	1050	71449	68.05 ×
Subtraction	int32_t	32	1046	73532	70.30 ×
Dot Product	int32_t	32	1450	2944367	2030.60 ×
Cross Product	int32_t	32	1039	74652	71.85 ×
Addition	int8_t	64	1434	337409	235.29 ×
Subtraction	int8_t	64	1430	337525	236.03 ×
Dot Product	int8_t	64	1977	28260326	14294.55 ×
Cross Product	int8_t	64	1430	341568	238.86 ×
Addition	int16_t	64	2206	364147	165.07 ×
Subtraction	int16_t	64	2200	365598	166.18 ×
Dot Product	int16_t	64	3757	31168262	8296.05 ×
Cross Product	int16_t	64	2200	369687	168.04 ×
Addition	int32_t	64	3746	291650	77.86 ×
Subtraction	int32_t	64	3742	292077	78.05 ×
Dot Product	int32_t	64	7346	25915214	3527.80 ×
Cross Product	int32_t	64	3740	296144	79.18 ×
Addition	int8_t	128	5273	1345403	255.15 ×
Subtraction	int8_t	128	5278	1345428	254.91 ×
Dot Product	int8_t	128	11457	245374248	21416.97 ×
Cross Product	int8_t	128	5271	1362040	258.40 ×
Addition	int16_t	128	8362	1457864	174.34 ×
Subtraction	int16_t	128	8356	1458075	174.49 ×
Dot Product	int16_t	128	22724	270321852	11895.87 ×
Cross Product	int16_t	128	8358	1474422	176.41 ×
Addition	int32_t	128	14524	1163889	80.14 ×
Subtraction	int32_t	128	14516	1163932	80.18 ×
Dot Product	int32_t	128	45260	210673885	4654.75 ×
Cross Product	int32_t	128	14518	1180163	81.29 ×

Tabel 9: Comparație a timpului de rulare pe multiple teste și tipuri de date. Numărul de date paralele este de 8.

operații matriciale. *Vectorial Unit* se ocupă de operațiile care se pot reduce la operații element-cu-element, precum: adunarea și scăderea. Din această cauză timpii de rulare în cazul testelor de tip *Addition*, *Subtraction* și *Cross Product* sunt asemănătoare. Operația aritmetică diferă dar calea de date și cea de control sunt aceleași.

Operațiile de înmulțire de matrice, *Dot Product*, sunt operate de *Matrix Unit*, această unitate care are un hardware dedicate pentru astfel de operații. Datorită acestui lucru această operație obține o optimizare atât de ridicată.

Tipul de dată are un impact asupra numărului de tranzacții la memoria principală. Crescând lățimea tipului de dată și păstrând constantă dimensiunea matricelor, dimensiunea în memoria a matricelor crește. Acest lucru are un impact directa supra tranzacțiilor la memorie, dar impactul nu este unul semnificativ. Pentru operațiile la memorie folosim avantajul oferit de *AXI Burst*, care ajută la transferul volumelor mari de date.

Intern, orice operație aritmetică depinde doar de numărul de elemente. Acesta este cauza pentru care timpul de rulare crește cu dimensiunea matricei. Pe lângă numărul mai mare de operații, vor fi necesare și mai multe date.

Test	**Tipul de dată**	**Cicli de ceas**				**Accelerare**		
		Procesor	**4 lanes**	**8 lanes**	**16 lanes**	**4 lanes**	**8 lanes**	**16 lanes**
Addition	int8_t	336811	1944	1434	1202	173.26 ×	234.88 ×	280.21 ×
Subtraction	int8_t	337378	1940	1430	1196	173.91 ×	235.93 ×	282.09 ×
Dot Product	int8_t	28260690	5036	1977	1282	5611.73 ×	14294.73 ×	22044.22 ×
Cross Product	int8_t	341515	1940	1430	1196	176.04 ×	238.82 ×	285.55 ×
Addition	int16_t	363983	2716	2206	1988	134.01 ×	165.00 ×	183.09 ×
Subtraction	int16_t	365529	2703	2200	1975	135.23 ×	166.15 ×	185.08 ×
Dot Product	int16_t	31165445	9896	3757	2292	3149.30 ×	8295.30 ×	13597.49 ×
Cross Product	int16_t	369669	2703	2200	1975	136.76 ×	168.03 ×	187.17 ×
Addition	int32_t	291646	4247	3746	3551	68.67 ×	77.86 ×	82.13 ×
Subtraction	int32_t	291935	4252	3742	3554	68.66 ×	78.02 ×	82.14 ×
Dot Product	int32_t	25913921	19624	7346	4360	1320.52 ×	3527.62 ×	5943.56 ×
Cross Product	int32_t	296161	4252	3740	3554	69.65 ×	79.19 ×	83.33 ×

Tabel 10: Impactul numărul de date simultane asupra timpului de rulare. Dimensiunea matricilor este de 64×64. Procesorul este un procesor RISC-V.

În Tabelul 10 am analizat impactul numărului de date disponibile în paralel asupra performanțelor. Scenariul de test este același cu cel din Tabelul 9. Creșterea numărului de date în paralel conduce la o mai mare optimizare, dar aceasta nu este liniară. Timpul are trei componente: configurarea, operațiile la memorie și operația aritmetica. Creșterea numărului de date în paralele va reduce doar timpul de execuție al operației aritmetice. Celelalte două componente rămânând constante.

8 Concluzii

În această lucrare am prezentat efortul depus pentru extinde setul de instrucțiuni RISC-V. Domeniul de activitate ales de noi a fost optimizări pentru operațiile matriciale.

Lucrarea a urmărit un proces de cercetare-dezvoltare. Am început prin definirea specificațiilor funcționale și nefuncționale. În capitolele următoare am expus soluțiile alese pentru a descrie noile instrucțiuni și implementarea lor în compilator. Pentru folosirea instrucțiunilor am implementat un accelerator, care pentru a livra performanțe ridicate, are soluții hardware dedicate matricelor. La finalul lucrării am definit un mediu de test pe baza căruia am colectat multiple metrici și am analizat și interpretat aceste rezultate.

Proiectul a presupus definirea unei extensii a setului de instrucțiuni RISC-V pentru operații matriciale. Acceleratorul care execută instrucțiunile a fost implementa la nivel RTL în SystemVerilog. Pentru conectarea la procesor am ales o interfață de extensie definită pentru procesoarele RISC-V. Interfața aleasă, CV-X-IF, permite conectarea la mai multe procesoare RISC-V, fără a fi nevoie de modificări ulterioare.

Acceleratorul dispune de o memorie internă bidimensională. Am ales acest tip de memorie care este optimizat pentru matrice. Structura ei ne permite să accesăm date în paralel, elementele returnate sunt ori de pe aceeași linie ori aceeași coloană. Numărul de date în paralel este un parametru al acestui accelerator.

Instrucțiunile definite sunt de tip SIMD, o instrucțiune care operează cu mai multor date simultan. Instrucțiunile sunt de trei tipuri: definesc matricele în structura internă a acceleratorului, definesc operații cu memoria principala și definesc operații aritmetice cu matrici.

Un alt avantaj este definirea în mod dinamic a matricelor în accelerator. Pentru a definii o matrice sunt necesare: lungimea și lățimea (în elemente), tipul de dată și locația în memoria internă. Acceleratorul operează cu date numere întregi, cu și fără semn, pe 8, 16 sau 32 de biți.

Operațiile cu memoria principală, au ca scop aducerea operanzilor din memoria RAM și scrierea rezultatelor înapoi. Deoarece avem dimensiunile, în număr de elemente, și tipul de dată, dimensiunea în byți este calculată. Acceleratorul a fost gândit să lucreze cu C, astfel consideră că matricele sunt de tip *Row Major*, liniile sunt una după alta în memorie. Pentru a avea accese eficiente, am folosit avantajele oferite de *AXI Burst*, această metodă permite transferul în rafală a unui volum de date de 4KB.

Operațiile de matrice suportate sunt de două categorii: operații între matrice și operații de scalare a unei matrice. Operațiile matriciale suportate sunt: adunarea, scăderea, produsul vectorial și pro-

dusul scalar. Operațiile de scalare au fost gândite pentru a aplica transformări tuturor elementelor din acea matrice. Operațiile de scalare suportate sunt: adunarea, scăderea și înmulțirea. Al doilea operator este un număr, care va fi aplicat la toate elementele împreună cu operația dorită.

Pe partea de implementare hardware, acceleratorul dispune de circuite dedicate pentru fiecare tip de operație matriceala. De aceea există două tipuri de unități aritmetice. Cea mai importantă este cea pentru înmulțirea de matrice care dispune de o matrice sistolică, o topologie hardware dedicată acestei operații. Pentru restul de operații există o unitate care tratează matricele ca și vectori, acest tip de unitate se potrivește pentru operațiile de adunare și scădere. Fiecare tip de unitate aritmetică depinde direct proporțional de numărul de date disponibile în paralel, deoarece dorim să procesăm simultan toate datele pe care le putem accesa.

Acceleratorul rezolvă două tipuri de hazard în procesor. Primul hazard pe care îl rezolvă este hazardul de control. Instrucțiunile definite fiind de tip SIMD, se reduc instrucțiunile de control, în cazul nostru sunt eliminate pentru operații, fiind integrate în implementarea acceleratorului. Reducerea acestui tip de hazard îmbunătățește tipul de rulare, în cazul implementării noastre structurile repetitive, precum buclele, sunt executate automat la fiecare perioadă de ceas. Un alt hazard eliminat este cel structural, memoria internă a fost configurată să dispună de două porturi de citire și un port de scriere. A permis cai de date simplificate, nu am folosit alte memorii tampon sau sincronizatoare între operanzi. În acest fel la fiecare perioadă de ceas ambii operanzi sunt aduși din memorie și rezultatul este scris simultan.

Legat de performanțe, acceleratorul livrează optimizări ridicate. Acceleratorul este de cel puțin 68$\times$ mai rapid decât un procesor RISC-V. În măsurarea timpul pe accelerator am considerat și complexitatea adăugată - definirea matricelor și operațiile explicite cu memoria principală. Procesorul dispune și de un cache, care pentru o comparație justă avea datele în memorie. Operațiile de înmulțire sunt de peste 2000$\times$ mai rapide pe accelerator, datorită matricei sistolice.

Dacă creștem numărul de date accesibile în paralel crește eficiența acceleratorului. Timpul care se reduce este timpul operației aritmetice, deoarece numărul de date influențează unitățile aritmetice. Operațiile de definire și cele la memorie vor consuma același timp.

Analizând implementările pe FPGA, acceleratorul poate obține frecvențe de peste 190MHz. Dar dacă numărul de date în paralel trece de peste 32, frecvența maximă scade drastic, deoarece calea de control pentru matricea sistolică devine cale critică. Analizând resursele, ele cresc direct proporțional cu numărul de date în paralel. Arhitectură de tip *pipeline* depinde de volumul de date, și cu creșterea numărului de date și numărul de LUT-uri și FF-uri. Resursele cresc în calea de date, calea de control depinde doar de dimensiunea memoriei interne, pe care nu am modificat-o între teste.

ACKNOWLEDGEMENTS

This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNC-S/CCCDI - UEFISCDI, project number PN-IV-P8-8.1-PME-2024-0022, within PNCDI IV.

The ISOLDE project, nr. 101112274 is supported by the Chips Joint Undertaking and its members Austria, Czechia, France, Germany, Italy, Romania, Spain, Sweden, Switzerland.

Bibliografie

[1] "ISOLDE Project," Edacentrum. (Jun. 2025), [Online]. Available: https://www.isolde-project.eu/ (visited on 06/10/2025).

[2] "Bine ați venit pe site-ul proiectului ISOLDE!" IMT. (Jun. 2025), [Online]. Available: https://www.imt.ro/isolde_PNIV/ (visited on 06/10/2025).

[3] C. B. Ciobanu, H. Gâlmeanu, A. Puşcaşu, *et al.*, "RISC-V Accelerators, Enablement and Applications for Automotive and Smart Home in the ISOLDE Project," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Springer, 2024, pp. 215–230.

[4] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[5] "System architecture | Cloud TPU | Google Cloud," Google. (), [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm (visited on 01/15/2025).

[6] T. Moreau, T. Chen, L. Vega, *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[7] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, "RedMule: A mixed-precision matrix?matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration," *Future Generation Computer Systems*, vol. 149, pp. 122–135, 2023, **ISSN**: 0167-739X. **DOI**: https://doi.org/10.1016/j.future.2023.07.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002546.

[8] H. Kim, G. Ye, N. Wang, A. Yazdanbakhsh, and N. S. Kim, " Exploiting Intel Advanced Matrix Extensions (AMX) for Large Language Model Inference," *IEEE Computer Architecture Letters*, vol. 23, no. 01, pp. 117–120, Jan. 2024, **ISSN**: 1556-6064. **DOI**: 10.1109/LCA.2024.3397747. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/LCA.2024.3397747.

[9] "Part 1: Arm Scalable Matrix Extension (SME) Introduction," Zenon Xiu. (May 2024), [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction (visited on 01/15/2025).

[10] "The Standalone AI Computer and its Programming Model," Tensorrent. (Aug. 2024), [Online]. Available: https://hc2024.hotchips.org/assets/program/conference/day1/88_HC2024.Tenstorrent.Jasmina.Davor.v7.pdf (visited on 01/15/2025).

[11] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V edition: The hardware software interface*, Second. The Morgan Kaufmann, 2021, **ISBN**: 978-0-12-820331-6.

[12] *About RISC-V International*, https://riscv.org/about/, Accesat: 14.03.2025.

[13] *What is RISC-V?* https://www.synopsys.com/glossary/what-is-risc-v.html, Accesat: 14.03.2025.

[14] C. B. Ciobanu, G. Stramondo, C. de Laat, and A. L. Varbanescu, "MAX-PolyMem: high-bandwidth polymorphic parallel memories for DFEs," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2018, pp. 107–114.

[15] C. B. Ciobanu, *Customizable Register Files for Multidimensional SIMD Architectures*. Delft University of Technology, Mar. 8, 2013, **ISBN**: 978-94-6186-121-4. [Online]. Available: https://resolver.tudelft.nl/uuid:6da2ee07-99df-450d-93bd-2367725f4f70.

[16] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Multimedia rectangularly addressable memory," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 315–322, 2006.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Sixth. Morgan Kaufmann, 2019, **ISBN**: 978-0-12-811905-1.

[18] A. Barredo, J. M. Cebrián, M. Moretó, M. Casas, and M. Valero, "POSTER: An Optimized Predication Execution for SIMD Extensions," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2019, pp. 479–480.

[19] D. Vuță-Popescu, I. C. Antofi, C. B. Ciobanu, and C. Z. Kertész, "SIMD Extensions-A Historical Perspective," in *2024 IEEE 30th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, IEEE, 2024, pp. 108–115.

[20] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The ARM scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[21] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[22] T. Edamatsu and D. Takahashi, "Acceleration of large integer multiplication with Intel AVX-512 instructions," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, 2018, pp. 211–218.

[23] "GNU toolchain for RISC-V, including GCC," RISC-V collab. (May 2024), [Online]. Available: `https://github.com/riscv-collab/riscv-gnu-toolchain` (visited on 04/15/2024).

[24] "RISC-V Opcodes," RISC-V collab. (Nov. 2025), [Online]. Available: `https://github.com/riscv/riscv-opcodes` (visited on 01/15/2024).

[25] "Matrix accelerator RISC-V Opcodes," Alexandru Puscasu. (Nov. 2025), [Online]. Available: `https://github.com/alex2kameboss/MA-riscv-opcodes` (visited on 01/15/2025).

[26] "Matrix Accelerator GNU toolchain for RISC-V, including GCC," Alexandru Puscasu. (Feb. 2025), [Online]. Available: `https://github.com/alex2kameboss/MA-riscv-gnu-toolchain` (visited on 01/15/2025).

[27] "CVA6," OpenHW Group. (May 2025), [Online]. Available: `https://github.com/openhwgroup/cva6.git` (visited on 04/01/2025).

[28] "VCU128 Evaluation Board User Guide (UG1302)," AMD. (May 2022), [Online]. Available: `https://docs.amd.com/r/en-US/ug1302-vcu128-eval-bd` (visited on 04/15/2025).

[29] "Versal ACAP Memory Resources Architecture Manual (AM007)l," AMD. (Nov. 2024), [Online]. Available: `https://docs.amd.com/r/en-US/am007-versal-memory/Block-RAM-and-UltraRAM-Differences` (visited on 01/15/2025).

REZUMAT

Această lucrare analizează extinderea setului de instrucțiuni RISC-V pentru operații matriciale. Pentru a testa instrucțiunile propuse, a fost implementat un accelerator conectat direct la un procesor RISC-V, folosind o interfață standard de extensie, CV-X-IF. Implementarea acceleratorului a fost realizată în SystmVerilog. Acceleratorul are o memorie bidimensională, care ne permite accesul datelor într-un format matricial, implementarea folosită permite accesarea mai multor date de pe aceeași linie sau aceeași coloană. Suplimentar acceleratorul oferă regiștrii matriciali definiți software, acest lucru oferă o flexibilitate ridicată programatorului. Acceleratorul folosește topologii hardware optimizate pentru operațiile matriciale.

Operațiile definite sunt de tip SIMD, acest lucru reduce dimensiunea codului sursă și numărul de instrucțiuni executate, îmbunătățind timpi de rulare. Instrucțiunile adăugate permit definirea de regiștrii matriciali, operații cu memoria principală și operații matriciale. Operațiile suportate sunt adunarea, scăderea, produsul vectorial și scalar. Suplimentar există și operații care permit scalarea matricelor, operații care se aplică fiecărui element din matrice. Aceste instrucțiuni au fost adăugate și în compilatorul C pentru RISC-V, suportul fiind la nivel de asamblor.

Pentru a testa soluția propusă am rulat două tipuri de teste, în primul am analizat optimizarea față de un procesor și implementarea pe un FPGA. Pentru aceste teste a fost realizat un sistem de testare complet funcțional care include: un procesor RISC-V, acceleratorul, o magistrală și o memorie principală. Pentru testele de performanță a fost realizat și un program în C care măsoară timpi de rulare a diverselor operații matriciale pe procesor și accelerator. Optimizarea obținută este de peste 68$\times$ față de operațiile pe procesor. Pentru înmulțirea de matrice am implementat o matrice sistolică, o arhitectură hardware optimizată pentru înmulțiri de matrice, obținând o optimizare de peste 2000$\times$. Pe partea e FPGA am obținut frecvențe peste 190MHZ pentru acceleratorul implementat.

ABSTRACT

This thesis focuses on extending the RISC-V ISA for matrix operations. For the proposed extension, we implemented a tightly-coupled accelerator, by using a standard extension interface called CV-X-IF. The accelerator is implemented at RTL in SystemVerilog. The accelerator incorporates a bidimensional memory, which is optimized for matrices accesses, the implemented memory allows access of multiple data in parallel. The bidimensional memory allows us to access multiple data on same row or same column. An important feature is software defined matrix registers; this improves flexibility for programmers. For matrix operations, we will use optimized hardware topology.

The defined instructions are on SIMD type. This approach reduces code size and the number of committed instructions, which improve runtime. The new instructions allow defining a matrix, memory operations and matrix operations. Supported matrix operations are addition, subtraction, cross and dot product. Moreover, the accelerator supports matrix-scalar operations, these operations allow scale to scale one matrix. The proposed intrusions have compiler support on RISC-V C toolchain. The compiler support is at assembler level.

To evaluate the proposed solution, we analyzed two kinds of metrics: we measured optimization against a simple RISC-V core and FPGA metrics. For those tests we implemented a test environment which included: a RISC-V core, the accelerator, a crossbar and RAM. For the performance tests, we implemented a C benchmark which measures runtime for various matrix operations on both accelerator and RISC-V core. The accelerator is more than 68$\times$ times faster. For matrix multiplication we implemented a systolic array, enhanced for GEMM, and reach more than 2000$\times$ improvement in runtime. On the FPGA side, we could achieve more than 190MHz for our accelerator.