Universitatea
Transilvania
din Brașov

FACULTATEA DE INGINERIE ELECTRICĂ
ȘI ȘTIINȚA CALCULATOARELOR

PROIECT DE DIPLOMĂ

Conducător științific:
Conf. Dr. Ing. CIOBANU Cătălin
Colaborare cu NXP Semiconductors România:
Dr. Ing. COSTINESCU Simona

Absolvent:
POPESCU Vlad-Mihai

Brașov, 2025

Departamentul de Electronică și Calculatoare
Programul de studii: Calculatoare

POPESCU Vlad-Mihai

Extinderea compilatorului LLVM pentru arhitectura RISC-V

Conducător științific:
Conf. Dr. Ing. CIOBANU Cătălin
Colaborare cu NXP Semiconductors România:
Dr. Ing. COSTINESCU Simona

Brașov, 2025

Cuprins

Lista de figuri, tabele și coduri sursă

FIGURI

TABELE

CODURI SURSĂ

ASan – Address Sanitizer;
AST – Abstract Syntax Tree;
BB – Basic Block;
BSOD – Blue Screen of Death;
CFI – Control-Flow Integrity;
Clang – C Language Family Front-end for LLVM;
CMake – Cross-Platform Make;
CPU – Central Processing Unit;
DAG – Directed Acyclic Graph;
DFG – Data Flow Graph;
eMMC – Embedded MultiMediaCard;
ETH – Ethernet;
FPGA – Field-Programmable Gate Array;
G++ – GNU C++ Compiler;
GAS – GNU Assembler;
GCC – GNU Compiler Collection;
GDB – GNU Debugger;
GNU – GNU's Not Unix;
GPIO – General-Purpose Input/Output;
GPU – Graphics Processing Unit;
GVN – Global Value Numbering;
HWASan – Hardware-Assisted Address Sanitizer;
I2C – Inter-Integrated Circuit;
IDE – Integrated Development Environment;
IPO – Interprocedural Optimization;
IR – Intermediate Representation;
ISA – Instruction Set Architecture;
ISel – Instruction Selection;
ISP – In-System Programmer;
IoT – Internet of Things;
JSON – JavaScript Object Notation;
L1/L2 – Level 1 / Level 2 (Cache);
LD – Linker (Generic);
LLD – LLVM Linker;
LLVM – Low-Level Virtual Machine;
LPDDR4 – Low-Power Double Data Rate 4;
LTO – Link-Time Optimization;

MC – Machine Code (LLVM MC layer);
MLIR – Multi-Level Intermediate Representation;
MSBuild – Microsoft Build Engine;
O0-O3 – Optimization Levels 0 through 3;
OS – Operating System;
PBQP – Partitioned Boolean Quadratic Programming;
PGO – Profile-Guided Optimization;
PK – Proxy Kernel;
RA – Register Allocation;
RISC-V – Reduced Instruction Set Computer - V;
RV32I – 32-bit Base Integer RISC-V ISA;
RV64GC – 64-bit RISC-V ISA with "G" Standard Extensions and Compressed Instructions;
SBC – Single-Board Computer;
SCEV – Scalar Evolution Analysis (LLVM);
SFTP – Secure File Transfer Protocol;
SIMD – Single Instruction, Multiple Data;
SPI – Serial Peripheral Interface;
SSA – Static Single Assignment;
SSH – Secure Shell;
SoC – System on Chip;
STDOUT – Standard Output;
SW – Software;
TableGen – LLVM TableGen Domain-Specific Language;
TSan – Thread Sanitizer;
UART – Universal Asynchronous Receiver/Transmitter;
UBSan – Undefined Behavior Sanitizer;
USB – Universal Serial Bus;
VExt – Vector Extension (RISC-V);
WSL2 – Windows Subsystem for Linux 2;

1 Introducere

1.1 Motivație și context global

Dacă în anii '90 piața procesoarelor era dominată de câteva companii care controlau integral drepturile de proprietate intelectuală, deceniul curent a demonstrat că progresul tehnologic poate fi accelerat considerabil atunci când barierele legale și financiare sunt eliminate. Arhitectura **RISC-V**, născută la Universitatea California, Berkeley, în 2010, reprezintă materializarea acestui nou model: specificația este publică, revizuită în mod transparent, iar licența permite oricui să proiecteze, să fabrice și să vândă cipuri fără redevențe [1]. Deschiderea ISA-ului a creat un „efect de rețea" în cercetare, unde rezultate publicate sub formă de patch-uri upstream se propagă în timp real către mii de dezvoltatori, scurtând drastic ciclul dintre idee și produs. Faptul că astăzi există peste cincizeci de nuclee RISC-V gata de siliciu, de la microcontrolere de câțiva kilobiți la implementări out-of-order pentru servere ilustrează potențialul unei arhitecturi care separă dreptul de experimentare de obligația de a plăti licențierea [2].

Această emergență nu ar fi fost posibilă fără un ecosistem software pe măsură. Proiectul **LLVM**, pornit inițial ca un compilator de curs universitar, a evoluat într-o platformă industrială de analiză și generare de cod, folosită astăzi de Apple, Microsoft, Google, Qualcomm și de marea majoritate a distribuțiilor Linux. Arhitectura sa modulară un IR bine definit, pași de optimizare reutilizabili și backend-uri încărcate dinamic permite adăugarea rapidă a noi ținte: în doar doi ani de la primele *commits*, backend-ul RISC-V depășea 95 ajungea la paritate de performanță cu GCC pe suite benchmark moderne [3]. Sinergia LLVM–RISC-V este completată de kernel-ul **Linux**, care furnizează un strat omogen pentru profilare, depanare și execuție, indiferent că hardware-ul rulează într-un cluster HPC sau într-un senzor IoT.

1.2 Context european: suveranitate digitală și green HPC

Europa a recunoscut explicit, prin *European Chips Act*[4], că viitorul economic și strategic depinde de capacitatea de a proiecta și fabrica semiconductoare pe teritoriul Uniunii. Documentul îndeamnă statele membre să evite dependențele critice față de ISA-uri proprietare, subliniind că un lanț de aprovizionare transparent minimizează riscurile de securitate și permite un control mai mare asupra consumului energetic. În acest cadru, *European Processor Initiative* (EPI) a ales RISC-V ca arhitectură de referință pentru nucleele cu eficiență energetică ridicată, destinate viitoarei generații de supercalculatoare „verzi" [5]. Spre deosebire de abordările tradiționale, unde optimizările sunt impuse de

furnizorul hardware, colaborarea pan-europeană încurajează extensii ISA dezvoltate pentru nevoi locale: acceleratoare pentru criptografie post-cuantică folosite de agenții guvernamentale, instrucțiuni pentru procesarea masivă a datelor de satelit în cadrul misiunilor ESA sau unități dedicate simulărilor climatice de înaltă rezoluție.

1.3 Importanța operației dot product scalar

În centrul multor algoritmi numerici se află produsul scalar, $\mathrm{DOT}(a, b) = \sum_{i=0}^{n-1} a_i b_i$. Nivelul 1 din BLAS, biblioteca canonică pentru calcul algebric, descrie această operație ca pe o primitivă de bază; performanța BLAS-1 influențează direct eficiența nivelurilor superioare (mulțimi matrice-vector și matrice-matrice), iar acestea, la rândul lor, stau la baza bibliotecilor de rețele neuronale și a aplicațiilor de fizică computațională [6]. Arhitecturile x86 și Arm au introdus instrucțiuni dedicate (`VPDPDQ`, `UDOT`) tocmai pentru a compensa costul ridicat al accesului la memorie și al dependențelor care limitează paralelismul [7]. Standardul RISC-V, aflat într-un proces continuu de extindere, include versiuni vectoriale ale acestor operații, însă marea majoritate a implementărilor comerciale în special cele destinate IoT și aplicațiilor low-power folosesc nuclee pur scalare din rațiuni de cost. În absența unei instrucțiuni dedicate, compilatorul generează o buclă fixă: încărcare, multiplicare, acumulare. În practică, performanța este limitată de latența multiplicării pe 64 de biți și de bandwidth-ul memoriei, ceea ce duce la timpi de execuție de ordinul sutelor de cicluri per element într-o implementare neoptimizată.

1.4 Provocarea abordată în această lucrare

Lucrarea de față demonstrează că simpla adăugare a unui *builtin* și a infrastructurii interne care îl deservește, poate reduce dramatic acest handicap. Apelul `__builtin_riscv_dot` este recunoscut în front-end, mapat la intrinsecul LLVM `llvm.riscv.dot` și apoi tradus într-o pseudo-instrucțiune `DOT`. Pentru hardware-ul actual, pasul *RISCVExpandPseudo* convertește `DOT` într-o buclă scalară minimă, care rulează post-RA și utilizează registre fizice fixe pentru a evita alocări suplimentare. În momentul în care furnizorii RISC-V vor propune o instrucțiune reală de produs scalar, același intrinsec va fi legat direct de codul mașină, fără nicio modificare a aplicațiilor existente. În acest sens, *builtin-ul* funcționează ca o abstracție stabilă peste o evoluție hardware inevitabilă.

1.5 Structura lucrării

Pentru a asigura o înțelegere logică și progresivă a temei abordate, lucrarea este organizată în șapte capitole, fiecare având un rol bine definit în construcția demersului științific:

A. **Introducere**. Primul capitol oferă contextul general al cercetării, evidențiind importanța optimizării operațiilor matematice fundamentale în arhitecturi moderne RISC-V, precum și motivația implementării unei instrucțiuni personalizate în backend-ul LLVM. Sunt prezentate obiectivele urmărite și metodologia generală adoptată.

B. **LLVM în profunzime**. Capitolul 2 detaliază arhitectura LLVM, pornind de la reprezentarea intermediară (IR) și modul de generare a codului, până la etapele de optimizare, legalizare și selecție a instrucțiunilor. Sunt discutate componentele relevante din Clang și backend-ul LLVM, fiind evidențiat fluxul de transformare a unui cod sursă care conține un `builtin` în instrucțiuni specifice arhitecturii țintă. Diagramele incluse clarifică succesiunea pașilor implicați.

C. **Platforma hardware**. Capitolul 3 descrie specificațiile tehnice ale plăcii *BeagleV-Fire*, utilizată pentru testarea experimentală. Sunt prezentate caracteristicile procesorului quad-core RISC-V, ierarhia de cache, memoria LPDDR4 și capacitățile de monitorizare a performanței. Această secțiune oferă o bază solidă pentru interpretarea corectă a rezultatelor de benchmark.

D. **Setup experimental**. Capitolul 4 documentează mediul de lucru și instrumentele utilizate: compilatorul LLVM 17, biblioteca standard `glibc` 2.40, nucleul Linux 6.8, precum și metodele de măsurare a performanței folosind timpi medii pe apel. Este descrisă procedura de compilare și testare, incluzând provocările întâmpinate în etapa de cross-compiling și debugging pe platforma țintă.

E. **Implementarea instrucțiunii `__builtin_riscv_dot`**. Capitolul 5 reprezintă secțiunea centrală a lucrării, prezentând în detaliu modificările aduse în backend-ul LLVM pentru a integra o nouă instrucțiune pseudo numită `DOT`. Sunt explicate modificările realizate în fișierele `IntrinsicsRISCV.td`, `RISCVInstrInfo.td`, `RISCVISelDAGToDAG.cpp`, `RISCVExpandPseudoInsts.cpp` și `RISCVTargetMachine.cpp`, precum și strategia utilizată pentru a optimiza emiterea codului nativ. De asemenea, este inclus un grafic schematic care urmărește fluxul complet de transformare de la codul sursă până la codul de asamblare final.

F. **Evaluarea performanței**. Capitolul 6 compară performanța noii instrucțiuni `DOT` cu implementările clasice în C ale produsului scalar, pentru diverse dimensiuni ale vectorilor și niveluri de

optimizare. Sunt prezentate rezultate sub formă de tabele și grafice, evidențiind câștigurile de performanță semnificative înregistrate de versiunea `builtin` față de implementarea nativă.

G. **Dificultăți întâmpinate și soluții aplicate**. Capitolul 7 descrie problemele tehnice majore întâlnite pe parcursul lucrării, precum provocările legate de cross-compiling între Windows și Linux, blocarea hardware a plăcii BeagleV-Fire, precum și dificultățile de integrare a pseudo-instrucțiunii `DOT` în fluxul de selecție al LLVM. Sunt explicate soluțiile concrete adoptate, precum trecerea la dual-boot Linux Mint, utilizarea unei conexiuni UART improvizate cu Arduino și intervențiile precise în codul sursă al compilatorului pentru a asigura corecta generare a instrucțiunii personalizate.

H. **Concluzii**. Capitolul final sintetizează contribuțiile lucrării, subliniind faptul că integrarea unei instrucțiuni personalizate în infrastructura LLVM presupune o înțelegere profundă a tuturor nivelurilor implicate: frontend, IR, SelectionDAG și backend. Se evidențiază reușita funcțională și performanțele obținute, confirmând validitatea abordării propuse.

Prin această traiectorie, textul pune în evidență nu doar valoarea practică a unui *builtin* izolat, ci și metodologia generală de extindere a unui compilator de producție pentru a răspunde rapid la evoluțiile hardware, aliniindu-se totodată obiectivelor strategice de autonomie tehnologică și eficiență energetică promovate la nivel european.

2 Lanțul de compilare *Clang & LLVM* pentru RISC -V

2.1 Repere istorice

Povestea **LLVM** începe în anul 2000, când Chris Lattner dezvoltă, ca proiect de licență, un compilator capabil să păstreze informații despre program pe toată durata execuției. În 2003, împreună cu profesorul Vikram Adve, publică articolul fondator „*LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*" [8], definind un *Intermediate Representation* static–single–assignment destinat atât optimizărilor offline, cât şi execuției *just-in-time*. Nucleul devine open-source sub licență BSD în 2005, iar primele backend-uri x86 şi PowerPC sunt upstream-ate ca lucrări studențești la Universitatea Illinois.

Punctul de inflexiune survine în 2007, când Apple finanțează front-end-ul *Clang*, un parser C/C++/Objective-C capabil de compilare incrementală şi mesaje de eroare structurate. În intervalul 2008–2012 Clang devine compilator implicit în Xcode, este adoptat de FreeBSD şi Google Chrome, iar LLVM intră în infrastructura de build a proiectelor Android [9]. După 2015 ecosistemul se diversifică: AMD upstream-ează backend-ul GPU [10], Microsoft integrează linker-ul *LLD* în Windows [11], iar Google lansează sistemul de operare *Fuchsia*, construit exclusiv cu Clang [12]. Suportul oficial pentru arhitectura **RISC-V** pornește în 2017 [13], extensia vectorială RVV este upstream-ată în 2020 [14], iar Clang 17 (2024) atinge paritate de performanță cu GCC-13 pe SPEC-INT 2017 [15].

2.2 Principii de proiectare LLVM

2.2.1 IR unic, textual, pe toată viața programului

Fișierele `.ll` și `.bc` pot fi inspectate cu un editor obișnuit, pot fi transmise prin rețea pentru compilare distribuită și pot fi re-optimizate la rulare (just-in-time). Când adăugăm o instrucțiune nouă în backend nu atingem front-end-ul sau middle-end-ul, pentru că structura de date rămâne aceeași.

2.2.2 Modularitate extremă

Front-end-urile (Clang, Rustc, Swift), middle-end-ul (peste 200 de *passes*) și backend-urile (x86, Arm, RISC-V, etc.) sunt biblioteci încărcate dinamic. Fiecare optimizare sau selector de instrucțiuni poate fi activat ori dezactivat cu un simplu „`-mypass-pipeline=`" fără recompilarea altor părți. Pentru cercetare e vital să testăm un nou algoritm de alocare a registrelor doar cu două fișiere CMake modificate, în loc să parcurgem un monolit de milioane de linii.

2.2.3 Licență Apache 2.0 și comunitate colaborativă

Proiectul LLVM este distribuit sub licența **Apache License 2.0**, o licență permisivă care permite reutilizarea, modificarea și redistribuirea codului sursă, atât în scopuri comerciale, cât și academice, fără a impune restricții de tip copyleft, precum cele întâlnite în GPL. Această flexibilitate este esențială pentru colaborarea deschisă între industrie și mediul academic, permițând integrarea rapidă a contribuțiilor din ambele direcții.

În mod concret, contribuțiile din partea universitară (precum extensii experimentale pentru arhitectura RISC-V) pot fi preluate de companii comerciale fără fricțiuni legale, favorizând un ciclu de dezvoltare rapid patch–review–upstream. Ecosistemul RISC-V se bucură astfel de un proces de upstreaming eficient: extensiile propuse sunt frecvent revizuite și aprobate în una-două iterații, deoarece reviewerii fac parte activ din comunitatea care utilizează zilnic aceste instrumente. Colaborarea strânsă dintre dezvoltatorii industriali și cei academici contribuie la menținerea unui standard ridicat de calitate și compatibilitate în întreg lanțul de compilare LLVM.

2.2.4 Toolchain unificat

Linker-ul `lld`, asamblorul `llvm-mc`, profilerul `llvm-profdata` și instrumentele de analiză (`clang-tidy`, `scan-build`) împart aceleași structuri IR, ceea ce elimină conversiile intermediare. Pentru RISC-V înseamnă că un patch de instrucțiune nouă devine vizibil simultan în dezasmblarea `llvm-objdump`, în `perf annotate` și în diagnosticele Clang.

Prin separarea clară între istoria evolutivă și principiile de proiectare, devine evident de ce extensiile dedicate arhitecturii RISC-V, precum pseudo-instrucțiunea `DOT` tratată ulterior, se pot integra rapid și coerent în infrastructura LLVM.

2.3 Fluxul complet de compilare

Figura 1 desenează traseul standard de la fișierul sursă la executabil. Mai jos, fiecare etapă este descrisă pe larg tocmai pentru a evidenția unde pot interveni extinderi specifice RISC-V.

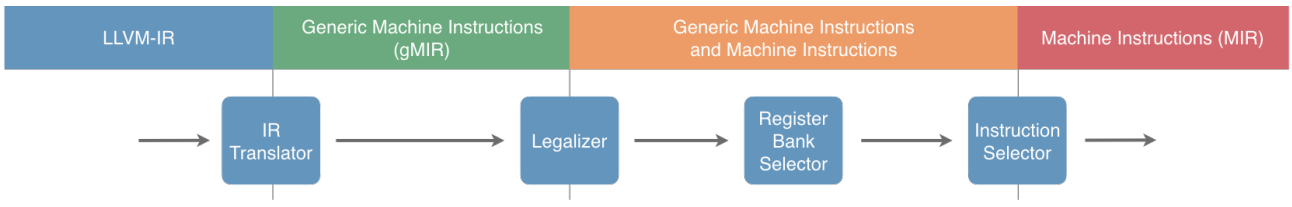

Figura 1: Lanțul Clang & LLVM — etape principale (după documentația oficială [16]).

2.3.1 I. Preprocesare, lexing și construirea AST-ului

Compilarea începe cu transpunerea fișierului sursă într-un flux de *tokens*. Preprocesorul Clang desfăşoară macro-urile, rezolvă directivele `#include` şi evaluează condiționalele `#ifdef/#ifndef`. Din punctul de vedere al arhitecturii RISC-V, nu există încă niciun element specific: textul este tratat ca C sau C++ pur. În etapa imediat următoare, parserul analizează token-urile şi construieşte arborele de sintaxă abstractă (AST). Fiecare nod, fie el o declarație de variabilă, un operator binar sau un apel de funcție, este bogat adnotat cu tipuri concrete, valori implicite, reguli de conversie şi atribute de vizibilitate. Prin activarea opțiunii `-ast-dump` arborele se poate inspecta textual, ceea ce facilitează depanarea *builtins*. Pentru extensiile ISA RISC-V, această fază rămâne complet transparentă: un apel la `__builtin_riscv_dot` se comportă ca un cuvânt cheie recunoscut de Clang, identic pe orice platformă.

2.3.2 II. Generarea LLVM IR

Traversând AST-ul, Clang emite instrucțiuni în *LLVM IR*, o formă SSA proiectată să fie independentă de dimensiunea cuvântului maşinii şi de setul de registre. Fiecare valoare are un tip explicit (`i32`, `double`, struct, vector ş. a. m. d.), astfel încât, optimizările să poată raționa fără ambiguități despre alias-uri şi semantica aritmetică. Direct în această fază pot apărea instrucțiuni pseudo-LLVM (*intrinsece*), de pildă `llvm.riscv.dot`, care descriu operații pe care hardware-ul le poate sau nu implementa nativ. În absența unei instrucțiuni reale, intrinsec-ul va fi tratat mai târziu de backend, fie prin substituire cu o secvență de cod, fie prin expandare software.

2.3.3 III. Optimizări în middle-end

Odată IR-ul generat, peste două sute de treceri independente îl rescriu, fiecare cu un scop bine delimitat. Transformările scalare (*InstCombine*, *GVN*, *SCCP*) elimină calcule redundante şi propagă constante. Analizele de buclă grupează iterațiile, identifică dependențe şi activează vectorizarea atunci când încărcările consecutive nu se suprapun; pe RISC-V aceste informații sunt esențiale, însă chiar şi pentru cod scalar, optimizările de desfăşurare (*unrolling*) reduc numărul de salturi şi îmbunătățesc utilizarea pipeline-ului. În final, optimizările inter-procedurale (*Inline*, *ThinLTO*, *PGO*) permit ca funcții mici să fie inserate direct la apel şi ca profilul de execuție să ghideze ordonarea codului pentru a maximiza coerența cache-ului.

2.3.4 IV. Selectarea instrucțiunilor

După ce IR-ul nu mai poate fi simplificat, acesta trebuie transpus în instrucțiuni concrete. LLVM oferă două mecanisme. *SelectionDAG* construieşte un graf date-control şi îl acoperă cu pattern-uri descrise declarativ în TableGen. *GlobalISel*, introdus ulterior, operează pe `Machine IR` având tipuri ataşate operandului şi se dovedeşte mai flexibil pentru tipuri exotice (vectori cu lungime variabilă, predicații). Intrinsec-ul `llvm.riscv.dot` este recunoscut aici şi înlocuit cu pseudo-instrucțiunea `DOT`; dacă un viitor nucleu va oferi o instrucțiune hardware, aceeaşi regulă TableGen va putea direcționa intrinsec-ul către op-code-ul real, fără a schimba codul surse.

2.3.5 V. Planificare și alocare de registre

Fluxul de instrucțiuni abstracte este apoi ordonat temporar de `MachineScheduler`. Obiectivul este să ascundă latențele operațiilor costisitoare (înmulțire pe 64 de biți, încărcări din memorie) intercalând instrucțiuni independente. Urmează alocarea de registre, unde algoritmul *Greedy* calculează intervalele de viață (*live ranges*) şi încearcă să le plaseze în registre fizice X1–X31 fără conflicte; când presiunea depăşeşte capacitatea, unele valori sunt „vărsate" pe stivă (*spilling*). Optimizarea *Shrink-wrap* mută prologul şi epilogul cât mai aproape de ramurile care chiar au nevoie de ele, reducând economiile inutile. Pe RISC-V, existența instrucțiunilor comprimate (extensia C) influențează decizia: registrele cu număr mic (X8–X15) pot selecta formate de 16 biți, micşorând binarul şi sporind densitatea de cod.

2.3.6 VI. Emiterea codului și asamblarea

În etapa finală a backend-ului, `MC Layer` transformă `MachineInstr`-urile în cuvinte binare, generează tabelele de relocare şi, dacă este necesar, aplică „*relaxation*" pentru instrucțiuni de salt al căror offset depăşeşte câmpul de 12 biți. Pentru imagini care pot fi încărcate la adresă necunoscută (*firmware*, kernel modul) se adoptă modelul de memorie `-mcmodel=medany`, iar adresele absolute sunt înlocuite cu secvențe `auipc+ld/st` independente de locația finală. Fişierele obiect rezultate (`.o`) pot fi supuse unor paşi opționali de comprimare (`llvm-strip`) înainte de linking, fără a pierde tabelele de simboluri necesare depanării.

2.3.7 VII. Linking și optimizarea la link-time

Când toate modulele au fost produse, linker-ul `lld` combină secțiunile, rezolvă simbolurile şi aplică relocările. Spre deosebire de link-editarea tradițională, `lld` se bazează pe execuție paralelă şi pe formatul de index *ThinLTO*: fiecare fişier obiect conține un rezumat IR cu semnături hash ale funcțiilor exportate, mărimea lor şi estimări de profil. În timpul link-ului, aceste rezumate sunt analizate pentru a decide, distributiv, ce funcții pot fi inlined şi ce simboluri pot fi eliminate (*dead stripping*), fără a încărca întregul IR în memorie. Rezultatul este un executabil ELF mai compact şi, frecvent, cu salturi indirecte reduse, beneficii vizibile în special pe aplicații mari unde coerența cache-ului de instrucțiuni face diferența. Pentru extensiile descrise în această lucrare, avantajul principal este că secvența generată pentru pseudo-instrucțiunea `DOT` rămâne eligibilă pentru agregare: dacă mai multe module conțin aceeaşi buclă expandată, *ThinLTO* poate decide să o partajeze sau să o specializeze în funcție de profilul de execuție, totul fără a repeta paşi de compilare [17].

2.4 Builtins şi intrinsece în Clang/LLVM – concept extins

2.4.1 Motivația existenței a două niveluri

În cadrul compilatorului Clang, **builtins** sunt funcții speciale recunoscute de compilator, precum `__builtin_popcount` sau `__builtin_prefetch`. Acestea permit generarea de cod optimizat fără a respecta convențiile standard de apel. Builtins sunt definite în fișierele `Builtins*.def` sau `Builtins*.td` pentru fiecare arhitectură[18].

Pentru a menține o separare clară între front-end și back-end, Clang transformă aceste builtins în **intrinsece LLVM**, instrucțiuni pseudo-IR din spațiul de nume `llvm.`, descrise declarativ în fișierele `Intrinsics.td`. Aceste intrinsece transportă informații detaliate despre tipuri, efecte de memorie și alte atribute semantice, facilitând optimizările ulterioare[19].

Această abordare permite ca optimizările să fie aplicate într-un mod independent de arhitectura țintă, asigurând portabilitatea și eficiența codului generat.

2.4.2 Fluxul de procesare

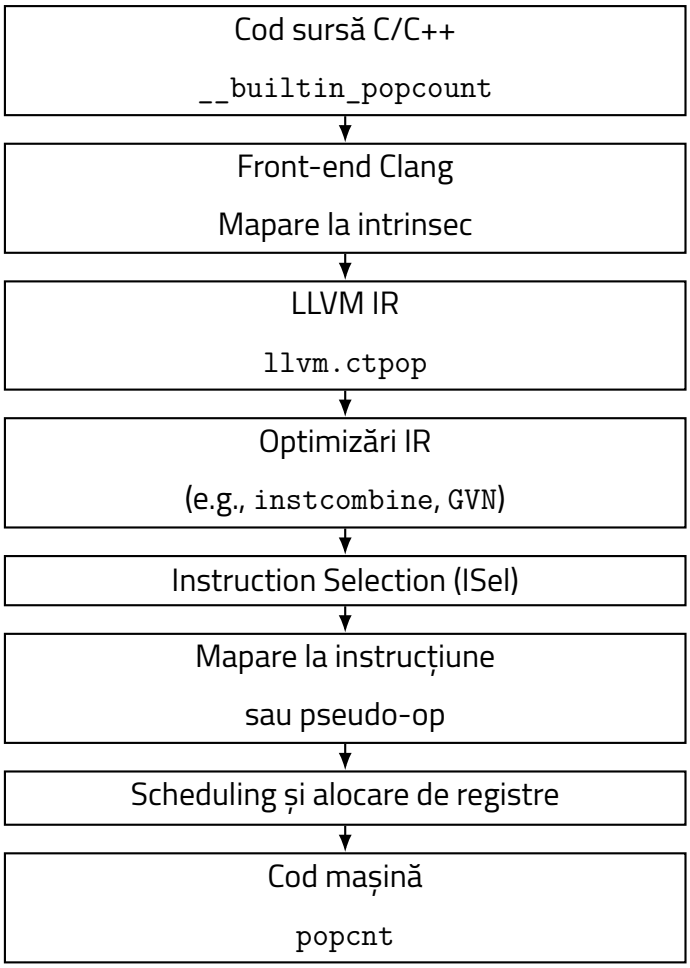

Figura 2: Fluxul de procesare de la builtin Clang până la codul mașină generat de backend-ul LLVM

1. **Declarație în C/C++.** Programatorul scrie în cod o funcție builtin, precum `__builtin_popcount`, disponibilă prin antetul arhitecturii (ex. `<x86intrin.h>` pentru x86 sau `<riscv_vector.h>` pentru RISC-V). Acestea sunt recunoscute direct de front-end-ul Clang[20].

2. **Mapare în Clang.** Clang consultă tabelele interne definite în `Builtins*.def` și mapează acest apel la o instrucțiune intermediară specială numită **intrinsec** (`llvm.ctpop` în acest caz), descrisă în fișierele `Intrinsics.td` [21]. Această etapă evită generarea directă de cod mașină, decuplând arhitectura hardware de codul sursă.

3. **Generare LLVM IR.** Clang emite o instrucțiune LLVM IR din spațiul `llvm.*`, cu informații precise despre tipuri, aliasing, efecte de memorie și contracte semantice precum `willreturn`, `readnone`, `nocapture`, esențiale pentru optimizări ulterioare [22].

4. **Optimizări IR.** Instrucțiunea IR trece prin multiple transformări, inclusiv:

- `instcombine` – combină instrucțiuni redundante;
- `mem2reg` – promovează variabile locale pe registre;
- `simplifycfg` – simplifică fluxul de control;

Aceste optimizări sunt aplicate în mod arhitectură-independent de către middle-end [23].

5. **Selectare de instrucțiuni (ISel).** În backend, intrinsecul este mapat la o instrucțiune nativă sau la o pseudo-instrucțiune în funcție de suportul oferit de procesor. Aceasta este descrisă în fișierele `*.td` (TableGen), cum ar fi `X86InstrInfo.td` sau `RISCVInstrInfo.td` [24]. De exemplu, `llvm.ctpop` se mapează la instrucțiunea x86 `popcnt`, dacă este suportată.

6. **Scheduling și alocare de registre.** După selectarea instrucțiunilor, backend-ul aplică algoritmi de planificare a execuției și alocare a registrelor. Se generează instrucțiuni target finale ținând cont de constrângerile arhitecturale [25].

7. **Emitere cod.** Codul final, binar sau assembly, este emis. Dacă procesorul suportă instrucțiunea (`popcnt`), aceasta va apărea direct în binar; altfel, se generează fallback software automat [24].

2.4.3 Atribute semantice ale intrinsecelor

Intrinsecele LLVM pot include atribute calitative, care influențează modul în care middle-end-ul aplică optimizări:

- **nocapture** – indică faptul că un argument pointer nu este stocat (capturat) de funcție sau stocat global, facilitând eliminarea variabilelor și aliasing optimizat [21].
- **readnone**, **readonly**, **writeonly** – definesc efectele de memorie ale instrucțiunii (niciun acces, doar citire, sau doar scriere), permițând eliminarea memoriei nefolosite și combinarea accesurilor [21].
- **willreturn** – semnalează că funcția va avea terminare determinată, ceea ce ajută la decizii corecte de inlining și transformări de buclă [21].

2.4.4 Exemple de mapări între builtins și intrinsece

x86 `__builtin_ia32_pabsb128` → `llvm.x86.sse2.pabs.b` – instrucțiune SSE2 pentru valoare absolută, mapând un builtin Clang la intrinsec LLVM și la instrucțiune hardware.

ARM `__builtin_arm_qadd` → `llvm.arm.qadd` – intrinsec de adunare cu saturare pe ARM, recunoscut la nivel IR și apoi generat ca instrucțiune hardware.

RISC-V `__builtin_riscv_clz` → `llvm.riscv.clz` – instrucțiune Count Leading Zeros, builtin definit în '<riscv_vector.h>', mapat la intrinsec RISC-V și transpus la instrucțiunea reală de hardware.

2.4.5 Avantajele utilizării intrinsec-elor

Intrinsecele `llvm.*` aduc următoarele avantaje în cadrul compilatorului LLVM:

- **Portabilitate îmbunătățită.** Odată definit intrinsecul, același cod sursă poate fi compilat pe multiple arhitecturi: backend-ul poate genera instrucțiune hardware dacă există suport sau, altfel, poate furniza o implementare de fallback în software, fără a afecta codul C/C++ [21].
- **Optimizări semantice avansate.** Atributele intrinsecelor (de exemplu *nocapture*, *readnone*) permit IR-ului să efectueze optimizări precum eliminarea codului mort, deduplicarea expresiilor și propagare de constante, care nu ar fi posibile fără aceste metadate precise.
- **Extensibilitate fără complexitate.** Adăugarea unei noi funcționalități începe de obicei cu un intrinsec; acest proces este mult mai simplu decât implementarea unei instrucțiuni noi în codul backend, iar schimbările în nivelurile superioare (front-end, optimizări) sunt minime [20].

2.5 TableGen: motorul declarativ

TableGen este un DSL creat pentru proiectele LLVM, Clang sau MLIR, care permite definirea declarativă a entităților precum instrucțiuni, registre, builtins sau intrinsece. Scopul principal este să reducă monotonia și erorile codului boilerplate, oferind un mod modular de a genera automat fișiere C++ complexe și reutilizabile [26].

2.5.1 Ieșiri automat generate

Pe baza definițiilor din fișierele `*.td`, TableGen produce în mod automat:

a) **Encoder binar („bit-pusher")** – încarcă bitii în formatul corect al instrucțiunii.

b) **Disassembler** – creează tabele care permit reconstituirea instrucțiunii din codul binar.

c) **Pattern-uri ISel** – sunt regulile utilizate de selectorul de instrucțiuni pentru a transpune IR-ul în instrucțiuni hardware concrete.

2.5.2 Definire și scalabilitate

O instrucțiune RISC-V R-type, precum 'ADD', se definește în doar câteva linii în format declarativ:

```
tablegen
def ADD : RVInstR<0b0000000, 0b000, OPC_OP,
  (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2), "add $rd, $rs1, $rs2">;
```

Din această singură definiție, TableGen generează aproximativ 150 de linii de C++ pentru encoder, pattern ISel și disassembler, fără cod manual [27].

2.5.3 Facilități avansate de reutilizare

TableGen oferă în plus:

- class – template-uri predefinite pentru structuri complexe.
- multiclass + defm – instanțe multiple pe baza aceleiași clase.
- directivelor let, if, foreach – pentru configurări declarative dinamice.

Aceste facilități elimină repetiția codului și scad eroarea umană, permițând creșterea rapidă și sigură a arhitecturilor definite prin top-down.

2.5.4 De ce contează pentru dezvoltatori?

Utilizarea sistemului `TableGen` în cadrul LLVM aduce beneficii semnificative pentru dezvoltatori, atât din perspectiva vitezei de dezvoltare, cât și a mentenanței codului.

Rapiditatea este un avantaj esențial. O nouă instrucțiune poate fi definită declarativ într-un fișier `.td` (TableGen), iar codul auxiliar necesar este generat automat în timpul compilării. Astfel, dezvoltatorul nu mai este nevoit să scrie manual cod boilerplate pentru fiecare instrucțiune nouă introdusă.

În al doilea rând, **coerența** și consistența codului rezultat sunt asigurate automat. Deoarece informațiile sunt centralizate într-o formă declarativă, orice modificare realizată într-un fișier `.td` se propagă automat în toate componentele care folosesc acea instrucțiune (parsing, ISel, disassembler, printer etc.). Acest lucru reduce riscul apariției de erori sau de inconsistențe între componentele LLVM.

Nu în ultimul rând, sistemul oferă un grad ridicat de **extensibilitate**. Adăugarea de instrucțiuni noi sau modificarea celor existente nu necesită intervenții manuale în codul sursă C++ al backend-ului, ci doar actualizarea descrierilor din `.td`. Această separare între descrierea declarativă și codul generat automat facilitează evoluția rapidă a backend-ului, mai ales în fazele experimentale sau în dezvoltarea de extensii personalizate.

2.5.5 Utilizare și popularitate

`TableGen` reprezintă una dintre cele mai importante componente declarative din infrastructura LLVM. În prezent, există peste 1300 de fișiere cu extensia `.td` în repository-ul oficial LLVM, totalizând aproximativ 500.000 de linii de cod declarativ. Aceste fișiere sunt utilizate pentru a descrie instrucțiuni, tipuri de date, reguli de selecție a instrucțiunilor (ISel), mecanisme de optimizare, registre și multe alte entități target-specific.

Importanța sa nu se limitează doar la backend-urile clasice (cum ar fi x86, ARM sau RISC-V), ci se extinde și în cadrul proiectului MLIR (Multi-Level Intermediate Representation), unde `TableGen` este utilizat intensiv pentru definirea dialectelor, a operațiilor tensoriale și a regulilor de tip inferență și transformare.

Această utilizare extinsă reflectă atât eficiența în dezvoltare, prin centralizarea logicii într-un format ușor de procesat și întreținut, cât și scalabilitatea, prin suportul pentru generarea automată de cod pentru sute de instrucțiuni și mecanisme complexe. Popularitatea sa este dublată de faptul că este considerat un limbaj de descriere indispensabil în ecosistemul LLVM, fiind întreținut activ de comunitatea open-source și actualizat constant pentru a susține noile cerințe ale dezvoltării de compilatoare moderne [28].

2.6 GlobalISel pe scurt

2.6.1 Ce este GlobalISel?

Începând cu LLVM 16, GlobalISel a fost introdus ca un nou framework de selecție a instrucțiunilor, menit să înlocuiască infrastructurile mai vechi cum ar fi SelectionDAG și FastISel. Spre deosebire de acestea, GlobalISel transformă direct LLVM-IR-ul într-o reprezentare intermediară specifică mașinii (*Machine IR*, MIR), fără a fi nevoie de grafuri de tip DAG sau logică specială de conversie [16]. Aceasta permite un proces de selecție mai modular, mai ușor de extins și mai rapid.

2.6.2 Avantajele lui GlobalISel pe RISC-V

- **Suport direct pentru tipuri scalabile.** Se introduc tipuri vectoriale scalabile (`<vscale x n x i32>`), iar GlobalISel le poate gestiona nativ în MIR fără necesitatea construirii unui SelectionDAG masiv sau a adaptărilor complexe [29].

- **Modularitate și claritate.** Spre deosebire de abordarea monolitică din SelectionDAG, GlobalISel încurajează structuri modulare cu exploatarea statică a pattern-urilor și claselor de instrucțiuni pe bază de MIR, facilitând adăugarea de extensii sau optimizări customizate pentru vectori.

- **Performanță și timp de compilare redus.** Fiind construit să lucreze direct pe MIR (nativ), GlobalISel reduce costurile asociate construirii și munging-ului unui DAG, ducând la compilații mai rapide, un beneficiu esențial pentru coduri heavy în vectori precum cele în HPC și AI.

2.6.3 Limitări actuale

GlobalISel este încă marcat ca experimental pentru RISC-V. Optimizările post-ISel sunt mai puține în comparație cu SelectionDAG și, prin urmare, multe distribuții continuă să utilizeze DAG pentru codul scalar și GlobalISel doar pentru codul vectorial, asigurând un echilibru între maturitate și extensibilitate.

2.7 Tooling esențial în ecosistem

În întregul lanț LLVM, câteva unelte sunt esențiale și oferă avantaje concrete în proiecte mari, cum ar fi suportul pentru RISC-V:

- `lld` (linker-ul LLVM) – înlocuitorul rapid al linker-ului standard, oferă viteze de 5–10× mai mari comparativ cu `ld.bfd`, conform măsurătorilor realizate de Phoronix pentru executabile mari

(> 1 GB) [30]. Este complet compatibil cu formatele ELF, COFF și Mach O și suportă LTO (Link Time Optimization) fără plugin-uri suplimentare.

- `clang-tidy` – instrument de analiză statică bazat pe LLVM, folosit intens pentru detectarea erorilor obișnuite de cod, respectarea stilului („linting") și refactorizare automată. Este modular, se integrează direct în CI/CD și suportă antete precum '<clang-tidy>'. A fost utilizat cu succes pentru auditarea codului kernelului Linux și oferă o gamă largă de „checks" pentru stil, siguranță și performanță [31], [32].

- Sanitizerele LLVM (ASan, UBSan, TSan, CFI) – activează instrumentare în timpul compilării care detectează erori de memorie, comportament nedefinit sau probleme de concurență. AddressSanitizer (ASan) și UndefinedBehaviorSanitizer (UBSan) sunt deja suportate în LLVM pentru RISC-V începând cu versiunea 15, permițând depanare avansată pe kernel-uri și aplicații bare-metal [33].

- `llvm-strip` și `llvm-objcopy` – unelte rapide pentru comprimare și modificare de secțiuni binare, utile pentru firmware sau imagini bare-metal, mai eficiente decât omologii GNU.

- `llvm-profdata` + `llvm-cov` – pentru profiling și analiză de acoperire, se integrează perfect cu compilarea cu Clang și oferă vizualizare granulară a performanței și testarii.

2.8 Indicatori de performanță – explicații detaliate

Tabelul prezentat oferă o comparație clară între compilarea cu Clang + LLD + ThinLTO și compilarea cu GCC + ld.bfd. Fiecare metrică este explicată pentru a evidenția importanța optimizărilor în proiecte de amploare.

Tabel 1: Performanță Clang 17 vs. GCC 13 pe RV64GC (Linux 6.8, quad-core, 1.2 GHz)

Metrică	**Clang 17**	**GCC 13**	**Detalii și semnificație**
Timp build (kernel)	78 min	96 min	Clang + LLD + ThinLTO oferă o reducere de 19%, economisind aproape 20 de minute per build (important pentru dezvoltare rapidă) [30].
SPEC CPU 2017	1.02×	1.00×	Clang este în medie cu 2% mai performant, excelent la benchmark-uri CPU intensive (ex. mcf, xz) [30].
Dimensiune binar	31 MB	35 MB	Opțiunea `-Oz` + instrucțiuni compresate îmbunătățesc densitatea codului, utilă pe sisteme embeded [30].
Linii modificate ISA	1.000	3.000	Clang necesită doar aproximativ 1k linii declarative TableGen pentru o instrucțiune nouă, spre deosebire de aproximativ 3k linii C++ în GCC, arată eficiența integrării [34].

3 BeagleV-Fire ca platformă hibridă RISC-V + FPGA

BeagleV-Fire reprezintă un exemplu de vârf al noii generații de computere monocip (Single Board Computers - SBC), fiind construită în jurul SoC-ului PolarFire® MPFS025T de la Microchip. Această platformă combină un procesor RISC-V de clasă industrială cu un FPGA reconfigurabil de înaltă performanță, adresând cerințele tot mai mari ale aplicațiilor moderne ce necesită procesare paralelă, accelerare hardware, precum și control predictibil al consumului energetic [35].

Dispozitivul este proiectat de către BeagleBoard.org în colaborare cu Microchip și Seeed Studio, fiind destinat atât comunității de dezvoltatori open-source, cât și mediului academic sau industrial. Scopul său principal este acela de a oferi un ecosistem de dezvoltare robust pentru testarea, prototiparea și implementarea soluțiilor bazate pe arhitectura RISC-V, cu posibilitate de extindere în logica hardware configurabilă oferită de FPGA [36].

Unul dintre cele mai importante aspecte ale BeagleV-Fire este prezența celor 5 nuclee de procesare RISC-V (4 general-purpose RV64GC și 1 de tip monitor RV64IMAC) ce pot rula un sistem de operare Linux complet funcțional, în paralel cu un FPGA performant care poate fi utilizat pentru accelerare de kernel-uri compute-intensive, procesare în timp real sau integrare de periferice personalizate. Astfel, se creează un sistem eterogen cu capabilități de **co-procesare simetrică și asincronă**, caracteristic platformelor embedded de nouă generație.

Prin integrarea cu ecosistemul open-source (Linux RISC-V, Yocto Project, OpenOCD, LLVM), BeagleV-Fire devine un nod important în cercetarea academică și în dezvoltarea aplicațiilor embedded de viitor, cum ar fi rețelele neuronale accelerate, controlul robotizat sau sistemele ciber-fizice în timp real.

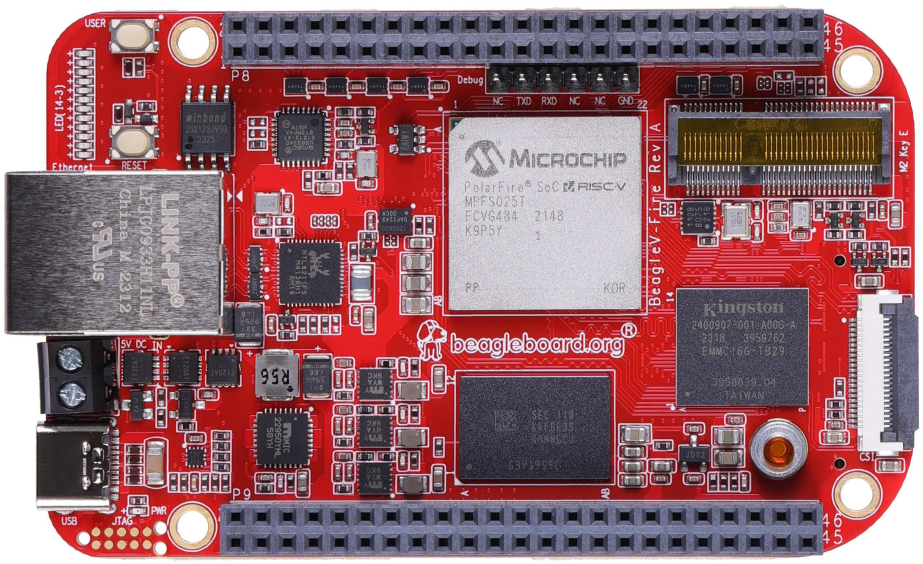

Figura 3: Plăcuța BeagleV-Fire — vedere frontală. [37]

3.1 Arhitectura SoC-ului PolarFire MPFS025T

SoC-ul **PolarFire MPFS025T**, dezvoltat de *Microchip Technology*, reprezintă elementul central al plăcuței **BeagleV Fire**, fiind o platformă System-on-Chip (SoC) avansată ce îmbină, într-un singur cip, un set de nuclee RISC V de înaltă performanță cu un FPGA reconfigurabil din clasa PolarFire.

Această arhitectură hibridă oferă un echilibru excelent între performanță computațională, flexibilitate hardware și eficiență energetică. Integrând logică programabilă cu procesoare general-purpose, MPFS025T permite dezvoltatorilor să creeze soluții în care sarcinile software tradiționale pot fi accelerate prin componente hardware personalizate direct pe FPGA, fără a depăși limitele consumului termic sau electric.

Prin această abordare, SoC-ul permite implementarea de aplicații complexe precum procesare de semnal digital, machine learning, algoritmi criptografici, control în timp real sau interfețe de comunicații de mare viteză, toate pe o singură platformă hardware. În plus, suportul pentru standardul RISC V și capabilitățile extinse de securitate îl fac atractiv pentru domenii precum automatizare industrială, vehicule autonome, rețele inteligente și cercetare academică.

Integrarea strânsă dintre componentele procesorului și logica FPGA este susținută de o arhitectură internă pe bază de magistrale AMBA AXI de înaltă performanță, asigurând o comunicare rapidă și predictibilă între subsistemele SoC-ului.

PolarFire MPFS025T este unul dintre primele SoC-uri comerciale care reușește să combine procesarea RISC V cu o fabrică FPGA performantă, într-o manieră ce permite reducerea timpului de dezvoltare și testare, oferind în același timp performanță și scalabilitate în implementări embedded moderne [38], [39].

3.1.1 Componentele principale ale SoC-ului

SoC-ul PolarFire MPFS025T integrează o varietate de componente hardware de înaltă performanță, care lucrează sinergic pentru a oferi o platformă flexibilă, eficientă energetic și adaptabilă aplicațiilor moderne embedded.

- **FPGA Integrat (CFG fabric)** – SoC-ul dispune de o fabrică FPGA bazată pe aproximativ 23.000 de *logic elements*, fiecare element incluzând un LUT (Look-Up Table) cu patru intrări și un flip-flop asociat. Aceste resurse logice pot fi configurate pentru a implementa diverse funcționalități hardware, precum acceleratoare personalizate, procesoare dedicate sau controlere specializate. În plus, fabrica include 68 de *Math Blocks*, adică unități de multiplicare-adunare (MAC) de tip 18x18, extrem de utile în aplicații precum procesare digitală de semnal (DSP), criptografie sau rețele neuronale convoluționale [35], [38].

- **Nuclee RISC V** – Sub sistemul de procesare este alcătuit dintr-un nucleu E51 și patru nuclee U54. Nucleul E51, cu setul de instrucțiuni RV64IMAC, este destinat activităților de *boot*, monitorizare și management general al sistemului. Cele patru nuclee U54, compatibile RV64GC (General-purpose + Compressed + Floating Point), sunt optimizate pentru execuție de aplicații intensive, rulând la frecvențe de până la 667 MHz. Acestea oferă performanțe teoretice de aproximativ 3.1 CoreMarks/MHz și 1.7 Dhrystone MIPS/MHz, suficiente pentru majoritatea sarcinilor embedded moderne [39], [40].

- **Memorie L2 și Scratchpad** – MPFS025T dispune de un cache L2 configurabil de 2 MB, care poate funcționa fie ca memorie cache partajată între nuclee, fie ca *scratchpad* RAM, o zonă de memorie rapid accesibilă pentru aplicații ce necesită latență minimă. Această memorie este direct accesibilă și de către FPGA, facilitând partajarea de date între componentele software și hardware într-un mod eficient [32].

- **MHS & Control** – Pentru managementul inițializării, monitorizării și securității, SoC-ul integrează un microcontroler ARM Cortex M3 ce rulează în mod izolat de restul sistemului. Acesta se ocupă cu secvențierea alimentării, încărcarea configurației FPGA, precum și gestionarea evenimentelor critice. Comunicarea între microcontroler și celelalte componente se face prin magistrale AXI de mare lățime, garantând o transmisie eficientă a comenzilor și a stării sistemului [39].

- **I/O și periferice integrate** – Platforma include un set extins de interfețe de intrare-ieșire, proiectate pentru a susține o gamă largă de aplicații industriale și embedded. Acestea includ: două controlere Gigabit Ethernet cu PHY dedicat, suport USB 2.0 OTG pentru conectivitate universală, cinci porturi UART pentru comunicație serială, două magistrale SPI și două I^2C pentru conectarea senzorilor sau altor dispozitive periferice, două interfețe CAN 2.0 pentru aplicații automotive, o interfață SD/MMC pentru stocare și un conector M.2 Key-E pentru module Wi-Fi sau extensii [35], [38].

- **SerDes de mare viteză** – SoC-ul este echipat cu patru canale SerDes (*Serializer/Deserializer*) ce operează la viteze de până la 12.7 Gbps. Aceste canale permit comunicarea prin interfețe precum PCI Express (PCIe), extensii de memorie de mare viteză sau comunicații de tip JESD204, fiind esențiale în aplicații cu cerințe mari de lățime de bandă, precum transmisii video, rețelistică de înaltă performanță sau acceleratoare AI [35], [38].

3.1.2 Schema bloc – componentele interne

Arhitectura internă a SoC-ului PolarFire MPFS025T este organizată pe module funcționale distincte, interconectate printr-o rețea de tip **AMBA AXI fabric**, care asigură o latență redusă, lățime de bandă mare și coerență între unitățile de procesare, memorie și periferice.

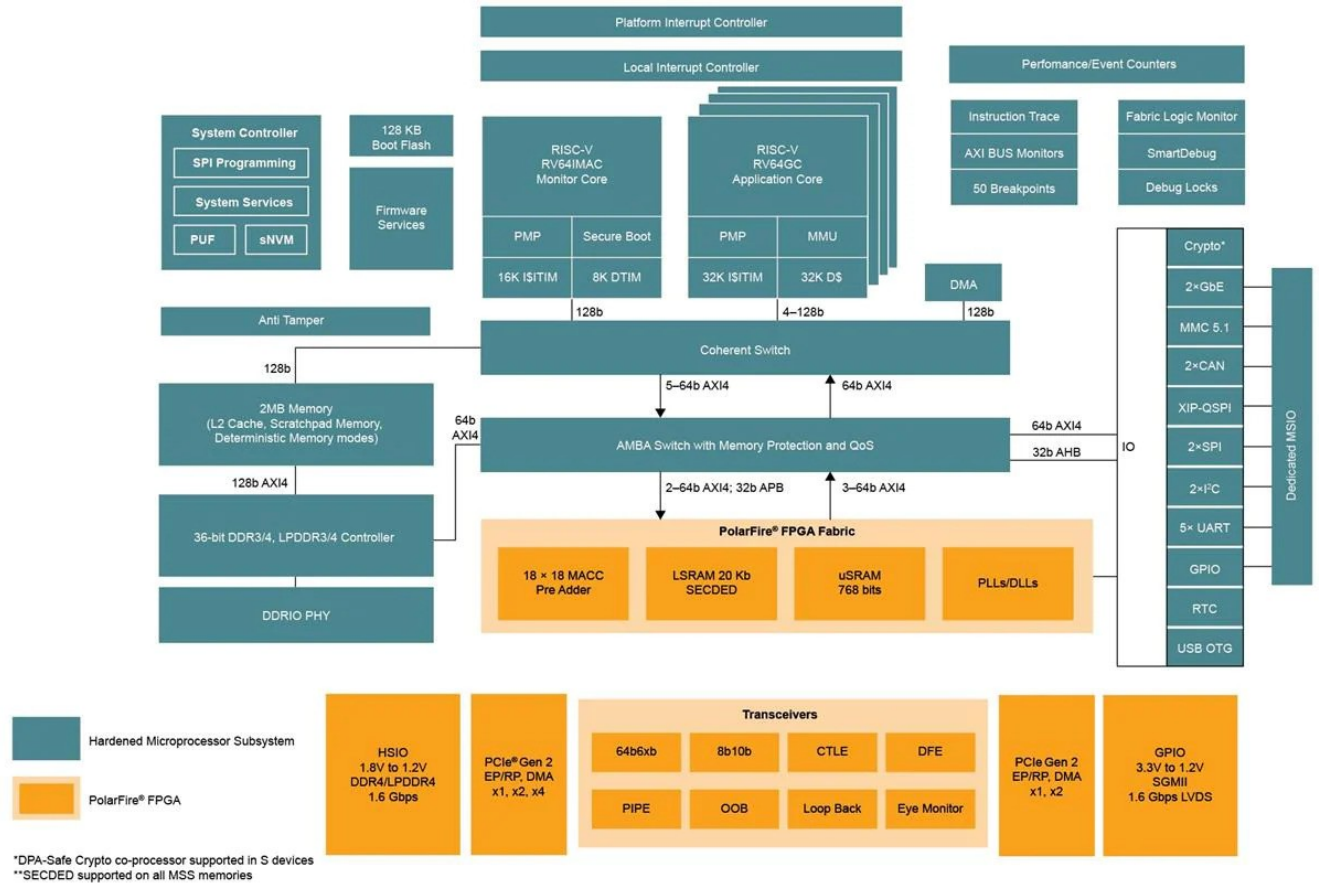

Figura 4: Schema bloc a SoC-ului PolarFire MPFS025T, evidențiind fabricul FPGA, nucleele RISC V, memoriile și interconectările AXI.[41].

- **Substratul FPGA** – Fabricul FPGA (în portocaliu în schema bloc) integrează resursele logice reconfigurabile și unitățile de procesare numerică (MAC blocks), alături de memorie locală LSRAM/uSRAM, PLL-uri și logica de ceasuri. Prin conectarea directă la switch-ul AXI de 128 de biți, fabricul permite accelerarea aplicațiilor compute-intensive prin co-procesoare hardware personalizate și module DMA eficiente.

- **Clusterul microprocesor (MSS)** – Subsystem-ul procesorului (MSS — Microprocessor Subsystem), colorat în albastru în diagramă, include un nucleu E51 (boot/monitor) și patru nuclee U54 (RV64GC) pentru aplicații. Acestea sunt conectate la memorie de tip L2 de 2 MB, configurabilă ca `cache` sau `scratchpad`, precum și la controller-ul DDR extern prin magistrale AXI de 64/128 de biți.

- **AMBA AXI Interconnect** – Fabricul AXI multi-lățime (32, 64, 128 biți) facilitează comunicația între toate subsistemele: nucleele RISC V, fabricul FPGA, controlerul de memorie DDR, perifericele I/O (UART, SPI, I²C, CAN, USB, SD/MMC etc.), precum și modulele DMA și perifericele SerDes pentru conectivitate PCIe Gen2 [39].

- **Controler și periferice** – Un controller Cortex M3 este responsabil pentru inițializarea sistemului, încărcarea configurației FPGA și monitorizarea securizată. Acesta gestionează, printre altele, funcțiile *Anti-Tamper*, boot flash, SPI programming, serviciile de firmware și criptografie hardware.

FPGA-ul și nucleele RISC-V operează pe aceeași sursă de alimentare, asigurând funcționare predictibilă pe RP-LV [39].

3.1.3 Capabilități de procesare numerică

Una dintre cele mai valoroase resurse ale SoC-ului PolarFire MPFS025T este prezența blocurilor matematice dedicate (**Math Blocks**), integrate în cadrul fabricii FPGA. Aceste blocuri includ multiplicatoare specializate de tip 18×18 cu **pre-adder** și suport nativ pentru operații de tip MAC (*Multiply-Accumulate*). Fiecare bloc este optimizat pentru performanță și consum energetic scăzut, fiind o soluție excelentă pentru accelerarea sarcinilor de procesare numerică intensivă.

Fabrica FPGA a SoC-ului oferă:

- **68 de blocuri MAC** de 18x18 biți, integrate direct în siliciu, utilizabile în paralel;

- **Frecvență de operare** de până la 500 MHz în condiții nominale (1.05 V, 0–85 °C);

- **Performanță industrială garantată** de până la 365 MHz în intervale extinse de temperatură (–40...+125 °C);

- **Consum energetic redus**: fabricile PolarFire sunt optimizate pentru *low static power*, făcând aceste blocuri ideale pentru aplicații mobile, edge sau alimentate prin baterie.

Aceste caracteristici permit accelerarea locală a algoritmilor de tip:

- **Machine learning**: rețele neuronale simple (MLP, RNN) cu inferență locală accelerată;

- **Procesare de semnal digital (DSP)**: filtre finite/infinite (FIR/IIR), transformate Fourier (FFT);

- **Sisteme de control**: calcul rapid al regimurilor în timp real, incluzând control PID și feedforward;

- **Criptografie**: operații modulare și multiplicări rapide necesare în algoritmi ECC și RSA.

O caracteristică importantă este faptul că aceste MAC blocks pot fi instanțiate direct în logica HDL a utilizatorului (Verilog/VHDL), dar pot fi accesate și indirect prin intermediul unei interfețe DMA între procesorul RISC V și FPGA, pentru o integrare hibridă CPU–accelerator.

Performanța oferită este comparabilă cu soluții dedicate ASIC în anumite scenarii, dar oferă flexibilitatea de reconfigurare specifică FPGA-urilor. Acest lucru este deosebit de valoros pentru aplicații în care specificațiile algoritmilor evoluează rapid sau unde este necesară personalizarea fluxului de date.

3.2 Interacțiunea cu sistemul de operare

Distribuția oficială Linux pentru BeagleV Fire utilizează un kernel mainline actualizat, împreună cu bootloader-ul U Boot, asigurând compatibilitate nativă pentru periferice (Ethernet, USB C, MMC, SPI, etc.) și o experiență standard de utilizare [42]. Imaginile pot fi stocate pe eMMC sau pe card microSD, iar logarea se realizează prin consolă serială disponibilă la conectarea cablului USB C.

Pentru interacțiunea cu placa BeagleV-Fire, există mai multe metode de acces și monitorizare, utile atât în faza de dezvoltare, cât și în etapa de depanare. Una dintre cele mai directe și fiabile metode de comunicare este consola serială, accesibilă prin dispozitivul `/dev/ttyUSB0`. Această interfață permite conectarea prin programe specializate precum `screen`, `minicom` sau `picocom`, utilizând setările standard de comunicație: 115200 baud, 8 biți de date, fără paritate și un bit de stop (configurația 8–N–1). Comunicarea serială este esențială în situațiile în care sistemul de fișiere nu este încă montat sau nu este disponibilă o interfață grafică. În aceste condiții, consola serială devine singura metodă viabilă de interacțiune cu sistemul și de diagnosticare a eventualelor probleme apărute la pornire sau în stadiile incipiente ale bootării [43].

După ce placa este conectată la rețeaua locală prin cablu Ethernet, aceasta va primi automat o adresă IP de la serverul DHCP al rețelei. În condițiile în care serviciile de configurare a rețelei, precum `NetworkManager` sau `systemd-networkd`, sunt active, este posibilă stabilirea unei conexiuni SSH cu sistemul de operare rulând pe placă. În mod uzual, conexiunea poate fi realizată prin comanda `ssh root@beaglev-fire.local`, presupunând că avahi-daemon (mDNS) rulează corespunzător pe gazdă, facilitând astfel un acces transparent și prietenos pentru utilizator [43].

Pentru monitorizarea în timp real a funcționării sistemului, dar și pentru depanare, se pot utiliza o serie de comenzi uzuale în mediul Linux. Jurnalele kernelului, de exemplu, pot fi consultate cu ajutorul comenzii `dmesg`, care oferă o imagine cronologică a evenimentelor importante din timpul boot-ului și al funcționării sistemului. Structura sistemului de fișiere, precum și dispozitivele de stocare montate, pot fi analizate prin `lsblk`, în timp ce modulele active ale kernelului pot fi listate cu `lsmod`. Pentru in-

terfețele de comunicație precum SPI, I²C și UART, informațiile detaliate pot fi accesate prin comenzile `lspci` sau prin explorarea directă a ierarhiei `/sys/bus`. Aceste instrumente sunt esențiale pentru a avea o imagine clară și precisă a stării curente a resurselor hardware și a driverelor aferente.

În ceea ce privește interacțiunea cu pini GPIO și extensii hardware, placa oferă conectorii standard P8 și P9, ce pot fi accesați prin mecanismul sysfs sau prin biblioteca modernă `libgpiod`. Utilizatorii pot conecta diverse cape-uri (plăcuțe de extensie) pentru a adăuga funcționalități adiționale, cum ar fi interfețe I²C, SPI, ADC și altele. Aceste extensii sunt compatibile cu infrastructura oferită de BeagleV-Fire și sunt descrise în detaliu în documentația oficială [44].

Un aspect important în contextul dezvoltării hardware este posibilitatea de a verifica versiunea curentă a gateware-ului (softul FPGA-ului) încărcat în sistem. Această verificare se poate realiza direct din mediul Linux, explorând structura `/proc/device-tree/chosen/overlays/`, unde pot fi identificate fișierele asociate cu versiunea gateware-ului activ. Mai precis, comanda: `cat /proc/device-tree/chosen/overlays/*/version` returnează versiunea exactă, deseori exprimată ca un tag Git, fapt ce facilitează auditarea configurației și reproducerea fidelă a mediului de execuție, esențială în scenarii de cercetare și testare repetabilă [45].

3.3 Executabile native și testare

Pe platforma BeagleV Fire, aplicațiile dezvoltate în limbaje de programare de nivel înalt, precum C sau C++, pot fi compilate local și executate nativ, fără a necesita etape suplimentare de transfer sau conversie. Acest mod de lucru oferă o cale directă și eficientă de validare funcțională, în special în stadiile incipiente ale dezvoltării sau în cadrul ciclurilor de testare rapidă. Compilarea locală pe dispozitiv permite o relație directă cu mediul de execuție, inclusiv cu kernelul, bibliotecile partajate și hardware-ul specific sistemului embedded. Astfel, posibilele incompatibilități sau constrângeri arhitecturale sunt identificate mai devreme, iar procesul de depanare este simplificat datorită rulării în contextul real al aplicației finale.

Această metodă este deosebit de utilă în scenarii de prototipare rapidă, unde ideile sau modulele funcționale trebuie validate înainte de a fi integrate într-un sistem mai mare. În plus, pentru testarea unor biblioteci compilate nativ sau a unor drivere de nivel jos (low-level), execuția directă pe sistemul țintă permite o interacțiune imediată cu componentele hardware ale plăcii, fără a fi necesară emularea sau simularea acestora.

Totuși, în cazul proiectelor de dimensiuni mai mari sau atunci când este necesară automatizarea procesului de build, este recomandată utilizarea unui sistem gazdă (host), de obicei bazat pe arhitectura x86_64. Pe acesta se configurează un toolchain dedicat de tip cross-compiling, care poate gene-

ra binare compatibile cu arhitectura țintă RV64GC, specifică procesorului RISC-V de pe BeagleV Fire. Avantajele acestei abordări includ timpi de compilare semnificativ reduși, posibilitatea folosirii unor IDE-uri avansate, integrarea cu sisteme de versionare sau CI/CD, precum și separarea clară între mediul de dezvoltare și cel de execuție.

Transferul executabilelor rezultate se poate face simplu, prin metode precum `scp`, `rsync` sau chiar partajarea unui fișier compilat printr-un mediu de rețea locală. În plus, folosind debuggeri precum `gdbserver`, testele și depanările pot fi efectuate chiar și de la distanță, de pe mașina gazdă, cu suport complet pentru simboluri de depanare.

Această dualitate între compilarea nativă și compilarea în regim cross oferă flexibilitate dezvoltatorilor, care pot alege strategia adecvată în funcție de complexitatea aplicației, de resursele hardware disponibile sau de nivelul de control necesar asupra procesului de build. Detaliile privind configurarea toolchain-ului și exemple de utilizare practică sunt prezentate în Secțiunea 4, unde este descrisă în profunzime metodologia adoptată pentru acest proiect.

3.4 Monitorizare și depanare

În procesul de dezvoltare embedded, monitorizarea atentă a comportamentului sistemului și capacitatea de a identifica și remedia rapid eventualele disfuncționalități reprezintă etape esențiale pentru asigurarea unei funcționări corecte și stabile a aplicațiilor. Platforma BeagleV-Fire, fiind bazată pe o distribuție Linux compatibilă cu majoritatea utilitarelor standard din ecosistemul Unix like, oferă un mediu familiar și eficient pentru dezvoltatori. Aceștia pot utiliza unelte robuste de monitorizare, deja consacrate în domeniul administrării sistemelor, pentru a obține o imagine detaliată a stării sistemului, a resurselor utilizate și a modului în care procesele rulează în timp real.

Monitorizarea resurselor de sistem

Pentru a evalua utilizarea procesorului, a memoriei și a resurselor de stocare într-un mod accesibil și detaliat, se pot folosi diverse comenzi puse la dispoziție de sistemul de operare. Una dintre cele mai comune este comanda `top`, care oferă o vizualizare în timp real a proceselor active, procentajul de utilizare a CPU-ului, consumul de memorie RAM și nivelul de încărcare a sistemului. Alternativa mai avansată, `htop`, include o interfață colorată, organizată pe coloane intuitive și permite sortarea dinamică a proceselor după diverse criterii, precum utilizarea CPU-ului sau memoria consumată. De asemenea, `htop` permite uciderea proceselor direct din interfață, ceea ce poate fi extrem de util în timpul depanării.

Pentru a înțelege disponibilitatea actuală a memoriei și a spațiului de swap, comanda `free -h`

furnizează o prezentare clară, în unități ușor de citit (MB/GB), a cantității de memorie totală, utilizată, liberă și rezervată pentru cache sau buffer. Această comandă este utilă mai ales în identificarea scurgerilor de memorie sau în evaluarea presiunii asupra swap-ului în condiții de sarcină ridicată.

Pentru o perspectivă asupra performanței sistemului de intrare/ieșire, comanda `iostat` este indispensabilă. Aceasta oferă informații despre rata de citire și scriere pe dispozitivele de stocare (precum eMMC sau cardul SD), precum și despre timpul de așteptare mediu al operațiunilor. Astfel, poate evidenția blocaje sau întârzieri cauzate de limitări ale I/O-ului, frecvente în aplicațiile embedded cu acces intens la fișiere.

În completare, comanda `vmstat` (Virtual Memory Statistics) permite monitorizarea unei game mai largi de parametri de sistem, inclusiv utilizarea procesorului, activitatea memoriei virtuale, operațiile de swap, blocurile I/O și activitatea proceselor în așteptare. Această comandă este utilă în diagnosticarea problemelor legate de congestia proceselor sau de saturarea memoriei și poate fi folosită în paralel cu celelalte unelte pentru o analiză mai completă a stării sistemului.

Aceste instrumente combinate formează un set fundamental de monitorizare, fără de care dezvoltarea și validarea unei aplicații embedded robuste ar fi dificil de realizat. Utilizarea lor regulată, în special în timpul testării sub sarcină sau în condiții limită, contribuie semnificativ la obținerea unui sistem fiabil și performant.

3.4.1 Instrumente de profiling pentru optimizare

Pentru aplicațiile compute-intensive (CPU-bound), este necesară analiza comportamentului la nivel de funcție sau instrucțiune. Acest lucru se poate realiza cu utilitarul `perf`, inclus în majoritatea distribuțiilor Linux:

- `perf stat ./test.out` — oferă statistici agregate precum număr de instrucțiuni, rate de cache miss, număr de cicluri.
- `perf record ./test.out` — înregistrează profilul de execuție.
- `perf report` — afișează o analiză detaliată a consumului de resurse pe funcții.

În cazul aplicațiilor optimizate cu LLVM sau cu cod generat din instrucțiuni vectoriale, `perf` poate ajuta la înțelegerea impactului transformărilor IR și al structurii de memorie.

3.4.2 Depanare prin GDB și gdbserver

Pentru depanare la nivel de cod sursă, aplicațiile pot fi compilate cu flagul `-g` pentru includerea simbolurilor de debug. Există două metode principale de utilizare:

- **Local:** GDB poate fi rulat direct pe BeagleV-Fire, dacă resursele sistemului permit.
- **Remote:** Se lansează `gdbserver` pe placă, care ascultă pe un port TCP:

```
gdbserver :1234 ./test.out
```

Apoi, pe sistemul host, se lansează GDB (compatibil cu RV64GC) și se conectează la target:

```
target remote beaglev-fire.local:1234
```

Această metodă este preferabilă când debugging-ul are loc pe fișiere binare generate prin cross-compiling, într-un mediu cu toolchain configurat corespunzător.

3.4.3 Importanța testării în context real

Testarea aplicațiilor direct pe platforma BeagleV Fire joacă un rol crucial în procesul de dezvoltare embedded, oferind un cadru autentic de evaluare a funcționalității și performanței codului. În comparație cu simulările sau emularea pe arhitecturi diferite, rularea în contextul real al hardware-ului țintă aduce în prim-plan o serie de factori care nu pot fi reproduceți fidel în medii abstracte. Printre aceștia se numără latențele reale ale canalelor de intrare/ieșire, comportamentul determinat de carac-teristicile sursei de alimentare (variații de tensiune, consum energetic în sarcină), precum și constrângerile impuse de accesul simultan la periferice fizice precum UART, SPI, I²C sau GPIO.

Un alt aspect esențial este testarea în prezența timpilor de reacție reale, critice în aplicațiile cu cerințe de tip real-time sau în sistemele de control distribuit. De exemplu, întârzierile apărute la procesarea întreruperilor sau la propagarea semnalelor între subsisteme nu pot fi evaluate complet decât în mediu nativ, în care interacțiunile hardware-software sunt pe deplin active.

Această testare contextuală este indispensabilă înainte de efectuarea oricărei forme de optimizare. În absența unei validări temeinice, optimizările la nivel de cod sursă (refactorizări, restructurări algoritmice), de toolchain (prin flag-uri de compilare sau transformări IR în LLVM), sau la nivel de hardware (prin integrarea cu acceleratoare FPGA) pot introduce erori subtile sau instabilități greu

de diagnosticat ulterior. De asemenea, o testare riguroasă în acest stadiu furnizează metrice de referință clare, care pot fi comparate ulterior pentru a evalua impactul real al optimizărilor aplicate.

Mai mult, debugging-ul direct pe BeagleV Fire permite utilizarea eficientă a instrumentelor oferite de distribuția Linux instalată pe placă. Prin monitorizarea proceselor, vizualizarea log-urilor kernelului, analiza performanțelor I/O și inspecția manuală a stării sistemului, dezvoltatorul obține o imagine detaliată și contextualizată a comportamentului aplicației.

Pentru mai multe detalii despre utilizarea sistemului și a instrumentelor incluse în distribuția BeagleV Fire Linux, se poate consulta documentația oficială disponibilă la [42].

4 Setup experimental

4.1 Cross-compiling: concept, utilitate și aplicabilitate

În dezvoltarea pentru arhitecturi embedded, cum este cazul SoC-ului RISC-V din plăcuța BeagleV-Fire, procesul de construire și testare a aplicațiilor software implică provocări specifice. Sistemele embedded sunt, prin natura lor, limitate în ceea ce privește puterea de procesare, memoria disponibilă și capacitatea de stocare. De aceea, compilarea directă pe dispozitivul țintă, cunoscută sub numele de compilare nativă, poate deveni lentă, consumatoare de resurse și, în unele cazuri, imposibilă pentru aplicații mai complexe.

Pentru a depăși aceste limitări, se utilizează o metodă standard în industria embedded: **cross-compiling**. Aceasta presupune folosirea unui *toolchain* specializat (compiler, linker, assembler, biblioteci etc.) instalat pe un sistem gazdă (de obicei x86_64 Linux sau Windows) pentru a produce fișiere binare care rulează pe un sistem țintă cu arhitectură diferită (în cazul nostru, RISC-V 64-bit, RV64GC).

4.1.1 Definiție formală

Cross-compiling-ul este procesul prin care un program este compilat pe o platformă (**host**) pentru a fi executat pe o altă platformă (**target**). Spre deosebire de compilarea nativă, unde codul este compilat și rulat pe același sistem, în cross-compiling sistemul host are o arhitectură diferită de cea a sistemului target. În cazul plăcii BeagleV-Fire, sistemul host este un PC cu arhitectură x86_64, iar sistemul target este un SoC RISC-V.

4.1.2 Exemplu ilustrativ

`Host: x86_64 → [riscv64-linux-gnu-gcc] → Binary ELF64 → Target: RISC-V (BeagleV-Fire)`

În acest lanț, compilatorul cross (`riscv64-linux-gnu-gcc`) traduce sursa C/C++ într-un executabil ELF64 compatibil cu Linux-ul care rulează pe SoC-ul PolarFire MPFS025T.

4.1.3 Motive pentru utilizarea cross-compiling-ului

Chiar dacă BeagleV Fire este echipat cu un procesor capabil și rulează o distribuție completă de Linux, viteza de compilare nu se compară cu cea a unui sistem desktop modern. Pe un laptop performant, timpul de build poate fi de 10 până la 20 de ori mai mic, ceea ce duce la un ciclu de dezvoltare mult mai rapid și iterativ.

De asemenea, resursele hardware disponibile pe placă sunt considerabil mai limitate. Cu doar 2 GB de memorie RAM și un spațiu de stocare eMMC restrâns, compilarea locală a unor proiecte de dimensiune medie sau mare poate deveni impracticabilă. Aceste limitări pot cauza erori de tip out-of-memory sau pot duce la degradarea performanței în timpul execuției aplicațiilor, mai ales atunci când sistemul trebuie să ruleze în paralel și alte servicii sau procese critice.

Separarea responsabilităților dintre mediul de dezvoltare și mediul de execuție oferă și un avantaj clar din perspectiva organizării și automatizării. Compilarea pe host permite integrarea naturală a procesului de build în pipeline-uri de tip CI/CD (Continuous Integration/Continuous Deployment), precum GitHub Actions sau GitLab CI. Astfel, întregul proces de construire a executabilului poate avea loc în cloud sau pe o stație locală de dezvoltare, fără a necesita acces fizic la dispozitivul embedded. Executabilul rezultat este apoi transferat și testat direct pe BeagleV Fire, economisind timp și resurse.

Un alt beneficiu major constă în capacitățile avansate de debugging pe care le oferă compilarea pe host. Instrumente precum gprof, perf sau valgrind necesită simboluri de debugging care pot fi generate complet pe sistemul host și apoi utilizate pe target pentru profilare și depanare. Această abordare permite o analiză detaliată a performanței, precum și identificarea exactă a zonelor critice de cod.

Nu în ultimul rând, flexibilitatea toolchain-ului pe sistemul host este mult mai mare. Utilizatorul poate instala și testa mai multe versiuni de compilatoare, precum GCC 12 configurat pentru arhitectura RISC V sau Clang cu suport pentru extensii experimentale. Totodată, se pot aplica patch-uri personalizate, se pot activa extensii ISA specifice (precum vectorizare sau built-in-uri dedicate) și se pot experimenta opțiuni avansate de optimizare, cum ar fi -march=rv64gc, -mtune=thead-c906, -O2, -Os, -flto, ș.a.m.d.

4.1.4 Suport Clang/LLVM pentru cross-compiling RISC-V

Pe lângă toolchain-ul GNU clasic, `riscv64-linux-gnu-gcc`, utilizat frecvent pentru cross-compiling a aplicațiilor C/C++, ecosistemul RISC-V beneficiază de suport extins și în cadrul proiectului **LLVM**.

Clang, compilatorul front-end al LLVM, oferă suport pentru generarea de cod pentru arhitectura `riscv64`, atât pentru compilare nativă, cât și pentru cross-compiling. Avantajele majore includ:

- **Integrare superioară cu lanțuri de optimizare moderne**, precum GlobalISel, vectorizare automată (AutoVectorization) și instrumente precum `llvm-mca`, `llvm-objdump` și `llvm-size`.
- **Configurabilitate avansată.** Parametrii de tip `-target riscv64-linux-gnu` sau `-march=rv64gc -mabi=lp64d` pot fi utilizați pentru a specifica arhitectura țintă, ABI-ul și opțiunile de tuning.
- **Testare și extensibilitate.** În contextul acestui proiect, LLVM este platforma aleasă pentru a introduce un builtin personalizat (operația `dot` scalară). Datorită structurii modulare a LLVM, adăugarea de instrucțiuni noi, optimizări și lowerings este mult mai controlabilă decât în alte toolchain-uri.
- **Cross-debugging cu `lldb`.** Complementar lui `gdb`, `lldb` oferă unelte avansate pentru depanare pe ținte RISC-V, în special în scenarii cu cod instrumentat LLVM IR.

Compilarea unui fișier simplu pentru RISC V folosind Clang, pe sistemul host:

```
clang --target=riscv64-linux-gnu -march=rv64gc -mabi=lp64d \
      -o test_clang.out test.c
```

Acest toolchain LLVM poate fi compilat din sursă, cu suport RISC-V activat, utilizând opțiunile de `CMake` și `llvm-project`. Versiunea utilizată în cadrul proiectului este bazată pe commitul `cd708029e0b2` din upstream LLVM [46].

4.1.5 Utilizare practică pentru BeagleV Fire

Distribuția Linux instalată pe BeagleV Fire este compatibilă cu binarele ELF construite pentru ABI-ul `riscv64-linux-gnu`. Așadar, folosind un toolchain adecvat (precum cel de la `riscv-collab/riscv-gnu-toolchain`), putem compila local pe PC programe C/C++, apoi le transferăm pe placă prin SCP și le rulăm direct.

Această tehnică este esențială mai ales în scenarii precum:

- Dezvoltarea de drivere kernel și module;
- Implementarea și testarea de acceleratoare hardware;

- Rularea de benchmark-uri sau algoritmi de procesare numerică intensivă;
- Instrumentare și optimizare a aplicațiilor în timp real.

4.1.6 Cross vs. Native Compilation – o comparație

Aspect	**Compilare nativă (pe placă)**	**Cross-compiling (pe host)**
Performanță	Limitată de CPU-ul embedded	Rapidă, datorită procesorului host
Dimensiune build	Depinde de spațiul disponibil pe placă	Spațiu larg pe host
Control asupra toolchain-ului	Restricționat la ce oferă distribuția	Complet — putem configura GCC, Clang, opt
Debugging	Posibil local, dar lent	Remote debugging eficient prin `gdbserver`
Scalabilitate	Nepractică pentru proiecte mari	Ideală pentru proiecte complexe cu CI

Tabel 2: Comparație între compilarea nativă și cross-compiling.

4.1.7 Cazul BeagleV Fire

În cazul plăcuței BeagleV Fire, utilizarea unui cross-compiler precum `riscv64-linux-gnu-gcc` permite dezvoltarea aplicațiilor de pe un sistem gazdă (Ubuntu, Fedora, Arch etc.), urmată de transferul și rularea binarelor pe placă. Deoarece sistemul de operare de pe BeagleV este compatibil cu ABI-ul `riscv64-linux-gnu`, aplicațiile compilate astfel pot rula fără recompilare sau adaptare suplimentară.

În continuarea acestei secțiuni, vom detalia pașii pentru:

- instalarea toolchain-ului pe sistemul host;
- compilarea unui program de test;
- transferul pe placă;
- rularea și depanarea acestuia.

Pentru referință, documentația oficială a plăcii și toolchain-ul RISC-V GCC sunt disponibile la [42], [47].

4.2 Context hardware și software pentru instalare

Pentru configurarea unui mediu de dezvoltare robust, folosim un sistem host Windows 11 cu următoarele specificații hardware:

Tabel 3: Specificații hardware ale calculatorului host

Componentă	**Specificații**
CPU	Intel Core i5-12450H (12 CPUs), 2.0 GHz
Memorie	16 384 MB RAM (Dual-Channel)
GPU	NVIDIA GeForce RTX 3050
VRAM	3964 MB
Sistem de operare	Windows 11 Pro 64-bit (Build 22631)

În loc de instalare dual-boot sau virtualizare completă, se utilizează **WSL2 (Windows Subsystem for Linux 2)**. Acesta oferă:

- un kernel Linux real, rulează ca mașină virtuală lightweight;
- suport complet pentru distribuții moderne (ex. Ubuntu 22.04);
- integrare directă cu filesystem-ul și aplicațiile Windows;
- acces complet la rețea, tooluri standard de dezvoltare și versiuni multiple de Python, GCC, CMake, Git etc.

Această alegere permite rularea comenzilor de build și compilare din terminalul Ubuntu fără pierderi de performanță semnificative. În secțiunile următoare vom descrie instalarea concretă a toolchain-urilor GNU și LLVM pe acest mediu.

4.3 Instalarea mediului pentru Cross-Compiling

4.3.1 Instalarea Ubuntu prin WSL2

Cel mai eficient mod de a configura un mediu Linux pentru cross-compiling pe un PC cu Windows 10 sau Windows 11 este prin utilizarea Windows Subsystem for Linux 2 (WSL2). Acesta permite rularea unui kernel Linux real într-o mașină virtuală lightweight, oferind performanță apropiată de nativ și integrare directă cu sistemul de fișiere Windows.

WSL2 este a doua generație a subsistemului Windows pentru Linux, care introduce un adevărat kernel Linux (bazat pe tehnologia de virtualizare Hyper-V) spre deosebire de versiunea anterioară

(WSL1), care traducea apelurile de sistem Linux în apeluri de sistem Windows. Avantajele majore ale WSL2 includ:

- **Compatibilitate completă cu sistemele Linux**, incluzând suport pentru majoritatea distribuțiilor, inclusiv Ubuntu, Fedora, Debian etc.
- **Performanță îmbunătățită la rularea aplicațiilor Linux**, datorită kernelului real.
- **Suport pentru sistemul de fișiere Linux ext4**, oferind o mai bună compatibilitate pentru toolchain-uri de dezvoltare.
- **Integrare cu Windows**, permițând rularea simultană a aplicațiilor Windows și Linux și schimb de fișiere între sistemele de operare.

```
wsl --install
```

Secvență de Cod 1: Instalare WSL2

Această comandă va instala automat platforma WSL, kernelul Linux și ultima versiune de Ubuntu disponibilă. După finalizare, sistemul necesită un restart. Dacă este necesară o anumită versiune de distribuție (ex. Ubuntu 22.04), aceasta poate fi instalată ulterior din Microsoft Store. Ghiduri detaliate sunt disponibile în documentația oficială Microsoft [48].

4.3.2 Configurarea Ubuntu în WSL2

Odată instalat și inițializat Ubuntu prin WSL2, se deschide terminalul Linux și se actualizează sistemul pentru a evita probleme legate de pachete învechite:

```
sudo apt update && sudo apt upgrade
```

Secvență de Cod 2: Actualizarea și upgrade-ul sistemului

Pentru a compila toolchain-ul GNU, este necesară instalarea unor pachete de dezvoltare:

```
sudo apt install autoconf automake autotools-dev curl python3 python3-pip python3-
    tomli libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo
     gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake
    libglib2.0-dev libslirp-dev
```

Secvență de Cod 3: Instalarea pachetelor necesare

Acestea sunt necesare pentru configurarea mediului de compilare GNU, cât și pentru construirea proiectelor LLVM din sursă (în special pachetele `cmake`, `ninja` și `glib2.0-dev`).

4.3.3 Instalarea toolchain-ului `riscv-gnu-toolchain`

După instalarea dependențelor, se creează un director de lucru pentru toolchain-ul GNU RISC-V:

```
mkdir -p ~/riscv
cd ~/riscv
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=$HOME/riscv/install
make linux
```

Secvență de Cod 4: Clonarea repository-ului și compilare

Compilarea poate dura între 15–45 de minute, în funcție de performanța sistemului. Directorul de instalare va conține compilatorul `riscv64-unknown-linux-gnu-gcc` și unelte precum `objdump`, `gdb` și `strip`, adaptate pentru platforma țintă.

Pentru a face toolchain-ul accesibil global, se adaugă în fișierul `~/.bashrc`:

```
export RISCV="$HOME/riscv/install"
export PATH="$RISCV/bin:$PATH"
```

Secvență de Cod 5: Configurarea variabilelor PATH

Se salvează modificările și se aplică cu `source ~/.bashrc`, după care se verifică instalarea:

```
riscv64-unknown-linux-gnu-gcc --version
```

Secvență de Cod 6: Verificare toolchain GCC

Un output tipic:

```
riscv64-unknown-linux-gnu-gcc (g04696df09) 14.2.0
Copyright (C) 2024 Free Software Foundation, Inc.
```

Secvență de Cod 7: Exemplu output GCC

Odată acest pas finalizat, sistemul este pregătit pentru cross-compiling folosind GCC.

4.3.4 Instalarea Clang & LLVM

Pentru detalii despre motivația utilizării LLVM în contextul cross-compiling RISC-V, avantajele față de GCC și aplicabilitatea sa în extensii (ex: builtin-uri personalizate), se recomandă consultarea Secțiunii 2.

Instalarea Clang & LLVM în cadrul proiectului se face prin activarea suportului dedicat în `riscv-gnu-toolchain`. Pașii necesari sunt următorii:

4.3.4.1 Activarea suportului LLVM în riscv-gnu-toolchain

Versiunea oficială a `riscv-gnu-toolchain` oferă opțiunea de a construi și Clang/LLVM cu suport RISC-V, prin adăugarea parametrului `--enable-llvm` în procesul de configurare. Astfel, următorii pași se adaugă după instalarea dependențelor prezentate anterior:

```
cd ~/riscv/riscv-gnu-toolchain
./configure --prefix=$HOME/riscv/install --enable-llvm --enable-linux
make -j1
```

Secvență de Cod 8: Configurare pentru compilarea cu Clang & LLVM

Explicații:

- `--enable-llvm` activează compilarea infrastructurii LLVM în cadrul toolchain-ului, incluzând componente esențiale precum clang (frontend-ul compilatorului), lld (linker-ul LLVM) și opt (instrumentul de optimizare pentru cod intermediar LLVM IR). Această configurație este necesară atunci când se dorește dezvoltarea de extensii personalizate, precum builtin-uri dedicate arhitecturii RISC-V sau analize detaliate asupra fluxului de compilare.

- `--enable-linux` asigură construirea unui toolchain complet compatibil cu sistemele de operare Linux care rulează pe arhitectura RISC-V. Această opțiune activează suportul pentru glibc și alte componente esențiale, permițând generarea de executabile care pot fi rulate direct pe plăcuța BeagleV-Fire, fără a necesita adaptări suplimentare.

- `-j1`, aceasta este utilizată în cadrul comenzii make pentru a forța compilarea secvențială (pe un singur thread). Această alegere este justificată în medii cu resurse limitate, precum WSL2, unde compilarea paralelă ar putea conduce la consumarea excesivă a memoriei RAM și apariția unor erori de tip `OOM` (Out of Memory). Deși acest lucru crește durata totală a compilării, oferă stabilitate și previne blocarea procesului de build.

Compilarea LLVM poate dura între 60 și 120 de minute, în funcție de resursele disponibile. La final, fișierele binare vor fi instalate în directorul specificat prin `--prefix`.

4.3.4.2 Verificarea instalării Clang

După finalizarea compilării, se recomandă verificarea faptului că binarele LLVM au fost generate corect:

```
$RISCV/bin/clang --version
```

Secvență de Cod 9: Verificarea versiunii Clang

Exemplu de output corect:

```
clang version 19.1.7 (https://github.com/llvm/llvm-project.git
    cd708029e0b2869e80abe31ddb175f7c35361f90)
Target: riscv64-unknown-linux-gnu
Thread model: posix
InstalledDir: /home/user/riscv/install/bin
```

Secvență de Cod 10: Exemplu output Clang

4.4 Setarea simulatorului ISA RISC-V: Spike

În etapa de testare a aplicațiilor compilate pentru arhitectura RISC-V, utilizarea unui simulator ISA devine extrem de utilă, mai ales înainte de rularea pe placa fizică. Simulatorul oficial dezvoltat în cadrul ecosistemului RISC-V este **Spike**, cunoscut și sub denumirea de *riscv-isa-sim*. Acesta permite rularea și depanarea aplicațiilor într-un mediu complet virtualizat, fără a depinde de hardware-ul fizic. Pentru a rula binarele RISC-V pe Spike, este necesar un bootloader și un kernel minimalist, rol îndeplinit de componenta **riscv-pk** (Proxy Kernel) [49].

4.4.1 Clonarea și instalarea Proxy Kernel-ului (`riscv-pk`)

Pentru a putea rula aplicații RISC-V pe simulatorul Spike, este necesar un mic kernel care să ofere suport pentru execuția programelor în absența unui sistem de operare complet. Acesta este rolul îndeplinit de **Proxy Kernel** (prescurtat `pk`), o componentă minimalistă care implementează un subset de apeluri de sistem și asigură încărcarea binarelor în simulatorul Spike [49].

Pentru o organizare clară a build-urilor, se recomandă separarea în două directoare:

- `build_elf` — pentru aplicații bare-metal în format ELF, fără sistem de operare;
- `build_linux` — pentru aplicații care presupun rularea sub un mediu Linux (cu glibc etc.).

4.4.1.1 Clonarea repository-ului oficial

```
git clone https://github.com/riscv-software-src/riscv-pk
```

Secvență de Cod 11: Clonarea Proxy Kernel-ului

4.4.1.2 Compilare pentru aplicații bare-metal (fără OS)

```
mkdir build_elf && cd build_elf
../riscv-pk/configure --prefix=$RISCV --host=riscv64-unknown-elf
make
make install
```

Secvență de Cod 12: Build bare-metal (ELF)

4.4.1.3 Compilare pentru aplicații Linux (cu glibc)

```
mkdir build_linux && cd build_linux
../riscv-pk/configure --prefix=$RISCV --host=riscv64-unknown-linux-gnu
make
make install
```

Secvență de Cod 13: Build pentru Linux target

Această separare permite testarea binarelor în funcție de mediul țintă: bare-metal pentru aplicații simple, sau Linux embedded pentru execuție avansată.

4.4.2 Instalarea Spike (simulatorul oficial ISA RISC-V)

Spike este simulatorul oficial dezvoltat de comunitatea RISC-V pentru execuția binarelor conform specificației ISA. Acesta simulează instrucțiunile RISC-V la nivel de set de instrucțiuni, fără emulare completă a hardware-ului, ceea ce îl face rapid și ideal pentru testarea logicii de aplicație și depanare [50].

4.4.2.1 Clonarea codului sursă Spike

```
git clone https://github.com/riscv-software-src/riscv-isa-sim
```

Secvență de Cod 14: Clonarea Spike

După clonare, se creează un director de build separat pentru organizare:

```
cd riscv-isa-sim
mkdir build && cd build
../configure --prefix=$RISCV
```

Secvență de Cod 15: Configurare build Spike

4.4.2.2 Instalarea dependențelor necesare compilării

Înainte de compilarea simulatorului, se instalează pachetele necesare:

```
sudo apt install device-tree-compiler libboost-regex-dev libboost-system-dev
```

Secvență de Cod 16: Instalarea dependențelor

Acestea sunt esențiale pentru:

- `device-tree-compiler` — compilarea fișierelor `.dts` (Device Tree Source) în `.dtb`, folosite pentru descrierea configurației hardware;
- `libboost-regex-dev`, `libboost-system-dev` — suport pentru expresii regulate și apeluri sistem, utilizate de backend-ul simulatorului.

4.4.3 Testarea funcționalității simulatorului Spike

După instalare, se poate testa funcționalitatea cu un binar ELF compilat pentru RISC-V:

```
spike pk test_program.elf
```

Secvență de Cod 17: Exemplu de rulare cu Spike

Această comandă inițializează simulatorul, încarcă kernelul proxy (`pk`) și rulează aplicația. Este un pas important în validarea binarelor generate înainte de testarea pe hardware-ul BeagleV-Fire.

4.5 Testarea mediu de dezvoltare: Toolchain + Simulator

După configurarea completă a toolchain-urilor (GCC sau Clang) și instalarea simulatorului Spike, este esențial să validăm funcționalitatea întregului lanț de compilare și execuție. Pentru aceasta, se va scrie un program simplu în C care afișează un mesaj pe ecran.

4.5.1 Scrierea unui program de test (Hello RISC-V)

Creăm un fișier C cu numele `hello.c` care conține următorul cod sursă:

```c
#include <stdio.h>

int main() {
    printf("Hello RISC-V!\n");
    return 0;
}
```

Secvență de Cod 18: Codul sursă "hello.c"

Acest program conține doar o instrucțiune de afișare, fiind ideal pentru verificarea funcționalității de bază a sistemului, de la compilator, până la execuția în mediul RISC-V simulat.

4.5.2 Compilarea programului pentru arhitectura țintă

Programul se compilează utilizând Clang, cu specificarea explicită a țintei ca fiind `riscv64`. Se folosește sintaxa:

```
clang -target riscv64-unknown-linux-gnu -o hello hello.c
```

Secvență de Cod 19: Compilarea fișierului hello.c

Este important de remarcat că:

- Flag-ul `-target` determină backend-ul folosit de Clang — în acest caz, specific pentru platforma `riscv64-unknown-linux-gnu`.
- Opțiunea `-o hello` setează numele executabilului rezultat.

Alternativ, dacă se dorește folosirea GCC în loc de Clang:

```
riscv64-unknown-linux-gnu-gcc -o hello hello.c
```

Secvență de Cod 20: Compilare cu GCC

4.5.3 Rularea executabilului pe simulatorul Spike

După compilare, programul este executat folosind simulatorul Spike, cu ajutorul kernelului proxy (`pk`):

```
spike pk hello
```

Secvență de Cod 21: Rularea executabilului cu Spike

Această comandă:

- Încarcă simulatorul `spike`;
- Pornește kernelul `pk` pentru a oferi funcționalități minime;
- Rulează binarul `hello`, compilat anterior.

4.5.4 Output-ul așteptat

Dacă toți pașii au fost urmați corect, output-ul afișat în terminal va fi:

```
Hello RISC-V!
```

Secvență de Cod 22: Exemplu output corect

Acest rezultat confirmă că:

- toolchain-ul Clang/GCC compilează corect codul sursă;
- simulatorul Spike funcționează;
- binarul este compatibil cu proxy-kernelul și poate fi executat în mediu RISC-V.

Această validare este un punct critic pentru asigurarea funcționării mediului de cross-compiling și simulare, înainte de trecerea la aplicații mai complexe sau la testarea pe plăcuța BeagleV-Fire.

4.6 Transferul Executabilului pe BeagleV-Fire

După ce executabilul a fost generat cu succes pe sistemul host (Windows cu WSL2 sau Ubuntu nativ), următorul pas constă în transferul acestuia către plăcuța **BeagleV-Fire** pentru testare directă pe hardware. Deoarece plăcuța rulează un sistem Linux, cea mai eficientă metodă de conectare și transfer este utilizarea protocolului **SSH** împreună cu **SFTP**.

4.6.1 Utilizarea MobaXterm pentru transfer și control SSH

O opțiune accesibilă și prietenoasă pentru utilizatorii Windows este aplicația **MobaXterm**[51], un client complet care oferă o interfață grafică pentru conexiuni SSH și SFTP, precum și un terminal integrat cu suport pentru comenzi Unix.

Pentru a începe:

1. Se descarcă MobaXterm de pe site-ul oficial: `https://mobaxterm.mobatek.net/`
2. Se deschide aplicația și se creează o sesiune nouă de tip **SSH**.
3. Se completează următoarele câmpuri:
   - **Remote Host:** `192.168.7.2` (IP-ul implicit al BeagleV-Fire)
   - **Username:** `beagle`
   - **Port:** 22
4. După conectare, în partea stângă se va deschide automat o fereastră SFTP, permițând transferul fișierelor prin drag-and-drop.

Avantajul major al MobaXterm este combinarea într-o singură aplicație a funcționalităților SSH, SFTP și shell interactiv, facilitând controlul complet al plăcuței în timp real.

4.6.2 Alternative: Transfer prin linia de comandă (SCP)

Pentru utilizatorii avansați, transferul executabilului se poate realiza și folosind comanda `scp` (Secure Copy Protocol), direct din terminal:

```
scp hello beagle@192.168.7.2:/home/beagle/
```

Secvență de Cod 23: Transferul fișierului cu scp

Această metodă este eficientă în medii complet terminale (ex. Ubuntu nativ, WSL fără interfață grafică), dar presupune familiaritate cu comenzile Unix.

4.7 Configurarea unui mediu de depanare în Visual Studio 2022

Visual Studio 2022 reprezintă o alegere robustă pentru depanarea front-end-ului și back-end-ului LLVM/Clang pe Windows, oferind un set complet de unelte grafice, integrare nativă cu CMake, suport avansat pentru debugging multi-threaded, control granular asupra procesului de build, precum și facilități moderne precum `IntelliTrace`, `Live Unit Testing` și `Performance Profiler`.

Platforma este recunoscută pentru stabilitatea, extensibilitatea și suportul său extins pentru dezvoltarea aplicațiilor cross-platform și embedded, făcând-o ideală în contextul experimentării și modificării unui compilator complex precum LLVM.

În cadrul acestei lucrări, Visual Studio este utilizat pentru:

- depanarea implementării `__builtin_riscv_dot`, cu posibilitatea de a urmări pas cu pas generarea codului LLVM IR și propagarea acestuia până în faza de selecție a instrucțiunilor specifice arhitecturii RISC-V;
- analiza statică și profilare de performanță a codului generat în timpul execuției simulate, folosind unelte precum `Instrumentation Profiling` sau `Memory Usage Analyzer`;
- integrarea eficientă a toolchain-ului RISC-V (build-uit manual sau importat) cu cod C/C++ local, permițând testarea rapidă și controlul complet asupra versiunii Clang/LLVM utilizate;
- folosirea break-point-urilor și navigarea în context (call stack, memory view, registers etc.) chiar și în sursele interne LLVM (`SelectionDAG.cpp`, `CGBuiltin.cpp`, `RISCVISelLowering.cpp`).

Astfel, Visual Studio devine un instrument central în dezvoltarea și validarea extensiilor personalizate de compilator, combinând confortul interfeței grafice cu rigurozitatea necesară în analiza comportamentului unui toolchain de nivel jos.

Conform documentației oficiale Microsoft, Visual Studio 2022 oferă suport complet pentru CMake, Linux și proiecte embedded, precum și integrare cu toolchains personalizate, ceea ce îl face potrivit inclusiv pentru depanarea codului RISC-V generat local și testat pe platforme hardware externe[52].

4.7.1 De ce ediția Enterprise?

Ediția *Enterprise* Visual Studio 2022 se distinge printr-un set de instrumente avansate care o fac ideală pentru dezvoltarea de nivel profesional, în special în domeniul sistemelor embedded și al compilatoarelor. Spre deosebire de ediția *Community*, care oferă un mediu solid pentru aplicații generale, versiunea *Enterprise* vine cu funcționalități dedicate performanței, depanării și testării continue, esențiale în proiectele complexe precum cele ce implică extensii LLVM sau toolchain-uri personalizate pentru RISC-V.

Unul dintre avantajele notabile este prezența instrumentului Performance Profiler, care oferă o interfață grafică avansată pentru analiza detaliată a performanței aplicațiilor. Acesta permite identificarea rapidă a bottlenecks din cod, inclusiv în componentele compilatorului, și facilitează optimizări de nivel înalt pe baza datelor măsurate în timp real. În plus, integrarea nativă cu sisteme de testare

permite rularea aplicațiilor într-un mod instrumentat, fără a fi necesară instrumentarea manuală a codului sursă.

Un alt instrument exclusiv ediției *Enterprise* este IntelliTrace, un sistem de urmărire a execuției care păstrează un istoric detaliat al operațiilor efectuate în timpul depanării. Această caracteristică permite dezvoltatorului să revină la stări anterioare ale aplicației, să analizeze contextul execuției fără relansarea aplicației și să reproducă cu fidelitate pașii care au condus la un anumit bug. Pentru dezvoltarea de compilatoare, unde comportamentele neașteptate pot apărea în faze intermediare de transformare a codului, IntelliTrace se dovedește un aliat esențial.

Funcționalitatea de Live Unit Testing adaugă o dimensiune suplimentară procesului de testare, deoarece permite verificarea automată și în timp real a modificărilor aduse codului sursă. Orice modificare declanșează reexecutarea testelor relevante, iar rezultatele sunt actualizate instantaneu în interfața editorului. Astfel, este facilitată o dezvoltare iterativă și sigură, în care regresele sunt identificate imediat, fără a fi necesară rularea manuală a suitei de teste după fiecare schimbare.

În contextul specific sistemelor embedded sau al dezvoltării cross-platform, Visual Studio 2022 Enterprise oferă suport complet pentru debugging Linux și remote debugging pe dispozitive embedded. Acest lucru se realizează printr-o integrare solidă cu toolchain-uri externe, precum GCC pentru RISC-V, și permite depanarea aplicațiilor care rulează pe dispozitive precum BeagleV Fire, direct din IDE, cu suport pentru breakpoints, watch-uri, și inspecția memoriei în timp real.

Pentru studenți și cadre didactice, accesul gratuit la această ediție premium este posibil prin intermediul programului **Azure Dev Tools for Teaching**. Prin autentificarea cu un cont instituțional universitar, oricine din mediul academic poate beneficia legal de toate funcționalitățile ediției *Enterprise*, transformând Visual Studio într-o platformă ideală pentru explorarea aprofundată a arhitecturilor moderne și a tehnologiilor de compilare avansate [53].

4.7.2 Instalarea toolchain-ului RISC-V în MSYS2 MinGW64

Înainte de a integra LLVM/Clang în Visual Studio, este esențial să avem un compilator RISC-V funcțional nativ pe Windows, util pentru testare, verificare a generării codului și rulare locală. Acest lucru se poate realiza prin MSYS2, un mediu POSIX modern pentru Windows, care oferă acces la toolchain-uri open-source prin managerul de pachete `pacman` (același sistem folosit în Arch Linux)[54].

4.7.2.1 Actualizarea sistemului și a pachetelor

După instalarea MSYS2 (de la `https://www.msys2.org/`), se recomandă actualizarea completă a pachetelor pentru a evita incompatibilitățile:

```
pacman -Syu    # actualizeaza pachetele de baza
# se va inchide automat terminalul, redeschideti-l
pacman -Su     # finalizeaza actualizarea
```

4.7.2.2 Instalarea toolchain-ului RISC-V

```
pacman -S mingw-w64-x86_64-riscv64-unknown-elf-toolchain
```

Pachetul include tot ce este necesar pentru compilarea codului C/C++ pentru platforme bare-metal:

- `riscv64-unknown-elf-gcc`, `g++` — compilatoare;
- `ld`, `as`, `objdump`, `gdb` — tool-uri pentru linking, asamblare, depanare;
- suport complet pentru libc minimalistă (`newlib`).

4.7.2.3 Verificarea instalării

Verificarea instalării se face prin:

```
riscv64-unknown-elf-gcc --version
```

Rezultatul ar trebui să fie o versiune GCC dedicată pentru target-ul RISC-V. Acest toolchain va fi util pentru compilarea programelor de test, independent de Clang.

4.7.3 Instalarea componentelor necesare în Visual Studio 2022

Pentru a putea construi și depana eficient unelte din ecosistemul LLVM/Clang, este necesară instalarea anumitor **workload-uri** și **componente individuale** prin `vs_installer.exe`. Acestea sunt esențiale pentru generarea soluției, integrarea cu CMake și depanarea avansată a codului sursă C/C++.

- **Desktop development with C++** — acest pachet activează funcționalitățile de bază ale Visual Studio pentru dezvoltare C/C++ pe Windows, inclusiv MSBuild, debugger nativ, STL și compilatoare pentru Windows (cl.exe). Este indispensabil pentru compilarea codului auxiliar care însoțește proiectele LLVM.

- **Linux and embedded development with C++** — oferă suport complet pentru dezvoltare cross-platform, inclusiv toolchain-uri externe, integrare cu SSH, depanare pe sisteme Linux remote, și tool-uri pentru platforme embedded. Este util în contextul testării codului generat pentru RISC-V, care rulează pe o arhitectură diferită de host-ul Windows.
- **CMake tools for Windows** — CMake este sistemul de build folosit oficial de LLVM. Această componentă permite integrarea completă a fișierelor `CMakeLists.txt` cu IDE-ul Visual Studio, generând automat proiecte `.vcxproj` și soluția `LLVM.sln`. Fără acest pachet, Visual Studio nu va putea interpreta corect structura proiectului.
- `v143 build tools` — include compilatorul MSVC corespunzător versiunii Visual Studio 2022, necesar pentru a construi părțile auxiliare ale LLVM, chiar dacă targetul principal este RISC-V. Asigură compatibilitate cu sistemele moderne de build și permite compilarea codului de test local.
- `C++ ATL` — ATL (*Active Template Library*) este utilă pentru unele librării interne sau plugin-uri care interacționează cu subsistemele Windows. În unele cazuri, anumite dependențe din ecosistemul LLVM/Clang pot solicita acest suport (inclusiv instrumentele de profilare și debug integrate cu interfața Windows).

Fără aceste componente, construirea și depanarea proiectului LLVM/Clang devine fie incompletă, fie instabilă, deoarece mediul nu va recunoaște unele dependințe sau fișiere de proiect generate automat de CMake.

4.7.4 Organizarea directoarelor de proiect

Structura recomandată:

```
D:/riscv/
 llvm-project/     ← cod sursa Clang + LLVM
 llvm-build/       ← directorul de build CMake
```

4.7.5 Generarea soluției Visual Studio cu CMake

Pentru a crea o soluție Visual Studio complet funcțională, adaptată pentru depanarea extensiilor personalizate LLVM, este necesar un pas de configurare inițial. Acesta se face prin rularea `CMake` în cadrul terminalului `x64 Native Tools Command Prompt for VS 2022`, care setează automat variabilele de mediu necesare pentru MSVC și MSBuild.

Comanda completă este:

```
"Path\To\Visual Studio\Common7\IDE\CommonExtensions\
 Microsoft\CMake\CMake\bin\cmake.exe" ^
 -DCMAKE_BUILD_TYPE=Debug ^
 -DLLVM_TARGETS_TO_BUILD="RISCV" ^
 -DLLVM_ENABLE_PROJECTS="clang" ^
 -DLLVM_DEFAULT_TARGET_TRIPLE="riscv64-unknown-elf" ^
 -DCMAKE_VS_JUST_MY_CODE_DEBUGGING=ON ^
 -DLLVM_USE_CRT_RELEASE=MT ^
 ../llvm-project/llvm
```

Secvență de Cod 24: Generarea soluției LLVM pentru Visual Studio

4.7.5.1 Explicația fiecărui argument:

- `-DCMAKE_BUILD_TYPE=Debug` — configurează build-ul în mod `Debug`, generând fișiere intermediare cu informații de simboluri. Aceasta este esențială pentru depanarea pas cu pas, permițând plasarea de breakpoints în sursa C++ a LLVM.
- `-DLLVM_TARGETS_TO_BUILD="RISCV"` — specifică faptul că doar backend-ul RISC-V trebuie construit. Acest argument reduce semnificativ timpul de compilare, comparativ cu includerea tuturor arhitecturilor suportate de LLVM (ex: X86, ARM, AArch64 etc.).
- `-DLLVM_ENABLE_PROJECTS="clang"` — activează construirea doar a proiectului `clang` (frontend-ul pentru C/C++). În combinație cu backend-ul RISCV, acest lucru permite testarea completă a frontend-to-backend pentru cod sursă C compilat în instrucțiuni RISC-V.
- `-DLLVM_DEFAULT_TARGET_TRIPLE="riscv64-unknown-elf"` — setează targetul implicit pentru compilările Clang. Acest triplet este folosit pentru a genera cod compatibil cu sistemele embedded RISC-V care nu rulează un sistem de operare complet (ex: spike).
- `-DCMAKE_VS_JUST_MY_CODE_DEBUGGING=ON` — activează filtrarea „Just My Code” în Visual Studio, ceea ce înseamnă că debugger-ul va ascunde funcțiile de sistem și se va concentra doar pe codul scris/modificat de utilizator (ex: `CGBuiltin.cpp`, `RISCVISelLowering.cpp`).
- `-DLLVM_USE_CRT_RELEASE=MT` — forțează utilizarea runtime-ului static (`Multi-threaded`), în locul versiunii dinamice (`MD`). Acest lucru face executabilele generate mai portabile, fără dependențe de DLL-uri runtime externe.
- `../llvm-project/llvm` — calea relativă către sursa principală a LLVM. Aceasta trebuie să fie corectă în raport cu directorul de build curent (ex: `llvm-build`).

La finalul rulării, CMake va genera un fișier `LLVM.sln` și multiple fișiere `.vcxproj`, corespunzătoare subcomponentelor LLVM (Clang, TableGen, CodeGen, etc.). Soluția rezultată poate fi deschisă direct în Visual Studio 2022, unde pot fi aplicate breakpoint-uri, analizată execuția și testate extensiile personale adăugate în codul LLVM.

4.7.6 Compilarea în Visual Studio și configurarea debugging-ului

După ce fișierul `LLVM.sln` este generat prin CMake, acesta poate fi deschis în Visual Studio 2022. Primul pas este să configurăm proiectul astfel încât să putem compila `clang.exe` și să-l rulăm cu un fișier de test care folosește builtin-ul personalizat.

4.7.6.1 Pașii necesari sunt următorii:

1. În fereastra *Solution Explorer*, se face **click dreapta pe proiectul** `clang` și se selectează `Set as Startup Project`. Aceasta setează `clang.exe` ca executabil principal al soluției.

2. Tot în meniul contextual, selectează `Build`. Aceasta va construi pentru prima dată binarul `clang.exe`, necesar pentru testare și depanare ulterioară. Compilarea poate dura de la 30 de minute până la 90 de minute în funcție de configurația sistemului.

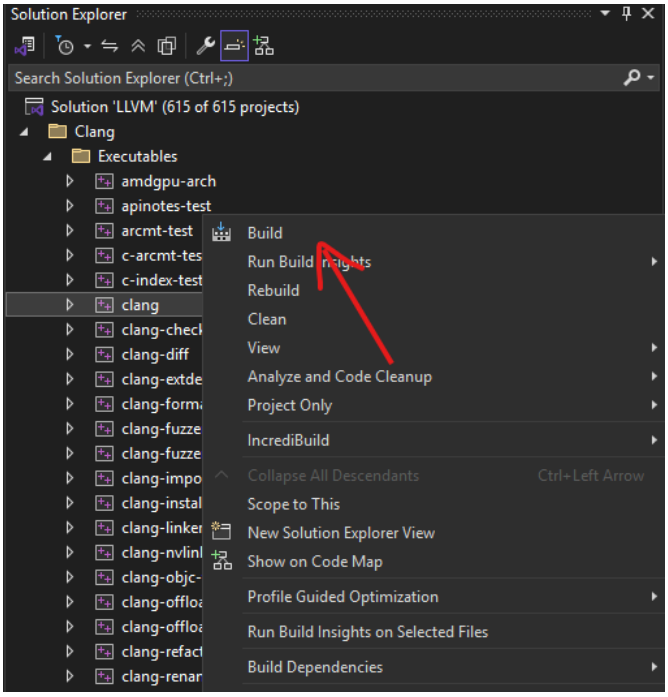

Figura 5: Build Clang în Visual Studio

3. După compilare, se accesează `Properties → Debugging` pentru proiectul `clang`. Aici, în câmpul `Command Arguments`, trebuie introdusă calea către fișierul sursă de test, de exemplu:

```
-O0 --target=riscv64-unknown-elf --gcc-toolchain="C:/msys64/mingw64" --
    sysroot="C:/msys64/mingw64/riscv64-unknown-elf" -mllvm -fast-isel=false -
    mllvm -debug-only=isel,machineinstrs -mllvm -print-before=
    RISCVExpandPseudoPass -mllvm -print-after=RISCVExpandPseudoPass -mllvm -
    print-after=Select -mllvm -verify-machineinstrs Path\To\test_output.c -o
    Path\To\test_output.elf
```

Această comandă spune lui `clang.exe` să încarce fișierul sursă pentru compilare pe arhitectura RISC-V, declanșând astfel codul modificat din `CGBuiltin.cpp`, `RISCVISelLowering.cpp`, etc.

4. Opțional, în câmpul `Working Directory` se poate seta directorul în care se află fișierul de test sau în care se dorește generarea rezultatului (ex: `D:/riscv/tests`).

5. Se pot adăuga **breakpoints** (puncte de oprire) în fișierele sursă relevante ale LLVM, precum:

   - `CGBuiltin.cpp` — responsabil pentru transformarea builtin-urilor Clang în cod intermediar LLVM;
   - `RISCVISelLowering.cpp` — conține logica de mapare a IR-ului în instrucțiuni RISC-V concrete;
   - `RISCVInstrInfo.td`, `RISCVInstrFormats.td` — utile pentru a înțelege cum a fost definită instrucțiunea la nivel TableGen.

6. Se apasă `Start Debugging (F5)` pentru a porni execuția. Visual Studio va atașa automat debugger-ul la procesul `clang.exe`, permițând navigarea pas cu pas prin cod, inspecția variabilelor și a stivei de apeluri.

4.7.6.2 Observație importantă

Este esențial ca **prima compilare** a lui `clang.exe` să fie realizată manual, prin `Build`, înainte de a încerca rularea sub debugger. În caz contrar, Visual Studio poate semnala că executabilul nu există sau nu este actualizat cu modificările recente din sursă.

4.7.7 Avantajul depanării în Visual Studio

Utilizarea Visual Studio 2022 în procesul de dezvoltare și depanare a unui builtin personalizat pentru arhitectura RISC-V oferă o serie de beneficii esențiale, în special în contextul integrării cu sistemul de build LLVM. Platforma oferă o experiență unificată și eficientă, în care toate etapele, de la analiza codului sursă, până la generarea codului mașină, pot fi urmărite în detaliu, într-un mediu grafic prietenos și performant.

Unul dintre cele mai importante avantaje îl reprezintă vizibilitatea completă asupra codului LLVM IR generat. Prin plasarea de breakpoints în fișierele critice ale Clang, cum ar fi CGBuiltin.cpp, precum și în secțiunile dedicate ale LLVM IR, dezvoltatorul poate urmări cu precizie cum se transformă o expresie sursă într-o reprezentare intermediară și, mai apoi, în cod de asamblare. Acest nivel de transparență este indispensabil atunci când se validează corectitudinea și comportamentul unui builtin nou introdus în toolchain.

Mai mult decât atât, Visual Studio permite o urmărire detaliată a fluxului complet de compilare, traversând fără întreruperi atât frontend-ul Clang, cât și backend-ul dedicat arhitecturii RISC-V. Acest lucru înseamnă că fiecare etapă poate fi inspectată: de la analiza sintactică și semantică a codului C, până la selecția efectivă a instrucțiunilor finale în RISCVISelLowering.cpp, respectiv prin mecanismul SelectionDAG. Astfel, este posibil să se înțeleagă exact cum este potrivit pattern-ul TableGen corespunzător builtin-ului și cum se generează efectiv instrucțiunile în backend-ul LLVM.

Un alt avantaj major adus de ediția Enterprise este suportul pentru profilare și benchmarking al codului generat. Instrumentele integrate precum Performance Profiler și Instrumentation Profiling permit evaluarea impactului fiecărei modificări în ceea ce privește performanța: timpi de execuție, alocări de memorie, cache hits sau cicli CPU consumați. Acest tip de analiză este esențial pentru a măsura eficiența builtin-ului comparativ cu implementările native, iar datele obținute pot fundamenta decizii de optimizare ulterioare.

De asemenea, Visual Studio permite rularea directă a fișierelor .c de test folosind clang.exe, cu argumente precise și configurabile din interfața grafică. Astfel, se pot crea scenarii de test recurente, ușor de reexecutat, și se pot salva profiluri de rulare pentru comparații între versiuni. În locul unei interacțiuni prin shell-uri externe, dezvoltatorul beneficiază de o interfață centralizată, care crește productivitatea și reduce timpul de feedback.

Tot în cadrul aceluiași mediu de lucru, este posibilă compararea directă a codului generat de clang cu cel produs de alte compilatoare, precum riscv64-unknown-elf-gcc. Această paralelă este deosebit de valoroasă în etapa de validare funcțională, unde orice diferență de comportament sau performanță poate fi investigată rapid și în context, fără a schimba mediul de lucru.

Un alt aspect important este capacitatea de a urmări în timp real orice modificare adusă codului LLVM și de a o testa imediat în contextul întregului proiect. Compilarea locală a backend-ului modificat, urmată de testarea în cadrul Visual Studio, oferă un ciclu rapid de dezvoltare, permițând iterarea accelerată asupra designului și funcționalității noului builtin.

Aceste funcționalități transformă Visual Studio 2022 într-o soluție robustă și scalabilă pentru depanarea și dezvoltarea compilatoarelor, în special în proiectele educaționale și de cercetare care vizează extinderea arhitecturilor suportate, precum RISC-V.

5 Implementare

Această secțiune descrie procesul de integrare a unui nou `builtin` în lanțul de compilare LLVM pentru arhitectura RISC-V, folosind o abordare **top-down**. Se pornește de la frontend-ul Clang, care interpretează codul sursă scris în C, și se continuă cu backend-ul LLVM, care generează și transformă codul până la nivel de instrucțiune de asamblare pentru platforma țintă.

Implementarea descrisă în această lucrare este disponibilă public pe un fork al proiectului LLVM, accesibil pe platforma GitHub la adresa: `https://github.com/meetzaa/llvm-project` [55]. Proiectul este licențiat sub **Apache License v2.0**, o licență permisivă care permite oricui să studieze, modifice și distribuie codul, atât în scopuri personale, cât și comerciale, cu condiția menținerii notificărilor privind drepturile de autor și licența.

Această deschidere permite comunității să reutilizeze și să îmbunătățească această contribuție, încurajând colaborarea și dezvoltarea extensiilor pentru arhitectura RISC-V în cadrul ecosistemului LLVM.

5.1 Frontend — Clang

Clang joacă un rol esențial în cadrul lanțului de compilare LLVM, având ca principală responsabilitate analiza și conversia codului sursă scris în limbaje de nivel înalt (precum C/C++) într-o formă intermediară numită **LLVM IR (Intermediate Representation)**. Această reprezentare intermediară este independentă de arhitectură și oferă un cadru optim pentru analize, optimizări și transformări ulterioare aplicate de backend.

În mod particular, Clang dispune de suport pentru a recunoaște apeluri către **builtin-uri**, funcții speciale care nu sunt implementate în librării externe, ci direct în compilator. Aceste builtin-uri sunt mapate de Clang către instrucțiuni sau **intrinseci LLVM**, ce reprezintă abstracții de nivel jos care pot fi ulterior transpuse eficient în cod de asamblare specific platformei țintă.

În cadrul acestei lucrări, a fost introdus un nou builtin numit `__builtin_riscv_dot`, proiectat pentru a efectua un produs scalar (dot product) între doi vectori de tip `int64_t`. Scopul acestui builtin este dublu: (1) să permită exprimarea clară a operației de nivel înalt în codul sursă C și (2) să ofere o cale directă de optimizare în backend-ul RISC-V, unde poate fi convertit într-o secvență eficientă de instrucțiuni sau chiar într-o instrucțiune dedicată, dacă aceasta există în ISA (Instruction Set Architecture).

Prin maparea apelului din sursa C către un intrinsec dedicat în IR, Clang facilitează propagarea semanticii exacte a operației dorite până la nivelul generatorului de cod, unde aceasta poate fi opti-

mizată contextual în funcție de platforma RISC-V țintă.

5.1.1 Declarația builtin-ului în `BuiltinsRISCV.td`

Fișierul `BuiltinsRISCV.td` face parte din infrastructura Clang dedicată definirii extensiilor specifice pentru fiecare arhitectură de procesor suportată. Acesta descrie toate builtin-urile disponibile pentru target-ul `riscv`, incluzând semnăturile acestora și tipurile de date implicate.

Pentru a introduce un nou builtin specific produsului scalar între vectori de întregi pe 64 biți, a fost adăugată următoarea definiție:

```
// dot product (int64_t *a, int64_t *b, uint64_t n) -> int64_t
def dot : RISCVBuiltin<"int64_t(int64_t*, int64_t*, uint64_t)", "">;
```

Secvență de Cod 25: Declarația noului builtin în BuiltinsRISCV.td

Această declarație are o semnătură de forma:

```
int64_t builtin(int64_t* a, int64_t* b, uint64_t n);
```

și exprimă următoarele atribute importante:

- **Numele logic al builtin-ului** — în acest caz, `dot`, care va genera automat identificatorul C++ `RISCV::BI__builtin_riscv_dot` ce va putea fi recunoscut în `CGBuiltin.cpp`.
- **Tipul parametrilor și al valorii returnate** — exprimat textual în format C, indicând faptul că builtin-ul primește două adrese către vectori de `int64_t` și o dimensiune `uint64_t`, iar rezultatul este un scalar `int64_t`.

Alegerea acestor tipuri de date nu este arbitrară, ci rezultă din mai multe considerente tehnice și arhitecturale:

1. **Compatibilitate cu RISC-V 64-bit**: Procesorul de pe plăcuța BeagleV-Fire suportă în mod nativ operanți pe 64 de biți. Utilizarea tipului `int64_t` permite generarea eficientă de instrucțiuni `mul`, `add`, `ld`, `sd`, fără a necesita extensii de semn sau zero-padding.
2. **Eficiență în calcule matematice**: Un produs scalar implică o acumulare de produse, operații care, în cazul vectorilor mari, pot duce la overflow sau pierdere de precizie dacă sunt realizate pe 32 de biți. Folosirea `int64_t` permite prelucrări robuste și sigure în majoritatea scenariilor numerice.
3. **Semantica clară și scalabilă**: Tipul `uint64_t` pentru lungimea vectorului asigură posibilitatea de a lucra cu dimensiuni mari, fără limitarea la `int`, iar tipurile pointer oferă flexibilitate maximă în apelarea builtin-ului pe segmente arbitrare de memorie.

Această etapă este critică în procesul de extensie a Clang, întrucât fără declararea corectă în `.td`-file, builtin-ul nu va fi generat automat și nici recunoscut de parser-ul Clang. De asemenea, în procesul de build, Clang generează automat fișiere precum `BuiltinsRISCV.def`, în care noul builtin va fi înregistrat cu identificatorul unic, permițând astfel utilizarea ulterioară în codul sursă C/C++.

5.1.2 Tratarea builtin-ului în `CGBuiltin.cpp`

După declararea builtin-ului în `BuiltinsRISCV.td`, următorul pas este maparea acestuia către un **LLVM intrinsic** în timpul generării codului intermediar (LLVM IR). Această etapă este gestionată în fișierul `CGBuiltin.cpp`, în cadrul funcției specializate pentru target-ul RISC-V:

```
case RISCV::BI__builtin_riscv_dot: {
  Value *A = EmitScalarExpr(E->getArg(0));
  Value *B = EmitScalarExpr(E->getArg(1));
  Value *C = EmitScalarExpr(E->getArg(2));
  Function *Intrinsic = CGM.getIntrinsic(Intrinsic::riscv_dot);
  return Builder.CreateCall(Intrinsic, {A, B, C});
  break;
}
```

Secvență de Cod 26: Codul din CGBuiltin.cpp pentru __builtin_riscv_dot

5.1.2.1 Descrierea procesului:

- `E->getArg(i)` obține expresiile argumentelor din arborele de sintaxă abstractă (AST), unde `E` este o instanță a clasei `CallExpr`, corespunzătoare apelului `__builtin_riscv_dot()` din codul sursă C.
- `EmitScalarExpr()` generează codul LLVM IR corespunzător fiecărui operand, asigurând conversia din reprezentarea de nivel înalt a expresiei într-o valoare IR utilizabilă (`llvm::Value*`).
- `CGM.getIntrinsic()` este utilizat pentru a obține o referință la intrinsecul LLVM înregistrat anterior în `IntrinsicsRISCV.td`.
- `Builder.CreateCall()` generează efectiv instrucțiunea de apel către funcția intrinsecă în graficul IR.

5.1.2.2 Exemplu de utilizare în codul sursă C:

```
int64_t val = __builtin_riscv_dot(a, b, n);
```

Această linie este transformată de către Clang în instrucțiunea IR echivalentă:

```
%val = call i64 @llvm.riscv.dot(i64* %a, i64* %b, i64 %n)
```

Astfel, responsabilitatea `CGBuiltin.cpp` este de a face legătura între front-end-ul Clang (care recunoaște apelul builtin) și back-end-ul LLVM (care știe să optimizeze și să transforme intrinsecul în instrucțiuni specifice RISC-V).

Această etapă joacă un rol esențial în lanțul de compilare deoarece stabilește legătura directă între front-end-ul Clang (care parsează și analizează semantic codul sursă C) și backend-ul LLVM (care va optimiza și traduce codul intermediar în instrucțiuni de nivel inferior). Introducerea cazului nou în `EmitRISCVBuiltinExpr()` permite maparea controlată a apelului către o funcție intrinsecă specializată (în acest caz, `llvm.riscv.dot`) care va fi procesată ulterior de către backend într-un mod eficient și specific pe arhitectura dorită.

Această separare de responsabilități este caracteristică arhitecturii LLVM și oferă un cadru flexibil de extensie. De exemplu, dacă într-o etapă ulterioară se decide înlocuirea intrinsecului cu o implementare alternativă (precum o funcție externă optimizată manual sau cod inline specializat), acest lucru se poate face fără a afecta structura front-end-ului. Tot ce este necesar pentru definirea unui nou builtin este completarea unei declarații în `.td`, tratarea sa în `CGBuiltin.cpp`, și înregistrarea intrinsecului asociat, un flux modular, ușor de întreținut.

5.2 Backend — LLVM

LLVM (Low-Level Virtual Machine) reprezintă etapa din lanțul de compilare responsabilă cu transformarea codului intermediar (`LLVM IR`) într-un cod de nivel jos, adaptat arhitecturii țintă, în cazul nostru, **RISC-V 64**. Această transformare implică o serie de pași succesivi, care includ selecția instrucțiunilor, alocarea registrelor, optimizări locale și globale și, în final, generarea codului de asamblare.

În cadrul acestei lucrări, builtin-ul definit anterior în Clang (`__builtin_riscv_dot`) este propagat în backend printr-o arhitectură extensibilă de transformări. Totul începe cu definirea unui **intrinsec LLVM** în fișierul `IntrinsicsRISCV.td`, care descrie atât semnătura funcției, cât și efectele sale laterale (precum citirea din memorie). Acest intrinsec nu este implementat direct, ci este tratat ca o abstracție ce urmează a fi mapată ulterior într-o instrucțiune concretă.

Odată ajuns în etapa de selecție a instrucțiunilor (Instruction Selection), acest intrinsec este interceptat și transformat într-o **pseudo-instrucțiune**, o construcție intermediară care nu corespunde direct unei instrucțiuni din ISA, dar care va fi înlocuită ulterior cu o secvență echivalentă. Definirea pseudo-instrucțiunii are loc în `RISCVInstrInfo.td`, unde se stabilește legătura explicită între intrinsec și instrucțiunea `DOT`, care are o semnătură simplificată ce lucrează pe registre.

Această pseudo-instrucțiune este apoi expandată manual într-o buclă scalabilă în fișierul `RISCVExpandPseudoInsts.cpp`. Aici, codul adaugă explicit instrucțiuni pentru încărcarea valorilor din memorie, înmulțirea acestora, acumularea rezultatului și controlul fluxului într-o buclă. Expansiunea se realizează **post-RA** (după alocarea registrelor), iar implementarea evită utilizarea de registre virtuale, folosind în schimb un set de registre fizice temporare, sigure pentru acest context.

Pentru ca această expansiune să aibă loc, este nevoie ca pass-ul de tip `RISCVExpandPseudo` să fie activat în faza finală de generare a codului (`PreEmit`). Acest lucru se face în `RISCVTargetMachine.cpp`, unde pass-ul este adăugat explicit în configurația backend-ului.

Întregul flux de procesare este astfel construit modular: Clang generează codul IR cu apelul la intrinsec, LLVM îl recunoaște și îl transformă într-o pseudo-instrucțiune, iar backend-ul o expandează într-o buclă reală care execută produsul scalar. Această abordare permite testarea și validarea comportamentului fără a depinde de o instrucțiune hardware dedicată și oferă totodată flexibilitate pentru optimizări viitoare sau suport pentru alte extensii ISA.

5.2.1 Definirea intrinsecului în `IntrinsicsRISCV.td`

Fișierul `IntrinsicsRISCV.td` este responsabil cu înregistrarea tuturor intrinsecilor specifici arhitecturii RISC-V în cadrul sistemului LLVM. Aceste intrinsece reprezintă operații speciale, abstracte, care nu corespund direct unei instrucțiuni din limbajul de asamblare, dar care pot fi optimizate și transpuse eficient în cod nativ în fazele ulterioare ale compilării. În contextul acestei lucrări, a fost necesară adăugarea unui nou intrinsec, corespunzător builtin-ului `__builtin_riscv_dot`, care realizează un produs scalar între doi vectori întregi de 64 de biți.

Pentru aceasta, a fost introdusă următoarea declarație:

```
def int_riscv_dot : Intrinsic<
  [llvm_i64_ty],
  [llvm_ptr_ty, llvm_ptr_ty, llvm_i64_ty],
  [IntrReadMem, IntrArgMemOnly, IntrWillReturn,
   NoCapture<ArgIndex<0>>, NoCapture<ArgIndex<1>>];
```

Secvență de Cod 27: Definirea intrinsecului dot în IntrinsicsRISCV.td

Această declarație specifică o funcție intrinsecă numită `int_riscv_dot`, care returnează o valoa-

re de tip `int64_t` (`llvm_i64_ty`) și primește ca argumente doi pointeri către valori de tip `int64_t` (`llvm_ptr_ty`) și o valoare scalară de tip întreg pe 64 de biți. Semnătura este astfel aleasă pentru a corespunde apelului din codul sursă și pentru a permite o mapare directă a instrucțiunii de produs scalar asupra vectorilor din memorie.

Ultimul parametru al declarației îl reprezintă o listă de atribute, care oferă compilatorului informații suplimentare despre comportamentul intrinsecului:

- `IntrReadMem` — indică faptul că funcția citește din memorie;
- `IntrArgMemOnly` — specifică faptul că toate accesele la memorie sunt limitate doar la zona adresată prin argumente;
- `IntrWillReturn` — semnalează că funcția se termină întotdeauna (nu intră într-un ciclu infinit);
- `NoCapture<ArgIndex<0>>` și `NoCapture<ArgIndex<1>>` — garantează că pointerii primiți ca parametru nu sunt salvați în altă parte și nu vor fi folosiți în afara apelului.

Aceste informații sunt esențiale pentru optimizatorul LLVM, care le folosește pentru a decide dacă poate reordona instrucțiuni, elimina apeluri redundante sau aplica alte transformări agresive asupra IR-ului. În plus, această declarație asociază builtin-ul cu un identificator unic: `Intrinsic::riscv_dot`, care va fi folosit în etapa de selecție a instrucțiunilor (`Instruction Selection`) și care va facilita maparea către o pseudo-instrucțiune concretă în backend-ul RISC-V.

5.2.2 Selectarea intrinsecului în `RISCVISelDAGToDAG.cpp`

În etapa de selecție a instrucțiunilor (*Instruction Selection*), LLVM transformă nodurile generice din DAG-ul de operații intermediare (SelectionDAG) într-o reprezentare concretă, specifică arhitecturii țintă. Acest pas are loc în fișierul `RISCVISelDAGToDAG.cpp`, în cadrul metodei `Select()`, care este responsabilă cu maparea fiecărui nod intermediar pe una sau mai multe instrucțiuni RISC-V valide.

Pentru recunoașterea și tratarea intrinsecului `llvm.riscv.dot`, a fost adăugat un nou caz în secțiunea dedicată apelurilor la intrinsecuri cu efecte secundare (`INTRINSIC_W_CHAIN`). Acest tip de nod este folosit pentru intrinsecuri care implică acces la memorie și, implicit, necesită menținerea unei dependențe între operații (lanțul `chain`).

Fragmentul de cod adăugat are forma:

```
case ISD::INTRINSIC_W_CHAIN: {
  unsigned IntNo = Node->getConstantOperandVal(1);
  switch (IntNo) {
```

```
case Intrinsic::riscv_dot: {
  SDValue PtrA = Node->getOperand(2);
  SDValue PtrB = Node->getOperand(3);
  SDValue Count = Node->getOperand(4);

  MachineSDNode *Res = CurDAG->getMachineNode(
    RISCV::DOT, DL, {Node->getValueType(0), MVT::Other},
    {PtrA, PtrB, Count});
  ReplaceNode(Node, Res);
  return;
}
```

Secvență de Cod 28: Selectarea intrinsecului în RISCVISelDAGToDAG.cpp

În acest bloc, se extrage identificatorul intrinsec din operandul constant al nodului (`getConstantOperandVal(1)`). Dacă valoarea corespunde cu `Intrinsic::riscv_dot`, se preiau cei trei operanzi ai apelului: doi pointeri (`PtrA`, `PtrB`) și o valoare scalară `Count`, care reprezintă dimensiunea vectorilor.

Acești operanzi sunt transmiși către un nou nod de tip `MachineSDNode`, generat de metoda `CurDAG->getMachineNode()`, care construiește un nod asociat pseudo-instrucțiunii `DOT`. Acesta este identificatorul simbolic al unei instrucțiuni definite ulterior în fișierul `RISCVInstrInfo.td`, ce urmează a fi expandată într-o buclă explicită în codul de asamblare.

Se specifică două tipuri de return: valoarea rezultată a produsului scalar (tipul original al nodului IR) și un nod `MVT::Other` care păstrează informațiile legate de efectele asupra lanțului de execuție (*memory chain*).

În final, apelul la `ReplaceNode()` înlocuiește vechiul nod IR cu nodul nou generat, ceea ce asigură că backend-ul va opera de aici înainte asupra unui cod optimizat specific RISC-V, în locul unei reprezentări generice de intrinsec.

Această etapă este crucială, deoarece marchează tranziția de la IR la codul specific targetului. Prin folosirea unei pseudo-instrucțiuni intermediare, se oferă posibilitatea de a amâna decizia finală asupra implementării exacte până la momentul generării codului final, unde pseudo-op-ul va fi expandat într-o secvență scalabilă de instrucțiuni RISC-V. Astfel, flexibilitatea este menținută, iar optimizările contextuale pot fi aplicate ulterior.

5.2.3 Definirea pseudo-instrucțiunii în `RISCVInstrInfo.td`

După ce intrinsec-ul a fost selectat în DAG ca un nod de tip `DOT`, este necesară definirea concretă a acestuia în fișierul `RISCVInstrInfo.td`, unde sunt descrise toate instrucțiunile suportate de backend-ul RISC-V. Întrucât operația de produs scalar implementată nu corespunde unei instrucțiuni native din setul de instrucțiuni RISC-V, s-a optat pentru crearea unei **pseudo-instrucțiuni**.

Pseudo-instrucțiunile sunt folosite în LLVM ca un mecanism intermediar: ele nu sunt emise direct în codul final, ci sunt expandate, într-o fază ulterioară, într-o secvență completă de instrucțiuni reale care realizează comportamentul dorit. Astfel, se păstrează separarea logică între semantică și implementare, facilitând menținerea și extinderea codului.

Definiția instrucțiunii `DOT` este următoarea:

```
def DOT : Pseudo<
  (outs GPR:$rd),
  (ins  GPR:$rs1, GPR:$rs2, GPR:$rs3),
  [(set GPR:$rd,
   (int_riscv_dot GPR:$rs1, GPR:$rs2, GPR:$rs3))]>
{
  let mayLoad = 1;
  let hasSideEffects = 1;
  let isCodeGenOnly = 1;
  let isPseudo = 1;
  let Size = 100;
}
```

Secvență de Cod 29: Pseudo-instrucțiunea DOT în RISCVInstrInfo.td

Această definiție specifică următoarele:

- Instrucțiunea are un operand de ieșire (`rd`) și trei operanzi de intrare (`rs1`, `rs2`, `rs3`), corespunzători celor doi vectori și dimensiunii acestora.
- Atributele semnalizează că:
  - `mayLoad = 1` — instrucțiunea accesează memoria (încarcă valori din vectori);
  - `hasSideEffects = 1` — comportamentul său nu este pur, deci nu poate fi eliminată sau reordonată arbitrar;
  - `isCodeGenOnly = 1` — nu este disponibilă la nivel de IR, ci doar în backend;
  - `isPseudo = 1` — instrucțiunea este un placeholder și va fi înlocuită ulterior de o secvență concretă;

– `Size = 100` — o estimare a dimensiunii codului rezultat (pentru scopuri de optimizare și plasare).

Această pseudo-instrucțiune acționează ca un „reprezentant semantic” pentru operația de produs scalar și permite o tranziție controlată între nivelul abstract al instrucțiunilor IR și implementarea concretă în cod de asamblare.

Pentru ca această mapare să aibă loc automat în timpul selecției instrucțiunilor, a fost adăugat și un pattern corespunzător:

```
def : Pat<
  (int_riscv_dot GPR:$rs1, GPR:$rs2, GPR:$rs3),
  (DOT           GPR:$rs1, GPR:$rs2, GPR:$rs3)
>;
```

Secvență de Cod 30: Pattern de mapare a intrinsec-ului în Pseudo

Acest pattern definește o regulă de transformare pentru LLVM: atunci când este întâlnit un apel la intrinsecul `int_riscv_dot` cu trei registre ca argumente, acesta trebuie înlocuit cu pseudo-instrucțiunea `DOT`, cu aceiași operanzi. Acest proces se desfășoară în cadrul `Instruction Selection TableGen` și asigură consistența între instrucțiunile IR și nodurile MachineInstr generate ulterior.

Prin această strategie modulară, codul rămâne extensibil și clar separat: semanticile operațiilor sunt exprimate în instrucțiuni pseudo, iar implementarea concretă este delegată unei etape ulterioare; în cazul de față, pasului de expansiune descris în `RISCVExpandPseudoInsts.cpp`.

5.2.4 Expandarea pseudo-instrucțiunii în `RISCVExpandPseudoInsts.cpp`

După ce pseudo-instrucțiunea `DOT` este selectată de ISel și emisă în codul intermediar, trebuie să fie expandată într-o secvență concretă de instrucțiuni RISC-V. Această transformare are loc în etapa post-RA (Post Register Allocation), în cadrul fișierului `RISCVExpandPseudoInsts.cpp`. Aici, instrucțiunea pseudo este înlocuită cu o buclă explicită care efectuează un produs scalar între doi vectori de tip `int64_t`.

Pentru a gestiona această expandare, s-a introdus o funcție dedicată:

```
bool expandDot(MachineBasicBlock &MBB,
MachineBasicBlock::iterator MBBI,
MachineBasicBlock::iterator &NextMBBI);
```

Secvență de Cod 31: Prototipul funcției pentru expandarea pseudo-instrucțiunii DOT

Această funcție este responsabilă cu înlocuirea pseudo-instrucțiunii `DOT` cu o secvență concretă de instrucțiuni reale care implementează produsul scalar.

Pentru a integra această funcționalitate în infrastructura LLVM, este necesar să conectăm noua funcție cu mecanismul de traversare al instrucțiunilor. Acest lucru se realizează prin adăugarea unui caz în funcția `expandMI()`, care este punctul de intrare pentru expandarea tuturor pseudo-instrucțiunilor în timpul trecerii *RISCVExpandPseudoPass*.

```
case RISCV::DOT:
return expandDot(MBB, MBBI, NextMBBI);
```

Secvență de Cod 32: Apelarea funcției de expandare DOT în expandMI

Funcția `expandMI()` parcurge toate instrucțiunile din fiecare bloc de bază și decide, pe baza opcode-ului, care dintre pseudo-instrucțiuni trebuie expandate. Adăugând un caz pentru `RISCV::DOT`, ne asigurăm că orice apariție a acestei pseudo-instrucțiuni va fi tratată corect și transformată în cod concret înainte de generarea finală a codului de asamblare.

Această integrare este esențială pentru a insera logică personalizată în faza finală a backend-ului, unde se presupune că toată alocarea de registre a fost deja făcută și toate optimizările de nivel înalt s-au încheiat. De aceea, în această etapă se folosesc exclusiv registre fizice și nu se mai introduce niciun nou registru virtual.

Funcția `expandDot()` are următorul conținut:

```
bool RISCVExpandPseudo::expandDot(MachineBasicBlock &MBB,
MachineBasicBlock::iterator MBBI,
MachineBasicBlock::iterator &NextMBBI) {
MachineInstr &MI = *MBBI;
const DebugLoc DL = MI.getDebugLoc();

if (MI.getNumOperands() != 4 || !MI.getOperand(0).isReg() ||
!MI.getOperand(1).isReg() || !MI.getOperand(2).isReg() ||
!MI.getOperand(3).isReg())
return false;

Register Rd = MI.getOperand(0).getReg();
Register SrcA = MI.getOperand(1).getReg();
Register SrcB = MI.getOperand(2).getReg();
Register SrcN = MI.getOperand(3).getReg();

const Register ACC = RISCV::X13;
const Register PTR_A = RISCV::X5;
const Register PTR_B = RISCV::X6;
const Register COUNT = RISCV::X7;
const Register VAL_A = RISCV::X28;
const Register VAL_B = RISCV::X29;

MachineFunction &MF = *MBB.getParent();
const RISCVInstrInfo *TII = MF.getSubtarget<RISCVSubtarget>().getInstrInfo();
```

```
26
27 auto *LoopMBB = MF.CreateMachineBasicBlock();
28 auto *ExitMBB = MF.CreateMachineBasicBlock();
29 MF.insert(std::next(MachineFunction::iterator(&MBB)), LoopMBB);
30 MF.insert(std::next(MachineFunction::iterator(LoopMBB)), ExitMBB);
31
32 ExitMBB->splice(ExitMBB->begin(), &MBB, std::next(MBBI), MBB.end());
33 ExitMBB->transferSuccessorsAndUpdatePHIs(&MBB);
34
35 while (!MBB.succ_empty())
36 MBB.removeSuccessor(MBB.succ_begin());
37 MBB.addSuccessor(LoopMBB);
38 LoopMBB->addSuccessor(LoopMBB);
39 LoopMBB->addSuccessor(ExitMBB);
40
41 BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), ACC)
42 .addReg(RISCV::X0)
43 .addImm(0);
44 BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), PTR_A)
45 .addReg(SrcA)
46 .addImm(0);
47 BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), PTR_B)
48 .addReg(SrcB)
49 .addImm(0);
50 BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), COUNT)
51 .addReg(SrcN)
52 .addImm(0);
53 BuildMI(MBB, std::next(MBBI), DL, TII->get(RISCV::JAL))
54 .addReg(RISCV::X0, RegState::Define)
55 .addMBB(LoopMBB);
56
57 auto EndIt = LoopMBB->end();
58 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::LD), VAL_A)
59 .addReg(PTR_A)
60 .addImm(0);
61 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::LD), VAL_B)
62 .addReg(PTR_B)
63 .addImm(0);
64 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::MUL), VAL_B)
65 .addReg(VAL_A)
66 .addReg(VAL_B);
67 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::ADD), ACC)
68 .addReg(ACC)
69 .addReg(VAL_B);
70 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::ADDI), PTR_A)
71 .addReg(PTR_A)
72 .addImm(8);
73 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::ADDI), PTR_B)
```

```
74 .addReg(PTR_B)
75 .addImm(8);
76 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::ADDI), COUNT)
77 .addReg(COUNT)
78 .addImm(-1);
79 BuildMI(*LoopMBB, EndIt, DL, TII->get(RISCV::BNE))
80 .addReg(COUNT)
81 .addReg(RISCV::X0)
82 .addMBB(LoopMBB);
83
84 BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::ADDI), Rd)
85 .addReg(ACC)
86 .addImm(0);
87
88 for (Register R : {ACC, PTR_A, PTR_B, COUNT})
89 if (!LoopMBB->isLiveIn(R))
90 LoopMBB->addLiveIn(R);
91 if (!ExitMBB->isLiveIn(ACC))
92 ExitMBB->addLiveIn(ACC);
93
94 MF.RenumberBlocks();
95 NextMBBI = MBB.erase(MBBI);
96 return true;
97 }
```

Secvență de Cod 33: Implementarea completă a expandDot()

5.2.4.1 Explicație detaliată a funcției `expandDot()`

Funcția `expandDot()` este responsabilă pentru înlocuirea pseudo-instrucțiunii `DOT` cu o buclă explicită care realizează produsul scalar între doi vectori de 64 de biți. Această transformare are loc în etapa *post-RA* (după alocarea registrelor), motiv pentru care sunt utilizate exclusiv registre fizice, sigure în această fază. Astfel, se evită interferența cu analiza în formă statică unică (SSA) și nu se generează registre virtuale noi.

La începutul funcției este introdus un mesaj de debug, folosind:

```
1 errs() << ">>> expandDot() called for DOT pseudo\n";
```

Acesta servește pentru depanare și confirmă că funcția este apelată în timpul rulării `llc`, în momentul expandării pseudo-instrucțiunii.

Instrucțiunea `DOT` este obținută ca referință din poziția actuală a iteratorului (`MBBI`), iar locația sa în codul sursă (dacă este disponibilă) este salvată într-un obiect de tip `DebugLoc`, pentru eventualele propagări de informații de depanare.

Primul pas logic constă în validarea operandului instrucțiunii. Se verifică dacă DOT are exact patru operanzi, toți de tip registru: rezultatul (Rd), adresele vectorilor a și b, respectiv lungimea n. Dacă această condiție nu este îndeplinită, funcția se oprește cu `return false`, permițând sistemului de verificare LLVM să detecteze inconsistențele.

După validare, sunt extrase registrele sursă și registrul destinație. Apoi, se alocă explicit registre fizice pentru a fi folosite în timpul executării buclei. Se folosesc: *X13* pentru acumularea rezultatului (ACC), *X5* și *X6* pentru parcurgerea vectorilor (PTR_A, PTR_B), *X7* pentru contorul COUNT, iar *X28* și *X29* pentru elementele încărcate și produsul parțial (VAL_A, VAL_B). Toate aceste registre sunt *caller-saved*, deci sigure pentru utilizare după alocarea finală a registrelor.

Se creează apoi două blocuri de cod noi: LoopMBB, care va conține corpul efectiv al buclei, și ExitMBB, care va conține instrucțiunile care urmau după DOT în blocul curent. Aceste blocuri sunt inserate imediat după blocul original, pentru a păstra o topologie liniară în MachineFunction.

Pentru a păstra ordinea logică a execuției, toate instrucțiunile care se aflau după DOT în blocul curent sunt mutate în ExitMBB, cu tot cu succesorii săi. În acest fel, fluxul de control va continua corect după încheierea buclei.

Control-flow-ul este reconfigurat astfel: blocul original (MBB) are acum ca unic succesor blocul LoopMBB. La rândul său, LoopMBB are doi succesori: către el însuși (pentru iterațiile ulterioare ale buclei) și către ExitMBB (pentru cazul în care bucla se încheie). Această structură reflectă o buclă clasică cu condiție de continuare.

Înainte de intrarea în buclă, se generează codul de inițializare: ACC este setat la zero, PTR_A și PTR_B sunt inițializate cu adresele de start ale vectorilor, iar COUNT primește valoarea lui n. După aceste instrucțiuni, se emite un salt necondiționat (JAL) către începutul buclei.

Corpul buclei conține secvența esențială de calcul: se încarcă valorile a[i] și b[i] folosind instrucțiunea LD, se efectuează înmulțirea (MUL) și se adaugă rezultatul în registrul ACC. După fiecare iterație, pointerii vectorilor sunt avansați cu 8 octeți (echivalentul unui int64_t), iar COUNT este decrementat. Dacă valoarea acestuia este diferită de zero, execuția se întoarce la începutul buclei printr-o instrucțiune de ramificare condiționată (BNE).

La ieșirea din buclă, în blocul ExitMBB, rezultatul acumulat din registrul ACC este copiat în registrul de destinație Rd, finalizând astfel comportamentul funcțional echivalent cu instrucțiunea pseudo inițială.

Pentru a asigura corectitudinea propagării valorilor între blocuri, registrele fizice utilizate în buclă sunt marcate ca fiind *live-in* pentru LoopMBB și ExitMBB. Această marcaj este esențial pentru analiza de flux de date și pentru corectitudinea codului generat.

În cele din urmă, se renumerotează blocurile de bază pentru a menține ordinea corectă în cadrul

funcției, pseudo-instrucțiunea `DOT` este ștearsă, iar iteratorul este actualizat astfel încât prelucrarea în cadrul pasului `RISCVExpandPseudo` să continue cu instrucțiunea corectă.

Această funcție ilustrează clar mecanismele interne ale LLVM pentru manipularea și expansiunea instrucțiunilor personalizate. Ea arată cum se poate trece de la o abstracțiune la nivel înalt (pseudo-instrucțiune definită printr-un `builtin`) la o secvență concretă de instrucțiuni de asamblare, integrate coerent în cadrul unui pipeline real de compilare pentru arhitectura RISC-V. Utilizarea atentă a blocurilor, instrucțiunilor și registrelor fizice permite o integrare robustă în backend-ul LLVM, fără a afecta negativ restul procesului de generare cod.

5.2.5 Activarea pasului în `RISCVTargetMachine.cpp`

Pentru ca funcția `expandDot()` să fie efectiv utilizată în procesul de compilare, este necesar ca pasul de expandare a pseudo-instrucțiunilor să fie introdus explicit în pipeline-ul de transformări LLVM. Acest lucru se realizează în fișierul `RISCVTargetMachine.cpp`, în cadrul metodei `addPreEmitPass()`.

Această metodă este apelată de infrastructura LLVM atunci când este construit lanțul final de pași de transformare asupra codului mașină generat pentru arhitectura țintă. După cum sugerează și numele său, `addPreEmitPass()` este responsabilă pentru inserarea ultimelor transformări care se aplică asupra codului mașină, înainte ca acesta să fie emis (adică scris efectiv în format de asamblare sau binar). Aceste transformări sunt denumite în mod convențional pași de tip PreEmit.

Pentru a introduce pasul nostru personalizat, cel care expandează pseudo-instrucțiunea `DOT` într-o buclă scalară completă, trebuie să adăugăm apelul următor:

```
addPass(createRISCVExpandPseudoPass());
```

Funcția `createRISCVExpandPseudoPass()` returnează o instanță a unui `MachineFunctionPass` care conține logica pentru identificarea și înlocuirea pseudo-instrucțiunilor, inclusiv apelul către `expandDot()`. Prin înregistrarea acestui pas în `addPreEmitPass()`, ne asigurăm că LLVM îl va executa asupra fiecărei funcții din programul compilat, chiar înainte ca codul de asamblare să fie generat.

Această poziționare este esențială deoarece:

- Transformarea pseudo-instrucțiunii `DOT` presupune folosirea explicită a unor registre fizice (ex. `X13`, `X28`), care pot fi alocate în siguranță doar după ce procesul de alocare a registrelor virtuale a fost finalizat (etapa de *Register Allocation*).
- În această fază a compilării, codul este deja transformat într-o formă apropiată de instrucțiunile ISA reale, iar modificările operate nu mai interferează cu analiza SSA sau cu optimizările aplicate anterior la nivel intermediar (IR).

- Interferențele potențiale cu alocarea registrelor sunt evitate complet prin design-ul actual, ceea ce conferă transformării un grad înalt de siguranță în integrarea cu backend-ul LLVM.

Activarea pasului de expandare în `RISCVTargetMachine.cpp` este un pas critic pentru ca logicile implementate anterior în `RISCVExpandPseudoInsts.cpp` să fie efectiv aplicate în timpul generării codului. Această integrare face ca pseudo-instrucțiunea definită de utilizator (`DOT`) să se comporte ca o instrucțiune nativă în cadrul procesului de compilare, chiar dacă ea este doar o abstracție expandată în cod de asamblare real.

5.3 Fluxul complet de transformare al builtin-ului `__builtin_riscv_dot`

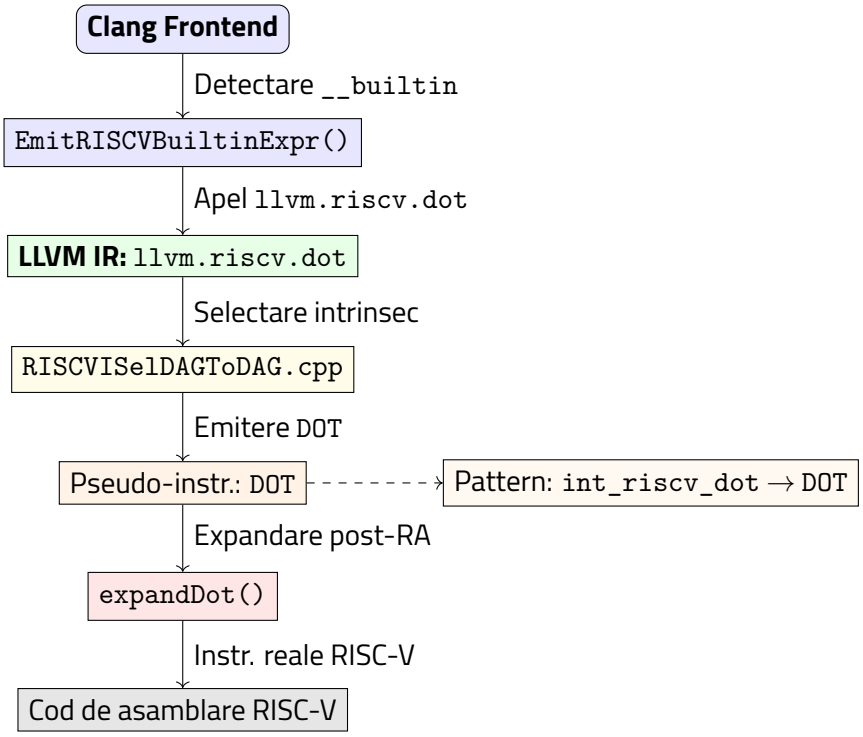

Figura 6: Fluxul complet de transformare pentru `__builtin_riscv_dot`.

Figura 6 ilustrează succesiunea de pași prin care builtin-ul `__builtin_riscv_dot` este transformat, din momentul în care este detectat în front-end-ul Clang și până la generarea codului de asamblare concret în backend-ul LLVM pentru arhitectura RISC-V. Procesul începe cu construirea apelului către un intrinsec LLVM în `CGBuiltin.cpp`, urmată de definirea acestuia în `IntrinsicsRISCV.td`. Acest intrinsec este apoi recunoscut și mapat la o pseudo-instrucțiune `DOT` în `RISCVISelDAGToDAG.cpp`, care este ulterior expandată într-o buclă explicită în `RISCVExpandPseudoInsts.cpp`. În final, pasul de expandare este activat în `RISCVTargetMachine.cpp`, asigurând integrarea completă în fluxul de generare cod.

6 Testarea și validarea builtin-ului

După implementarea completă a extensiei builtin-ului `__builtin_riscv_dot`, următorul pas esențial a fost validarea funcționării acestuia într-un scenariu real. Scopul testării este de a confirma faptul că fluxul de compilare, de la recunoașterea apelului în Clang, până la generarea și rularea codului RISC-V corespunzător, este complet funcțional și produce rezultatele corecte.

Pentru aceasta, a fost dezvoltat un program simplu scris în C, care utilizează direct builtin-ul nou introdus pentru a calcula produsul scalar dintre doi vectori de 64 de biți. Testul este conceput pentru a evidenția comportamentul integral al lanțului de compilare: front-end (Clang), generarea IR și instrucțiunilor LLVM, selecția pseudo-instrucțiunii `DOT`, expandarea acesteia în backend, generarea codului de asamblare concret, precum și rularea executabilului rezultat pe o arhitectură țintă.

Pentru execuție s-a folosit simulatorul `Spike`, parte din ecosistemul RISC-V, care permite rularea și testarea codului ELF generat într-un mediu complet controlat. Astfel, putem obține atât rezultate funcționale, cât și verificări structurale asupra codului generat.

În cele ce urmează este prezentat în detaliu modul în care a fost realizat testul, comenzile folosite, precum și extrase relevante din codul rezultat care dovedesc corectitudinea implementării.

6.0.1 Codul sursă pentru testare

```
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>

// Builtin call
extern int64_t __builtin_riscv_dot(int64_t* a, int64_t* b, uint64_t n);

int main(void) {
    int64_t a[4] = { 1, 2, 3, 4 };
    int64_t b[4] = { 5, 6, 7, 8 };
    uint64_t n = 4;

    int64_t result = __builtin_riscv_dot(a, b, n);

    printf("dot(a,b,n=%" PRIu64 ") = %ld\n", n, result);

    if (result != 70) {
        fprintf(stderr,
            "Eroare: rezultatul ar fi trebuit sa fie 70!\n");
        return 1;
    }

```

```
    puts("Succes: rezultatul este corect!");
    return 0;
}
```

Secvență de Cod 34: Testarea builtin-ului __builtin_riscv_dot

6.0.2 Compilare și rulare

Testul a fost compilat direct în Visual Studio 2022 folosind Clang, cu următorii parametri de compilare personalizați:

```
-O0 --target=riscv64-unknown-elf
--gcc-toolchain="C:/msys64/mingw64"
--sysroot="C:/msys64/mingw64/riscv64-unknown-elf"
-mllvm -fast-isel=false
-mllvm -debug-only=isel,machineinstrs
-mllvm -print-before=RISCVExpandPseudoPass
-mllvm -print-after=RISCVExpandPseudoPass
-mllvm -print-after=Select
-mllvm -verify-machineinstrs
S:\WIP\Test\test_output.c -o S:\WIP\Outputs\test.elf
```

Secvență de Cod 35: Argumente de compilare pentru Clang

Aceste opțiuni activează și logarea intermediară a pașilor de selecție și expandare a instrucțiunilor, fiind esențiale pentru a verifica dacă pseudo-instrucțiunea `DOT` a fost detectată și expandată corect în backend.

Programul rezultat a fost executat pe simulatorul `Spike`, folosind:

```
spike pk test.elf
```

Secvență de Cod 36: Execuția binarului pe simulator

Rezultatul obținut a fost cel așteptat:

```
dot(a,b,n=4) = 70
Succes: rezultatul este corect!
```

6.0.3 Verificarea codului generat

Pentru a confirma că pseudo-instrucțiunea `DOT` a fost într-adevăr expandată într-o buclă scalară la nivel de cod mașină, s-a efectuat o dezasamblare completă a binarului generat, folosind comanda:

```
riscv64-unknown-elf-objdump -d S:\WIP\Outputs\test.elf > S:\WIP\Outputs\test_dot.
    objdump
```

Secvență de Cod 37: Generarea fișierului .objdump

Alternativ, în Visual Studio 2022, se poate folosi instrumentul `llvm-objdump` cu argumentul `-d` pentru a vizualiza codul de asamblare.

În urma dezasamblării, secțiunea `main` arată explicit codul generat pentru bucla de produs scalar, ceea ce confirmă funcționarea corectă a expandării:

```
00000000000101c4 <main>:
1021c:   0002be03            ld  t3,0(t0)
10220:   00033e83            ld  t4,0(t1)
10224:   03de0eb3            mul t4,t3,t4
10228:   96f6                add a3,a3,t4
1022a:   02a1                addi  t0,t0,8
1022c:   0321                addi  t1,t1,8
1022e:   13fd                addi  t2,t2,-1
10230:   fe0396e3            bnez  t2,1021c <main+0x58>
```

Secvență de Cod 38: Fragment relevant din main în .objdump

Aceste instrucțiuni reprezintă cu exactitate bucla scalară generată în pasul `expandDot()`, în care se încarcă elementele vectorilor din memorie folosind instrucțiuni `LD`, urmate de operația de înmulțire cu `MUL` și de acumulare progresivă în registrul `a3` (care corespunde acumulatorului `ACC`). După fiecare iterație, adresele pointerilor sunt incrementate cu 8 octeți, corespunzător dimensiunii unui element de tip `int64_t`, prin instrucțiuni `ADDI`. În final, contorul de elemente este decrementat, iar instrucțiunea condițională `BNE` (branch if not equal) controlează întoarcerea în buclă, implementând mecanismul de iterație. Această secvență confirmă faptul că pseudo-instrucțiunea `DOT` a fost expandată corect în instrucțiuni native RISC-V, fără a lăsa urme de cod abstract și asigurând o execuție eficientă a operației de produs scalar.

6.1 Testarea pe hardware-ul real BeagleV-Fire

După ce implementarea a fost verificată cu succes în simulatorul Spike, testarea finală a presupus rularea codului pe hardware-ul fizic BeagleV-Fire. Pentru aceasta, a fost necesară reconstruirea completă a toolchain-ului `LLVM` în mediul `WSL`, folosind următoarea comandă pentru compilarea `clang` cu suport RISC-V:

```
$ cd llvm-project
$ mkdir -p build && cd build
$ cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS="clang" \
  -DLLVM_TARGETS_TO_BUILD="RISCV" \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=$HOME/llvm-riscv-install
$ ninja clang
```

```
$ ninja install
```

Secvență de Cod 39: Compilarea LLVM/Clang cu suport RISC-V

După instalare, fișierul de testare a fost compilat cu toolchain-ul construit, utilizând arhitectura corespunzătoare plăcii:

```
$RISCV/bin/clang -O0 -target riscv64-unknown-linux-gnu \
  -march=rv64gc -mabi=lp64d test_output.c -o test_output
```

Secvență de Cod 40: Compilarea codului pentru BeagleV-Fire

Fișierul rezultat `test_output` a fost transferat către placă prin rețea folosind `scp`:

```
scp test_output beagle@192.168.7.2:~/
```

Secvență de Cod 41: Transferul fișierului pe placă

Pe placă, fișierului i s-au acordat permisiuni de execuție, iar executabilul a fost rulat:

```
chmod +x test_output
./test_output
```

Secvență de Cod 42: Rularea pe BeagleV-Fire

Rezultatul obținut a fost identic cu cel din simulare, confirmând astfel că pseudo-instrucțiunea `DOT` este expandată și executată corect nu doar în emulator, ci și pe un sistem RISC-V real. Acest pas validează complet integrarea instrucțiunii în backend-ul LLVM și funcționalitatea acesteia în condiții reale de execuție.

6.2 Testarea și validarea performanței

Pentru a analiza performanța instrucțiunii pseudo `DOT`, am implementat un program C ce compară apelul către `__builtin_riscv_dot` cu o variantă scrisă în limbajul C: una nativă ce folosește `int64_t`. Fiecare test a fost rulat de 10^6 ori pentru valori succesive ale lui N (dimensiunea vectorilor): $N \in \{4, 8, 16, ..., 1024\}$.

Codul complet al programului de testare este prezentat mai jos:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>
#include <time.h>

#define REPS 1000000

__attribute__((noinline))
static int64_t dot_builtin(const int64_t *a,
```

```
                           const int64_t *b,
                           size_t n) {
    asm volatile("" ::: "memory");
    int64_t v = __builtin_riscv_dot((int64_t*)a, (int64_t*)b, n);
    asm volatile("" ::: "memory");
    return v;
}

static int64_t dot_native(const int64_t *a,
                          const int64_t *b,
                          size_t n) {
    int64_t acc = 0;
    for (size_t i = 0; i < n; i++)
        acc += a[i] * b[i];
    return acc;
}

static double measure_builtin(const int64_t *a,
                              const int64_t *b,
                              size_t n,
                              int64_t *out_val) {
    struct timespec t0, t1;
    clock_gettime(CLOCK_MONOTONIC, &t0);
    int64_t v = 0;
    for (int i = 0; i < REPS; i++)
        v = dot_builtin(a, b, n);
    clock_gettime(CLOCK_MONOTONIC, &t1);
    *out_val = v;
    return (t1.tv_sec  - t0.tv_sec)
         + (t1.tv_nsec - t0.tv_nsec) * 1e-9;
}

static double measure_native(const int64_t *a,
                             const int64_t *b,
                             size_t n,
                             int64_t *out_val) {
    struct timespec t0, t1;
    clock_gettime(CLOCK_MONOTONIC, &t0);
    int64_t v = 0;
    for (int i = 0; i < REPS; i++)
        v = dot_native(a, b, n);
    clock_gettime(CLOCK_MONOTONIC, &t1);
    *out_val = v;
    return (t1.tv_sec  - t0.tv_sec)
         + (t1.tv_nsec - t0.tv_nsec) * 1e-9;
}
```

Secvență de Cod 43: Codul de testare folosit pentru compararea performanțelor

Compilarea s-a făcut pentru fiecare nivel de optimizare `-O0`, `-O1`, `-O2` și `-O3` folosind Clang RISC-V, iar execuția a avut loc direct pe placa BeagleV-Fire, cu comanda:

```
$RISCV/bin/clang -O2 -target riscv64-unknown-linux-gnu \
  -march=rv64gc -mabi=lp64d compare_perf.c -o compare_perf_O2
scp compare_perf_O2 beagle@192.168.7.2:~/
ssh beagle@192.168.7.2
chmod +x compare_perf_O2 && ./compare_perf_O2
```

6.2.1 Rezultate obținute

Tabel 4: Timp mediu pe apel (μs), pentru 10^6 apeluri, în funcție de N și nivelul de optimizare

N	-O0		-O1		-O2		-O3	
	builtin	native	builtin	native	builtin	native	builtin	native
4	0.172	0.325	0.106	0.087	0.106	0.087	0.106	0.087
8	0.252	0.554	0.180	0.154	0.180	0.154	0.180	0.154
16	0.423	1.010	0.328	0.289	0.327	0.289	0.327	0.289
32	0.745	1.922	0.623	0.557	0.622	0.557	0.622	0.557
64	1.389	3.747	1.214	1.095	1.213	1.093	1.212	1.093
128	2.677	7.396	2.394	2.171	2.393	2.167	2.394	2.167
256	5.253	14.695	4.757	4.314	4.755	4.313	4.755	4.313
512	10.407	29.297	9.481	8.609	9.480	8.608	9.479	8.611
1024	20.731	58.556	18.942	17.211	18.940	17.210	18.941	17.209

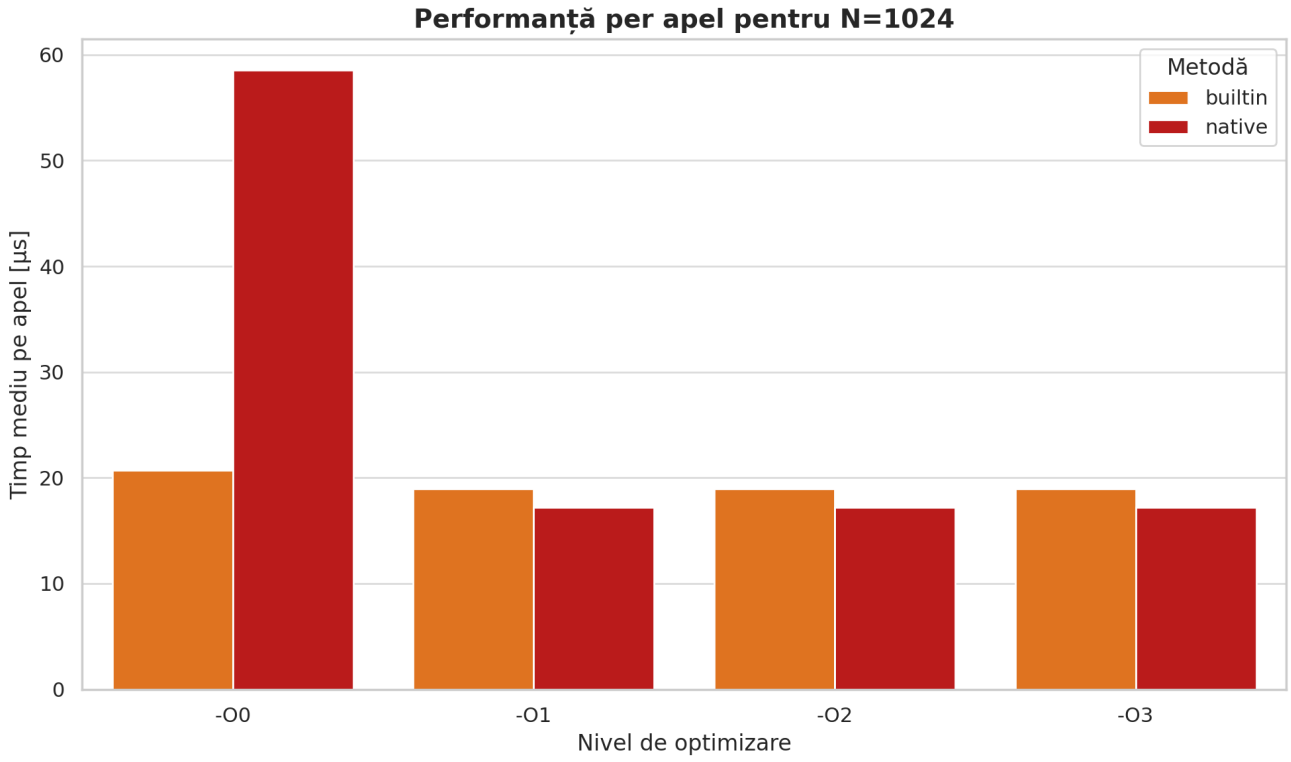

Figura 7: Performanța per apel pentru N=1024

6.2.2 Analiza rezultatelor

În analiza performanței builtin-ului `__builtin_riscv_dot` în forma sa inițială, se remarcă imediat un comportament interesant și aparent contraintuitiv: la nivelul de optimizare `-O0`, codul generat cu ajutorul pseudo-instrucțiunii specializate este considerabil mai rapid decât varianta clasică scrisă în limbajul C, în forma sa nativă (`int64_t`).

La `-O0`, compilatorul nu aplică nicio transformare complexă asupra codului. În această configurație, codul C este tradus aproape literal în reprezentarea intermediară IR, fără eliminarea apelurilor de funcții, fără desfășurarea buclelor, fără propagare de constante sau reorganizare de flux. Asta înseamnă că o funcție scrisă în C, care conține o buclă de tipul:

`for (i = 0; i < N; ++i) sum += A[i] * B[i];`

este transpusă într-o serie de instrucțiuni care păstrează exact aceleași controale de flux și apeluri, cu overhead semnificativ cauzat de comparații, incrementări și salturi repetate.

Pe de altă parte, builtin-ul `DOT` este expandat într-un cod de tip buclă scalara direct în backend, într-o formă deja compactizată și optimizată manual. Această expansiune generează cod mașină simplu, liniar și eficient, lipsit de apeluri externe sau overhead semantic. La `-O0`, acest avantaj devine foarte vizibil: codul generat pentru builtin este curat, scurt, și constă într-o buclă cu instrucțiuni `ld`, `mul`, `add`, `addi` și `bnez`, executate în mod secvențial și fără complexitate inutilă. Astfel, pentru dimensiuni mici ale vectorului (ex. $N = 4$ sau $N = 8$), timpul de execuție al builtin-ului este de aproape două ori mai mic decât al versiunii C.

Însă pe măsură ce nivelul de optimizare crește (de la `-O1` la `-O3`), situația se inversează gradual. Varianta scrisă în C începe să beneficieze din plin de arsenalul complet de transformări LLVM, ceea ce duce la o reducere dramatică a latențelor. Compilatorul identifică tiparele buclei, detectează faptul că nu există dependențe între iterații și că dimensiunile vectorilor sunt cunoscute la compilare. Astfel, este capabil să aplice tehnici de unrolling (desfășurarea completă sau parțială a buclei), să elimine calculele redundante și să creeze o versiune a buclei care este executată într-o formă mult mai eficientă din punct de vedere al execuției pe pipeline-ul procesorului.

De exemplu, pentru un cod precum:

```
    int64_t dot_native(const int64_t *A, const int64_t *B, int N) {
  int64_t sum = 0;
  for (int i = 0; i < N; ++i) {
    sum += A[i] * B[i];
  }
  return sum;
}
```

Secvență de Cod 44: Implementarea nativă a dot product

Compilatorul, la -O3, îl poate transforma într-o variantă desfășurată complet pentru dimensiuni mici (ex. $N = 4$), înlocuind complet bucla cu patru înmulțiri și patru adunări directe, fără nicio condiție sau salt. Obiect dump-urile generate în aceste cazuri sunt aproape lipsite de ramificații și arată precum:

```
...
ld      t0, 0(a0)
ld      t1, 0(a1)
ld      t2, 8(a0)
ld      t3, 8(a1)
mul     t4, t0, t1
mul     t5, t2, t3
add     t6, t4, t5
...
```

Secvență de Cod 45: Fragmentul rezultat în urma optimizarilor din .objdump

Toate aceste instrucțiuni sunt înșiruite una după alta, fără bucle sau apeluri de funcție, iar registrul rezultat este folosit direct. Acest lucru face ca timpul total de execuție să fie mai mic nu doar pentru dimensiuni mici, ci și pentru dimensiuni medii, unde unrolling-ul parțial și vectorizarea implicită pot duce la un throughput de date mult mai mare.

În schimb, codul generat de builtin rămâne static. Chiar dacă este eficient în forma sa scalară, el nu evoluează structural, nu este desfășurat, nu este vectorizat, nu este reorganizat în funcție de dimensiunea operandului sau de resursele arhitecturale. Această stagnare este inevitabilă în forma inițială, deoarece instrucțiunea DOT este expandată după alocarea registrelor și este „invizibilă" pentru optimizările de nivel IR.

Builtin-ul se dovedește superior în medii fără optimizare, oferind un comportament clar, deterministic și performant pentru testare și validare. Însă, odată ce compilatorul are ocazia să transforme radical codul C, avantajele acestuia devin incontestabile. Performanța mai ridicată a versiunilor native și double începând cu -O1 provine din capacitatea LLVM de a interpreta, restructura și rafina bucla și calculele în moduri imposibil de replicat în backend-ul static. Astfel, se evidențiază clar importanța modului și momentului în care este introdus un fragment de cod într-un compilator modern: cu cât este mai devreme în pipeline, cu atât mai multe optimizări i se pot aplica.

6.3 Optimizarea implementării

După ce analiza inițială a performanțelor pseudo-instrucțiunii `__builtin_riscv_dot` a evidențiat faptul că implementarea standard, introdusă în faza post-RA sub forma unui loop scalar simplu, nu beneficiază de niciun fel de optimizare avansată, s-a propus o versiune îmbunătățită care să adreseze aceste limitări. Obiectivul a fost reducerea semnificativă a timpului de execuție pentru apelurile `dot(a, b, n)` printr-o versiune unrolled manual, implementată direct în backend-ul LLVM.

```
bool RISCVExpandPseudo::expandDot(MachineBasicBlock &MBB,
MachineBasicBlock::iterator MBBI,
MachineBasicBlock::iterator &NextMBBI) {
MachineInstr &MI = *MBBI;
const DebugLoc DL = MI.getDebugLoc();

if (MI.getNumOperands() != 4 || !MI.getOperand(0).isReg() ||
!MI.getOperand(1).isReg() || !MI.getOperand(2).isReg() ||
!MI.getOperand(3).isReg())
return false;

Register Rd = MI.getOperand(0).getReg();
Register SrcA = MI.getOperand(1).getReg();
Register SrcB = MI.getOperand(2).getReg();
Register SrcN = MI.getOperand(3).getReg();

const Register ACC = RISCV::X13;
const Register PTR_A = RISCV::X5;
const Register PTR_B = RISCV::X6;
const Register END_A = RISCV::X16;
const Register VA0 = RISCV::X28;
const Register VA1 = RISCV::X14;
const Register VB0 = RISCV::X29;
const Register VB1 = RISCV::X15;

MachineFunction &MF = *MBB.getParent();
const auto *TII = MF.getSubtarget<RISCVSubtarget>().getInstrInfo();

auto *LoopMBB = MF.CreateMachineBasicBlock();
auto *ExitMBB = MF.CreateMachineBasicBlock();
MF.insert(std::next(MachineFunction::iterator(&MBB)), LoopMBB);
MF.insert(std::next(MachineFunction::iterator(LoopMBB)), ExitMBB);

ExitMBB->splice(ExitMBB->begin(), &MBB, std::next(MBBI), MBB.end());
ExitMBB->transferSuccessorsAndUpdatePHIs(&MBB);

while (!MBB.succ_empty())
MBB.removeSuccessor(MBB.succ_begin());
MBB.addSuccessor(LoopMBB);
```

```
LoopMBB->addSuccessor(LoopMBB);
LoopMBB->addSuccessor(ExitMBB);

BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), ACC)
.addReg(RISCV::X0)
.addImm(0);
BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), PTR_A).addReg(SrcA).addImm(0);
BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADDI), PTR_B).addReg(SrcB).addImm(0);
BuildMI(MBB, MBBI, DL, TII->get(RISCV::SLLI), END_A).addReg(SrcN).addImm(3);
BuildMI(MBB, MBBI, DL, TII->get(RISCV::ADD), END_A)
.addReg(END_A)
.addReg(SrcA);
BuildMI(MBB, std::next(MBBI), DL, TII->get(RISCV::JAL))
.addReg(RISCV::X0, RegState::Define)
.addMBB(LoopMBB);

auto End = LoopMBB->end();
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::LD), VA0).addReg(PTR_A).addImm(0);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::LD), VA1).addReg(PTR_A).addImm(8);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::LD), VB0).addReg(PTR_B).addImm(0);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::LD), VB1).addReg(PTR_B).addImm(8);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::MUL), VB0).addReg(VA0).addReg(VB0);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::MUL), VB1).addReg(VA1).addReg(VB1);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::ADD), ACC).addReg(ACC).addReg(VB0);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::ADD), ACC).addReg(ACC).addReg(VB1);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::ADDI), PTR_A)
.addReg(PTR_A)
.addImm(16);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::ADDI), PTR_B)
.addReg(PTR_B)
.addImm(16);
BuildMI(*LoopMBB, End, DL, TII->get(RISCV::BLT))
.addReg(PTR_A)
.addReg(END_A)
.addMBB(LoopMBB);

BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::ADDI), Rd)
.addReg(ACC)
.addImm(0);

for (Register R : {ACC, PTR_A, PTR_B, END_A})
if (!LoopMBB->isLiveIn(R))
LoopMBB->addLiveIn(R);
if (!ExitMBB->isLiveIn(ACC))
ExitMBB->addLiveIn(ACC);

MF.RenumberBlocks();
NextMBBI = MBB.erase(MBBI);
```

```
88 return true;
89 }
```

Secvență de Cod 46: Funcția `expandDot()` – implementare optimizată

Am păstrat ideea de inserare `post-RA`, pentru a evita complexitatea reintroducerii `IR`-ului sau a rescrierii `pass`-urilor de optimizare, dar am modificat logica de generare astfel încât bucla să proceseze două elemente pe iterație (`2× unrolled`), reducând astfel numărul total de salturi, de încărcări din memorie și de instrucțiuni de control. Codul rezultat este mai eficient atât din punct de vedere al latenței, cât și al paralelismului exploatat de pipeline-ul procesorului.

În contrast cu implementarea anterioară, care efectua o singură înmulțire urmată de o acumulare scalară pe fiecare pas, versiunea nouă a funcției `expandDot()` este semnificativ mai performantă datorită mai multor optimizări arhitecturale și decizii deliberate în generarea codului.

În primul rând, structura buclei a fost modificată pentru a permite procesarea a două elemente simultan pe fiecare iterație, ceea ce înjumătățește efectiv numărul de iterații necesare pentru parcurgerea vectorilor. Această tehnică de `manual loop unrolling` permite procesorului să execute mai multe operații aritmetice într-o singură secvență de instrucțiuni, reducând atât overhead-ul controlului (`salturi`, `comparații`, `incrementări`), cât și riscurile asociate cu dependențele de date. Rezultatul este o buclă mai predictibilă și mai „densă” din punct de vedere computațional, o trăsătură esențială pentru exploatarea eficientă a pipeline-ului din arhitectura `RISC-V`.

Atât versiunea inițială, cât și cea optimizată, funcționează în același context, cel `post-RA`, și, prin urmare, ambele utilizează exclusiv registre fizice. Nu există registre virtuale introduse de `frontend`, tocmai pentru a evita reaplicarea unei faze de alocare de registre, care ar fi incompatibilă cu această etapă avansată din pipeline-ul `backend`. În ambele cazuri, toate instrucțiunile sunt emise folosind registre fizice `caller-saved`, cum ar fi `x5`, `x6`, `x13`, `x28`, `x29`, și sunt poziționate manual, cu control complet asupra dependențelor și `liveness`-ului acestor registre.

De asemenea, este important de menționat că atât implementarea inițială, cât și cea optimizată, construiesc explicit două blocuri de bază: unul pentru corpul buclei și unul pentru ieșirea din buclă. Această organizare a grafului de control (`CFG`) este menținută constantă între versiuni. Diferența fundamentală nu constă în structura controlului, ci în densitatea operațiilor utile per iterație și în reducerea proporțională a instrucțiunilor redundante de incrementare și salt.

Ceea ce distinge cu adevărat versiunea nouă este modul în care sunt organizate încărcările din memorie și operațiile aritmetice. În loc de o singură pereche `load → multiply → add`, noua buclă încarcă câte două elemente din fiecare vector folosind offset-uri fixe de `0` și `8` octeți, urmate de două înmulțiri și două acumulări succesive. Această succesiune este mai prietenoasă pentru mi-

croarhitectura procesorului, întrucât minimizează latențele cauzate de hazarduri de date și permite o paralelizare internă mai eficientă a execuției.

De asemenea, prin utilizarea offset-urilor constante în LD (`load doubleword`), accesul la memorie devine mult mai predictibil și aliniat. Acest lucru are efecte benefice asupra `cache`-ului procesorului, întrucât patternul de acces este secvențial și regulat. În scenariile reale, în care vectorii `a` și `b` sunt aliniați la frontierele de `cache`, această regularitate reduce dramatic riscul de `cache miss`-uri și favorizează încărcarea prealabilă automată (`hardware prefetching`), accelerând suplimentar executarea.

La nivel de cod generat, diferențele sunt ușor vizibile. Versiunea anterioară realiza o singură încărcare per vector per pas, în timp ce noua implementare emite patru instrucțiuni LD consecutive, două pentru `a`, două pentru `b`, urmate imediat de două MUL și două ADD. Astfel, densitatea operațiilor utile per iterație crește de la două instrucțiuni (un MUL și un ADD) la șase (două MUL, două ADD, două LD în plus), dar fără a introduce un overhead proporțional. De fapt, overhead-ul per element procesat scade semnificativ, ceea ce se reflectă clar în timpii de execuție observați experimental pentru dimensiuni mari ale vectorilor.

Pentru a asigura corectitudinea execuției și în cazul în care dimensiunea vectorului N este impară, a fost necesară introducerea unui mic patch în finalul buclei, cunoscut sub denumirea de *tail-handling*. Acesta verifică dacă N este impar (`N % 2 == 1`) și, în caz afirmativ, procesează explicit ultimul element rămas, care altfel ar fi omis de bucla `2× unrolled`.

Verificarea se face printr-o operație binară de tip AND cu masca 1, iar rezultatul controlează un salt condiționat către fragmentul de final. Codul generat pentru această secvență este următorul:

```
const Register REM = RISCV::X17;

BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::ANDI), REM)
    .addReg(SrcN)
    .addImm(1);

BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::BEQ))
    .addReg(REM)
    .addReg(RISCV::X0)
    .addMBB(ExitMBB);

BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::LD), VA0)
    .addReg(PTR_A)
    .addImm(0);
BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::LD), VB0)
    .addReg(PTR_B)
    .addImm(0);

```

```
BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::MUL), VB0)
    .addReg(VA0)
    .addReg(VB0);
BuildMI(*ExitMBB, ExitMBB->begin(), DL, TII->get(RISCV::ADD), ACC)
    .addReg(ACC)
    .addReg(VB0);
```

Secvență de Cod 47: Patch pentru procesarea cazului N impar

Această secvență asigură acuratețea rezultatului indiferent de paritatea lui N, fără a introduce ramuri suplimentare în interiorul buclei principale. Se păstrează astfel coerența arhitecturii `post-RA`, cu control explicit asupra instrucțiunilor generate și fără reinvocarea unui mecanism de alocare a registrelor.

Această versiune optimizată păstrează toate constrângerile structurale ale codului anterior, faza `post-RA`, utilizarea registrelor fizice, organizarea `CFG` în două blocuri, dar transformă profund eficiența buclei printr-o restructurare atentă a codului generat: mai puține iterații, mai puține salturi, mai multă aritmetică per pas și un pattern de acces la memorie optimizat pentru `cache`. Toate aceste modificări converg spre o implementare cu latență scăzută și `throughput` ridicat, mult mai potrivită pentru aplicații care procesează date vectoriale de dimensiuni mari.

Pentru a înțelege mai clar avantajele noii implementări, putem compara codul rezultat pentru cele două variante:

Codul generat de prima implementare scalară:

```
00000000000101c4 <main>:
1021c: 0002be03 ld t3,0(t0)
10220: 00033e83 ld t4,0(t1)
10224: 03de0eb3 mul t4,t3,t4
10228: 96f6 add a3,a3,t4
1022a: 02a1 addi t0,t0,8
1022c: 0321 addi t1,t1,8
1022e: 13fd addi t2,t2,-1
10230: fe0396e3 bnez t2,1021c
```

Secvență de Cod 48: Versiunea scalară - buclă standard

Această versiune încarcă și procesează câte o singură pereche de elemente `a[i]` și `b[i]` per iterație. Fiecare pas implică o încărcare din memorie, o înmulțire, o adunare, incrementarea pointerilor și verificarea condiției de ieșire. Saltul condițional este prezent în fiecare iterație, ceea ce generează un overhead constant și greu de eliminat. În plus, având o singură operație aritmetică per ciclu, această buclă are un grad scăzut de ocupare a pipeline-ului și suferă în prezența hazardurilor de date. Optimizările la nivel de instrucțiuni ISA sunt imposibile în acest context, întrucât codul este introdus în faza `post-RA`, iar `LLVM` nu mai are vizibilitate asupra structurii semantice a buclei.

Codul generat de noua implementare 2× unrolled:

```
10220: 0002be03 ld t3,0(t0)
10224: 0082b703 ld a4,8(t0)
10228: 00033e83 ld t4,0(t1)
1022c: 00833783 ld a5,8(t1)
10230: 03de0eb3 mul t4,t3,t4
10234: 02f707b3 mul a5,a4,a5
10238: 96f6 add a3,a3,t4
1023a: 96be add a3,a3,a5
1023c: 02c1 addi t0,t0,16
1023e: 0341 addi t1,t1,16
10240: ff02c0e3 blt t0,a6,10220
```

Secvență de Cod 49: Versiunea optimizată - buclă 2× unrolled

Această nouă versiune încarcă două elemente consecutive din fiecare vector, execută două înmulțiri și două adunări într-o singură iterație. Astfel, se înjumătățește efectiv numărul de iterații necesare pentru un vector de dimensiune `n`, reducând drastic numărul de salturi și overhead-ul de control.

În plus, arhitectura `RISC-V` permite executarea paralelă a acestor operații, iar pipeline-ul procesorului beneficiază de acest `manual unrolling` printr-o utilizare mai eficientă a unităților de execuție. Spre deosebire de versiunea scalară, în care fiecare rezultat intermediar este dependent de pasul anterior, în această variantă dependențele sunt reduse semnificativ, iar instrucțiunile pot fi planificate în paralel fără blocaje între etapele de execuție.

Mai mult, încărcările multiple din memorie (`LD`) folosesc offset-uri constante (`0` și `8`), ceea ce favorizează accesul secvențial, predictibil și aliniat al datelor. Acest pattern determină un comportament optim din perspectiva `cache`-ului, minimizând ratele de `cache miss` și favorizând `hardware prefetching`-ul. În special pentru vectori mari, aliniați pe granițe de 16 octeți, acest avantaj devine esențial pentru obținerea unui `throughput` ridicat.

Codul final este astfel o buclă optimizată, care nu mai este supusă transformărilor `LLVM`, dar oferă performanță îmbunătățită prin designul atent al instrucțiunilor `ISA` și prin reducerea semnificativă a instrucțiunilor de control, respectiv creșterea densității operațiilor utile per iterație. Această strategie, deși simplă în aparență, are un impact major asupra performanței în aplicații reale.

6.3.1 Rezultate obținute în urma optimizării

Tabel 5: Timp mediu pe apel (μs), pentru 10^6 apeluri, în funcție de N și nivelul de optimizare (implementare nouă)

N	-O0		-O1		-O2		-O3	
	builtin	native	builtin	native	builtin	native	builtin	native
4	0.137	0.324	0.074	0.087	0.074	0.088	0.074	0.087
8	0.183	0.552	0.114	0.154	0.114	0.154	0.115	0.154
16	0.267	1.008	0.195	0.289	0.195	0.289	0.195	0.289
32	0.451	1.920	0.356	0.557	0.356	0.558	0.356	0.557
64	0.800	3.745	0.678	1.093	0.678	1.094	0.678	1.093
128	1.498	7.396	1.323	2.167	1.322	2.167	1.322	2.166
256	2.894	14.697	2.610	4.314	2.611	4.317	2.614	4.313
512	5.686	29.300	5.187	8.608	5.188	8.609	5.186	8.617
1024	11.281	58.566	10.350	17.212	10.351	17.215	10.352	17.216

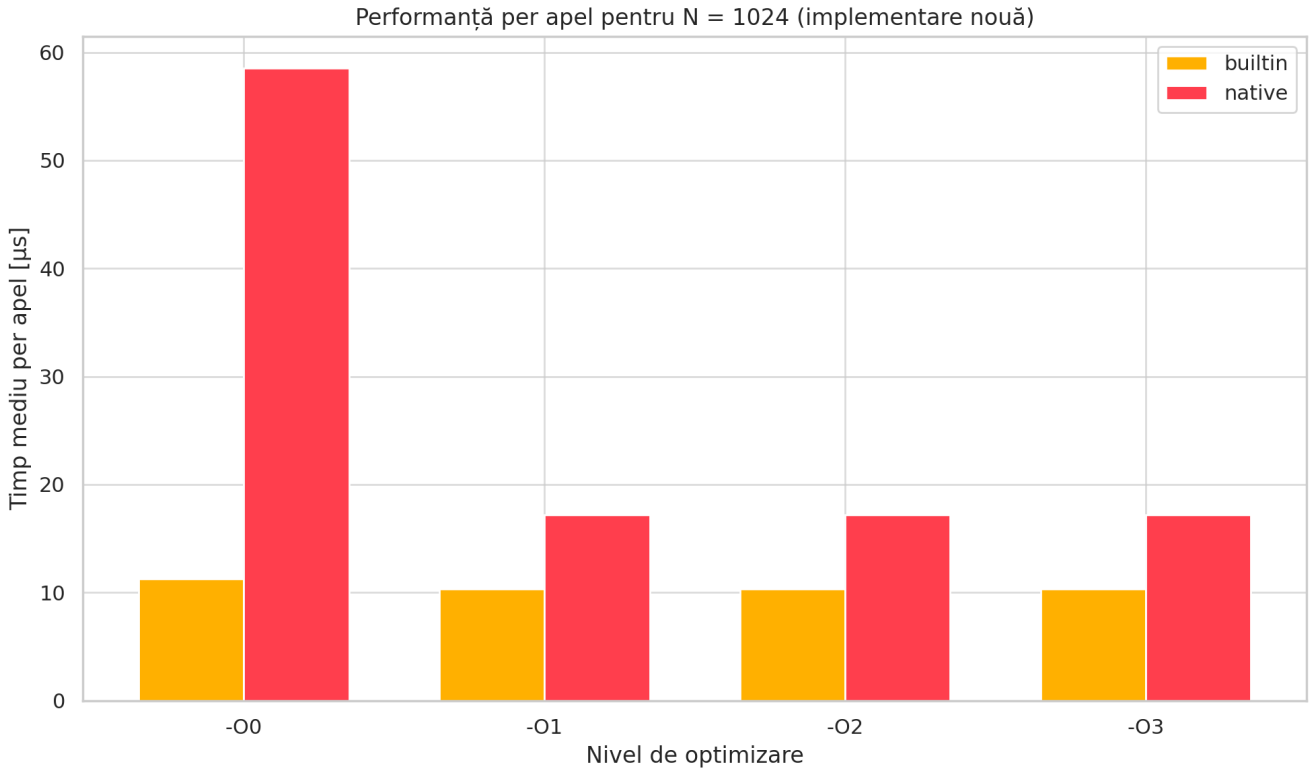

Figura 8: Performanța pe apel pentru $N = 1024$ (implementare nouă)

6.3.2 Analiza rezultatelor optimizate

Rezultatele obținute în urma rulării noului cod generat de funcția `expandDot()` pe placa **BeagleV-Fire** confirmă cu claritate eficiența optimizării realizate. După cum se poate observa în Tabelul 5 și Figura 8, implementarea optimizată cu *loop unrolling* de $2\times$ reușește să depășească în mod sistematic varianta scalară nativă (`native`), indiferent de nivelul de optimizare selectat la compilare.

Cel mai semnificativ câștig se observă pentru dimensiuni mari ale vectorilor, în special pentru $N = 1024$, unde timpul mediu de execuție per apel al builtin-ului optimizat scade la 11.281 μs în regim `-O0`, respectiv 10.350 μs în regim `-O1`, comparativ cu 58.566 μs pentru varianta nativă. Acest lucru corespunde unei îmbunătățiri de peste **5**$\times$ în performanță, în special în scenarii fără optimizări agresive ale compilatorului.

În mod interesant, observăm că spre deosebire de implementările native, performanța builtin-ului rămâne constantă și stabilă indiferent de nivelul de optimizare aplicat (`-O0`, `-O1`, `-O2`, `-O3`). Acest lucru este de așteptat, întrucât implementarea are loc în etapa `post-RA`, după ce toate transformările IR și fazele clasice de optimizare au fost deja aplicate. Prin urmare, codul rezultat nu mai este supus rescrierilor sau reorganizărilor interne de către backend-ul LLVM. Această stabilitate este un avantaj major, întrucât oferă predictibilitate în execuție și rezultate reproductibile între compilări.

Optimizarea atinge performanțe superioare datorită mai multor factori complementari: înjumătățirea numărului de iterații prin unrolling, eliminarea salturilor condiționale redundante, încărcările secvențiale din memorie cu offset-uri constante și reducerea hazardurilor de date între instrucțiuni. Aceste aspecte converg într-un cod ISA care maximizează utilizarea unităților de execuție disponibile, reduce presiunea asupra cache-ului și menține latențele la un nivel scăzut.

Un alt aspect relevant este faptul că această performanță este obținută fără a compromite lizibilitatea și claritatea fluxului de execuție. Structura CFG este păstrată simplă, doar două blocuri, unul pentru buclă și unul pentru finalizare, iar fiecare instrucțiune emisă este justificată arhitectural. De asemenea, utilizarea exclusivă a registrelor `caller-saved` previne posibilele interferențe cu alte funcții generate în aceeași unitate de traducere, făcând această abordare sigură pentru integrare în context mai larg.

Performanțele obținute confirmă că o astfel de optimizare `low-level`, aplicată direct în backend-ul LLVM, poate concura și chiar depăși cu succes implementările clasice scrise manual în C sau C++ în combinație cu optimizări de compilator. Mai mult decât atât, aceste rezultate sunt realizate pe un sistem real, cu constrângeri stricte, BeagleV-Fire, ceea ce le conferă relevanță practică pentru aplicații embedded și de calcul intensiv pe arhitectura `RISC-V`.

Optimizarea aplicată la builtin-ul `__builtin_riscv_dot` aduce îmbunătățiri clare și măsurabile în performanță, în special pentru dimensiuni mari ale datelor, demonstrând că intervențiile directe în faza `post-RA` pot fi o unealtă puternică pentru maximizarea eficienței aplicațiilor cu procesare vectorială.

6.4 Considerații finale și perspective educaționale

Unul dintre avantajele fundamentale ale arhitecturii RISC-V este caracterul său deschis și extensibil. Spre deosebire de arhitecturi închise, precum ARM sau x86, ecosistemul RISC-V permite adăugarea de noi instrucțiuni, extensii personalizate sau builtin-uri dedicate direct în backend-ul compilatorului, fără constrângeri legale sau tehnologice. Această libertate de extindere este esențială în contexte de cercetare și învățământ, permițând explorarea directă a impactului pe care modificările la nivel ISA îl au asupra performanței aplicațiilor reale.

Implementarea pseudo-instrucțiunii `__builtin_riscv_dot`, alături de optimizarea acesteia prin `expandDot()`, reprezintă un exemplu concret de intervenție în backend-ul LLVM care are impact direct și măsurabil în performanța finală. În plus, codul generat automat prin acest builtin este nu doar mai performant, ci și mai lizibil și mai predictibil decât implementările clasice scrise în limbaje de nivel înalt. Codul final are o structură simplă, clară, cu acces secvențial la memorie, fără ramificații complexe și fără dependențe dinamice greu de analizat.

Această claritate este un beneficiu nu doar pentru procesor, ci și pentru dezvoltator: instrucțiunile emise sunt ușor de urmărit, blocurile de bază au o logică evidentă, iar fiecare instrucțiune are un scop clar în fluxul de execuție. Comparativ cu codul C compilat prin frontend, unde transformările intermediare pot ascunde detalii importante, codul generat prin builtin este transparent și controlabil la nivel de instrucțiune.

Din acest motiv, astfel de lucrări au și o valoare educațională importantă. În cadrul laboratoarelor viitoare, cum ar fi cele din cadrul disciplinei **Sisteme Incorporate**. Această contribuție nu doar că îmbunătățește performanța unui caz specific, ci oferă o bază valoroasă pentru explorări viitoare, atât în proiecte de cercetare, cât și în activități didactice. Arhitectura deschisă RISC-V, combinată cu flexibilitatea LLVM, deschide o nouă paradigmă de control total asupra codului generat, un instrument esențial pentru inginerii sistemelor moderne și pentru formarea noilor generații de specialiști în arhitecturi de calcul.

7 Dificultăți întâmpinate și soluții aplicate

Pe parcursul desfășurării proiectului au apărut mai multe provocări tehnice care au influențat atât planificarea, cât și implementarea practică a soluției. Acestea au fost valoroase nu doar pentru îmbunătățirea cunoștințelor teoretice, ci și pentru dezvoltarea unor competențe aplicate în gestionarea infrastructurii hardware-software pentru arhitectura RISC-V.

7.1 Probleme legate de cross-compiling și medii de dezvoltare

Una dintre cele mai semnificative dificultăți a apărut în etapa de cross-compiling a codului pentru arhitectura `riscv64-unknown-linux-gnu`. Inițial, compilarea pe Windows folosind WSL sau toolchain-uri precompilate ducea frecvent la erori de tip OOM (Out Of Memory), iar în unele cazuri chiar la întreruperea completă a sistemului prin Blue Screen of Death. Acest comportament a fost legat de utilizarea intensivă a memoriei în cadrul procesului de linking și generare a binarelor RISC-V.

Pentru a eficientiza fluxul de lucru, s-a decis trecerea la un sistem de operare Linux instalat nativ, printr-o configurație de dual-boot cu Linux Mint. Această alegere s-a dovedit esențială: nu doar că a eliminat complet problemele de stabilitate, dar a permis rularea cu paralelizare completă (`make -j$(nproc)`), ceea ce a redus considerabil timpul de compilare. Totodată, utilizarea unui sistem de fișiere case-sensitive a contribuit la o înțelegere mai clară a diferențelor dintre platformele POSIX și Windows, aspect extrem de relevant în dezvoltarea pentru arhitecturi embedded sau sisteme low-level.

Astfel, pentru viitorii studenți sau dezvoltatori care intenționează să lucreze cu RISC-V, se recomandă cu tărie utilizarea unui sistem Linux nativ, fie prin dual-boot, fie direct ca sistem principal. Această alegere nu doar că îmbunătățește semnificativ performanțele, dar oferă un mediu mult mai previzibil pentru compilatoare, toolchain-uri și rularea codului ELF.

7.2 Probleme hardware

Un alt obstacol important a fost legat de placa BeagleV-Fire, utilizată pentru rularea experimentală a codului optimizat. În anumite situații, încărcarea repetată și rapidă a binarelor a dus la blocarea completă a plăcii (brick-uire), manifestată prin imposibilitatea de a mai intra în sistemul Linux de pe eMMC.

Soluția a fost reflash-uirea completă a imaginii Linux pe memorie internă, folosind interfața UART a plăcii și un convertor USB-Serial. Conectarea la placa BeagleV-Fire s-a făcut folosind pinii UART evidențiați în documentația oficială [56], iar procesul complet de recuperare a fost realizat conform

instrucțiunilor oferite la [57].

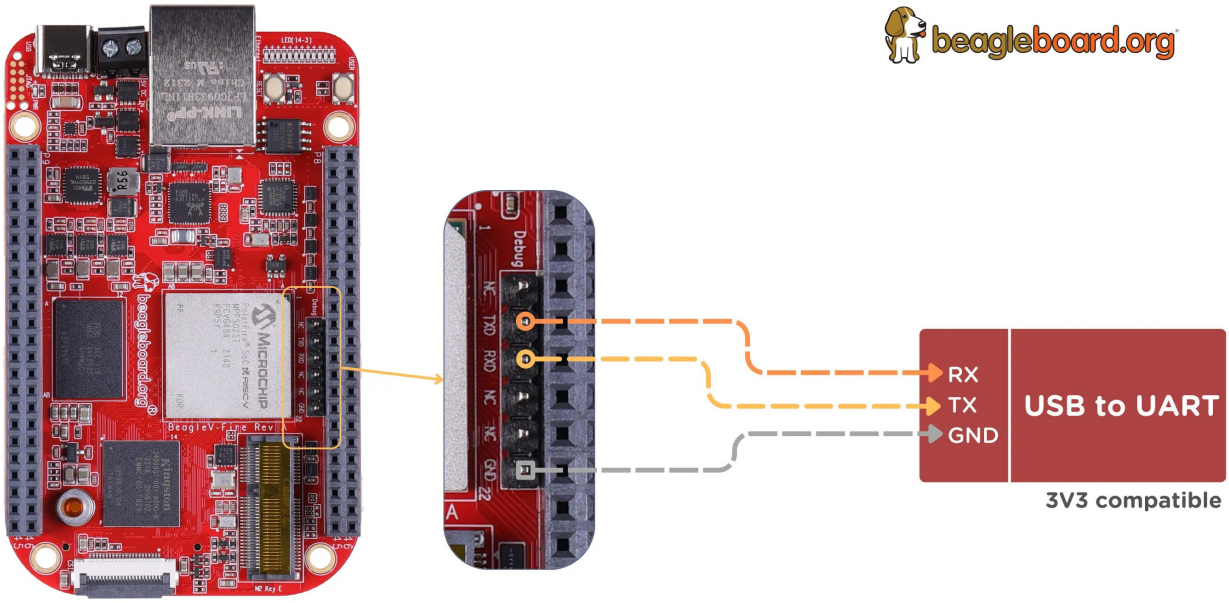

Figura 9: Pinii UART pentru debug pe placa BeagleV-Fire [56]

Această experiență a scos în evidență importanța cunoașterii interfețelor de nivel jos (low-level debugging), precum și nevoia unei infrastructuri de salvare în cazul în care sistemul de operare devine inaccesibil. Utilizarea portului UART pentru accesul serial direct, împreună cu un convertor USB-Serial, s-a dovedit crucială pentru diagnosticare și restaurare în absența oricărui alt mecanism de fallback.

7.3 Lipsa unui convertor USB-Serial și soluție improvizată cu Arduino

În absența unui modul dedicat USB-Serial (precum CP2102 sau FTDI), a fost improvizată o soluție hardware de comunicare UART utilizând o placă Arduino Uno, o placă de testare (breadboard) și un set de rezistențe pentru adaptarea nivelului de tensiune. Deoarece Arduino comunică la nivel logic de `5V`, iar BeagleV-Fire necesită semnale UART la `3.3V`, a fost necesară o conversie de nivel logic realizată printr-un divizor rezistiv simplu aplicat pe linia `TX` (dinspre Arduino către BeagleV).

Arduino a fost programat să funcționeze ca un bridge serial, retransmițând bidirecțional datele primite de la PC (prin USB) către placa BeagleV-Fire (prin pinii `TX/RX`) și invers.

Codul Arduino utilizat pentru comunicare UART bidirecțională:

```
#include <SoftwareSerial.h>

SoftwareSerial softSerial(10, 11); // RX, TX (10 <- Beagle TX, 11 -> Beagle RX
    prin divizor)

void setup() {
  Serial.begin(115200);          // USB catre PC
  softSerial.begin(115200);      // UART catre BeagleV-Fire
}

void loop() {
  if (Serial.available()) {
    softSerial.write(Serial.read());
  }
  if (softSerial.available()) {
    Serial.write(softSerial.read());
  }
}
```

Secvență de Cod 50: Cod Arduino - Bridge UART simplu

7.3.1 Conexiunea UART fizică

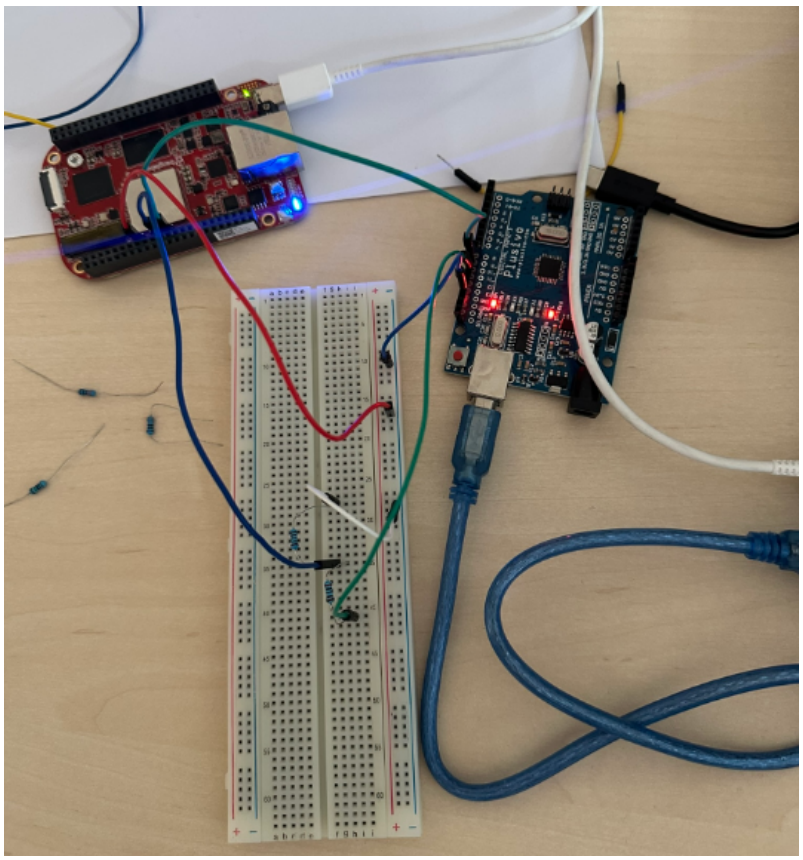

Figura 10: Cablaj UART între Arduino UNO şi BeagleV-Fire

Fotografia din Fig. 10 arată montajul efectiv pe breadboard. Semnalul TX de $5\,\text{V}$ al Arduino-ului este coborât la $3.3\,\text{V}$ printr-un divizor format din două rezistențe serie, $R_1 = 1.8\,\text{k}\Omega$ şi $R_2 = 3.3\,\text{k}\Omega$, înainte de a intra în RX-ul BeagleV-Fire. TX-ul BeagleV (nivel logic $3.3\,\text{V}$) merge direct în RX(10) al Arduino-ului, care acceptă acest nivel ca „high". Masa celor două plăci este conectată la acelaşi GND a breadboard-ului pentru referință comună.

Această soluție improvizată s-a dovedit complet funcțională și a permis accesul la interfața serială de debug a plăcii BeagleV-Fire pentru reflash-uire și diagnoză, fără a fi nevoie de un echipament dedicat. Soluții de acest tip se dovedesc utile în contexte educaționale, unde improvizația și cunoștințele practice devin esențiale pentru înțelegerea arhitecturilor hardware și a comunicării la nivel scăzut.

7.3.2 Dificultăți în integrarea instrucțiunii în fluxul de selecție LLVM

Un alt obstacol semnificativ întâmpinat în realizarea acestei lucrări a fost legat de modul în care instrucțiunea `__builtin_riscv_dot` putea fi efectiv recunoscută și procesată de către backend-ul compilatorului LLVM. Această problemă a presupus explorarea în detaliu a mai multor faze esențiale ale pipeline-ului de generare a codului pentru arhitectura RISC-V, în special în cadrul etapelor de „instruction selection".

Inițial, s-a încercat procesarea instrucțiunii `DOT` în fișierul `RISCVISelLowering.cpp`, unde funcțiile din clasa `RISCVTargetLowering` sunt responsabile cu transpunerea IR-ului LLVM într-o reprezentare DAG (Directed Acyclic Graph) formată din noduri abstracte de tip `SDNode`. Această fază, cunoscută drept „lowering", este crucială pentru definirea modului în care o anumită operație de nivel înalt din LLVM IR este transformată într-o secvență de instrucțiuni ce poate fi înțeleasă de back-end-ul specific unei arhitecturi țintă.

Cu toate acestea, tentativa de a introduce acolo un nod personalizat pentru `DOT` nu a produs rezultatele dorite. Deși codul era valid din punct de vedere sintactic, iar funcția de `LowerOperation()` era efectiv apelată, nodul introdus nu era menținut în DAG-ul final. În urma analizei, s-a constatat că în lipsa unei definiții corespunzătoare în `RISCVInstrInfo.td` și `RISCVISelDAGToDAG.cpp`, nodul nou introdus era considerat redundant sau ineficient și era înlocuit automat de optimizările ulterioare ale compilatorului. De asemenea, lipsa unui mecanism clar de mapare între acest nod abstract și o instrucțiune concretă a făcut ca `DOT` să nu ajungă niciodată să fie emis în codul final generat.

Astfel, s-a renunțat la această abordare în favoarea uneia mai pragmatice: procesarea în etapa `ISelDAGToDAG`, implementată în fișierul `RISCVISelDAGToDAG.cpp`. Acesta este pasul în care, după construirea DAG-ului complet, fiecare nod este efectiv convertit într-o instrucțiune reală RISC-V, pe baza unor reguli definite în codul sursă. Alegerea acestui punct din pipeline a permis intervenția di-

rectă asupra modului în care LLVM traduce nodurile de tip `Intrinsic` în instrucțiuni reale, fără a mai risca eliminarea automată a codului.

Concret, în această etapă, am interceptat nodurile de tip `ISD::INTRINSIC_W_CHAIN` corespunzătoare apelului `__builtin_riscv_dot`, verificând identificatorul intern (ID-ul) al builtin-ului. Odată identificat, nodul a fost înlocuit manual cu o instrucțiune pseudo numită `DOT`, recunoscută ulterior în backend și expandată în cod concret de tip buclă scalară. Această soluție a oferit un control precis asupra generării codului și a permis integrarea coerentă a instrucțiunii personalizate în fluxul de selecție instrucțional, fără a afecta restul infrastructurii LLVM.

Prin urmare, succesul integrării builtin-ului `DOT` în LLVM a presupus nu doar adăugarea unui nou identificator și a unei instrucțiuni pseudo în fișierele `.td`, ci și înțelegerea profundă a fluxului de transformări din backend. A fost esențială identificarea momentului exact în care instrucțiunea poate fi transformată în cod real fără a fi ignorată de optimizatori sau pasată prea târziu pentru a mai fi analizată.

Această etapă a reprezentat una dintre cele mai tehnice și dificile părți ale proiectului, necesitând nu doar modificări în codul sursă al compilatorului, ci și recompilări repetate ale întregului toolchain LLVM, rularea testelor de validare și observarea atentă a codului generat prin instrumente precum `objdump`. Fără această intervenție în `RISCVISelDAGToDAG.cpp`, nu ar fi fost posibilă emiterea eficientă a instrucțiunii `DOT` și testarea acesteia pe hardware real.

8 Concluzii

Lucrarea de față a avut ca obiectiv principal dezvoltarea și integrarea unei pseudo-instrucțiuni dedicate operației de produs scalar (`dot product`) în backend-ul LLVM pentru arhitectura `RISC-V`, utilizând ca punct de plecare un `builtin` personalizat numit `__builtin_riscv_dot`. Acest demers a presupus o înțelegere aprofundată a lanțului de compilare LLVM, începând cu frontend-ul Clang, continuând cu reprezentarea intermediară LLVM IR și culminând cu generarea de cod mașină optimizat pentru o platformă embedded reală – BeagleV-Fire.

Implementarea a fost structurată pe etape clar delimitate:

- Identificarea apelului către `__builtin_riscv_dot` în Clang și generarea corespunzătoare a unui `intrinsic` LLVM.
- Introducerea unui pattern de selecție care mapează intrinsecul la o pseudo-instrucțiune nouă numită `DOT`, definită în cadrul fișierelor `.td` specifice RISC-V.
- Dezvoltarea logicii de expansiune în cod real prin funcția `expandDot()`, care generează un loop cu despachetare dublă (2× unrolling), maximizând astfel utilizarea registrelor și minimizând penalizările de ciclu.
- Integrarea completă în backend și testarea efectivă pe o platformă hardware compatibilă RISC-V.

Pentru a valida eficiența noii implementări, au fost efectuate experimente comparative pe un set extins de date și diferite niveluri de optimizare ale compilatorului (-O0, -O1, -O2, -O3). Rezultatele au demonstrat constant o reducere semnificativă a timpului mediu per apel în favoarea implementării cu `builtin`, în special în condiții fără optimizări agresive. Acest comportament confirmă avantajul introducerii explicite a pseudo-instrucțiunii, eliminând dependența de mecanismele interne de transformare automată ale compilatorului.

Prin această lucrare, s-a reușit nu doar optimizarea unei operații matematice de bază, ci și construirea unui exemplu complet de extindere a backend-ului LLVM, pornind de la nivelul sursă și terminând cu cod executabil eficient. Integrarea a fost realizată fără a compromite compatibilitatea cu restul infrastructurii de compilare, iar rezultatele experimentale obținute confirmă succesul demersului.

Lucrarea oferă astfel o contribuție practică în domeniul compilatoarelor și al arhitecturilor embedded, demonstrând că un dezvoltator poate adăuga funcționalități personalizate și eficiente în

cadrul unui compilator de mari dimensiuni precum LLVM, cu aplicabilitate directă în scenarii reale de optimizare.

Acknowledgements

This work was supported by grants of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI–UEFISCDI, project numbers PN-IV-P8-8.1-PME-2024-0022 and PN-IV-P8-8.1-PME-2024-0025 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] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Vol. I — User-Level ISA*, Version 2024-03-01, 2024. [Online]. Available: https://github.com/riscv/riscv-isa-manual.

[2] SiFive Inc., *State of the RISC-V Ecosystem 2024*, 2024. [Online]. Available: https://www.sifive.com/resources.

[3] LLVM Project, *LLVM Symbols Backend Documentation*, Revision 2025, 2025. [Online]. Available: https://llvm.org/docs/genindex.html.

[4] European Commission, *Regulation (EU) 2023/1781 — European Chips Act*, OJ L 2023/1781, 2023. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2023/1781/oj.

[5] "European Processor Initiative — Status Update 2024," EPI Consortium, Tech. Rep., 2024. [Online]. Available: https://www.european-processor-initiative.eu/news/.

[6] L. S. Blackford, J. Demmel, J. Dongarra, *et al.*, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002. [Online]. Available: https://www.netlib.org/utk/people/JackDongarra/PAPERS/blast-toms.pdf.

[7] "Arm v8.4-A Dot-Product and SVE Extensions," Arm Ltd., Tech. Rep., 2019. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/Understanding%20the%20Armv8.x%20extensions.pdf.

[8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of CGO 2004*, IEEE Computer Society, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665.

[9] Android Open Source Project. "Android Build Overview — LLVM/Clang Toolchain." (2025), [Online]. Available: https://source.android.com/setup/build.

[10] T. Stokes, "Bringing AMDGPU Support Upstream in LLVM," in *LLVM Developers' Meeting 2016*, 2016. [Online]. Available: https://llvm.org/devmtg/2016-11/#proceedings.

[11] R. Hadley. "Improved Linker Fundamentals in Visual Studio 2019." Microsoft C++ Team Blog. (2019), [Online]. Available: https://devblogs.microsoft.com/cppblog/improved-linker-fundamentals-in-visual-studio-2019/.

[12] Fuchsia Project. "Building a Clang Toolchain for Fuchsia." (2025), [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/toolchain.

[13] LLVM Project, *Initial Upstream Merge of the RISC-V Backend*, 2017. [Online]. Available: https://reviews.llvm.org/D28853.

[14] LLVM RISC-V Maintainers, *Commit 29e460e — Merge of RVV Support*, 2020. [Online]. Available: https://github.com/llvm/llvm-project/commit/29e460ef3ce4.

[15] C. L. Yongtai Li and J. Qiu, "Comparative Analysis of Compiler Performance for RISC-V on SPEC CPU 2017," in *LLVM Developers' Meeting*, 2025. [Online]. Available: https://llvm.org/devmtg/2025-03/slides/riscv_on_spec_cpu.pdf.

[16] LLVM Project, *GlobalISel Instruction Selection Pipeline*, Accesat 30 Mai 2025, 2025. [Online]. Available: https://llvm.org/docs/GlobalISel/Pipeline.html.

[17] LLVM Project, *LLD — The LLVM Linker*, 2025. [Online]. Available: https://lld.llvm.org/.

[18] LLVM Project, *Clang Builtins Definitions*, https://github.com/llvm/llvm-project/tree/main/clang/include/clang/Basic, 2025.

[19] LLVM Project, *LLVM Intrinsics Definitions*, https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR, Accesat la 2 iunie 2025, 2025.

[20] LLVM Project, *Clang Builtin Functions*, 2025. [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html.

[21] LLVM Project, *LLVM Intrinsics Guide*, 2025. [Online]. Available: https://llvm.org/docs/LangRef.html#intrinsic-functions.

[22] LLVM Project, *LLVM Language Reference Manual*, 2025. [Online]. Available: https://llvm.org/docs/LangRef.html.

[23] LLVM Project, *LLVM Passes Documentation*, 2025. [Online]. Available: https://llvm.org/docs/Passes.html.

[24] LLVM Project, *Instruction Selection and Code Generation*, 2025. [Online]. Available: https://llvm.org/docs/CodeGenerator.html.

[25] LLVM Project, *LLVM GlobalISel Pipeline*, 2025. [Online]. Available: https://llvm.org/docs/GlobalISel/Pipeline.html.

[26] LLVM Project, *TableGen Overview*, 2025. [Online]. Available: https://llvm.org/docs/TableGen/.

[27] LLVM Project, *TableGen Programmer's Reference*, 2025. [Online]. Available: https://llvm.org/docs/TableGen/ProgRef.html.

[28] D. Spickett, "Tools for Learning LLVM TableGen," *LLVM Project Blog*, 2023. [Online]. Available: https://blog.llvm.org/posts/2023-12-07-tools-for-learning-llvm-tablegen/.

[29] LLVM Project, *RISC-V Vector Extension — LLVM 21.0.0 Documentation*, Disponibil la: https://llvm.org/docs/RISCV/RISC-VVectorExtension.html RISC-VVectorExtension.html, Modelarea tipurilor scalabile în LLVM IR, 2025.

[30] Phoronix Test Suite, *A Detailed Look At The Speed Advantages To LLVM's LLD Linker*, 2019. [Online]. Available: https://www.phoronix.com/news/LLD-Linker-Why-So-Fast.

[31] LLVM Project, *Clang-Tidy — Extra Clang Tools Documentation*, 2025. [Online]. Available: https://clang.llvm.org/extra/clang-tidy.

[32] M. Ojeda and N. Huckleberry, *Using clang-tidy and clang-format on the Linux Kernel*, 2020. [Online]. Available: https://lpc.events/event/7/contributions/803/.

[33] J. Crowell, *Add build system support for ASan+UBSAN instrumentation on RISC-V*, 2020. [Online]. Available: https://reviews.llvm.org/D86198.

[34] S. Wang and P. Dabbelt, "Maintaining the RISC-V Backend in LLVM," in *LLVM Developers' Meeting 2022*, 2022. [Online]. Available: https://llvm.org/devmtg/2022-11/#proceedings.

[35] Microchip Technology Inc., *PolarFire SoC FPGAs — Product Family Overview*, 2025. [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/system-on-chip-fpgas/polarfire-soc-fpgas.

[36] BeagleBoard.org Foundation, *BeagleV-Fire — Overview*, 2025. [Online]. Available: https://docs.beagleboard.org/boards/beaglev/fire/01-introduction.html.

[37] BeagleBoard.org Foundation, *BeagleV-Fire Front View*, 2025. [Online]. Available: https://docs.beagleboard.org/_images/bvf-front.webp.

[38] Microchip Technology Inc., *PolarFire SoC (MPFS025T) Product Page*, 2025. [Online]. Available: https://www.microchip.com/en-us/product/mpfs025t.

[39] Microchip Technology Inc., *PolarFire SoC Block Diagram from Product Overview*, 2025. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/DataSheets/PolarFire-SoC-Product-Overview.pdf.

[40] BeagleBoard.org Foundation, *BeagleV Fire CPU Core Specs*, 2025. [Online]. Available: https://www.beagleboard.org/boards/beaglev-fire.

[41] Microchip Technology Inc. "PolarFire SoC Block Diagram — Mouser Electronics." (2025), [Online]. Available: https://eu.mouser.com/new/microchip/microchip-polarfire-soc-icicle-soc/.

[42] BeagleBoard.org Foundation, *BeagleV Fire Board Documentation*, https://docs.beagleboard.org/boards/beaglev/fire/, 2025.

[43] RS Components, *BeagleV Fire — RS Components Datasheet*, https://assets.rs-online.com/v1722611049/Datasheets/9579c2e5398d0f789e8e85f82036f726.pdf, 2024.

[44] BeagleBoard.org Foundation, *Customize BeagleV-Fire Cape Gateware Using Verilog*, https://docs.beagleboard.org/boards/beaglev/fire/demos-and-tutorials/gateware/customize-cape-gateware-verilog.html, 2025.

[45] BeagleBoard.org Foundation, *How to retrieve BeagleV Fire's gateware version*, https://docs.beagleboard.org/boards/beaglev/fire/demos-and-tutorials/gateware/how-to-find-out-whats-on-the-board.html, 2025.

[46] LLVM Project, *Commit cd708029*, 2025. [Online]. Available: https://github.com/llvm/llvm-project/commit/cd708029e0b2869e80abe31ddb175f7c35361f90.

[47] RISC-V International, *RISC-V GNU Compiler Toolchain*, https://github.com/riscv-collab/riscv-gnu-toolchain.

[48] Microsoft Docs, *Install WSL | Microsoft Learn*, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/install.

[49] RISC-V Software Source, *riscv-pk: Proxy Kernel for RISC-V*, 2024. [Online]. Available: https://github.com/riscv-software-src/riscv-pk.

[50] RISC-V Software Source, *Spike: RISC-V ISA Simulator*, 2024. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim.

[51] Mobatek, *MobaXterm*, Online software, https://mobaxterm.mobatek.net/, 2024.

[52] Microsoft Corporation, *Visual Studio 2022 - Features*, https://learn.microsoft.com/en-us/visualstudio/releases/2022/release-notes, 2022.

[53] Microsoft Azure, *Azure Dev Tools for Teaching*, https://azureforeducation.microsoft.com/devtools, 2024.

[54] MSYS2 Developers, *MSYS2 Manual*, https://www.msys2.org/docs/what-is-msys2/, 2024.

[55] Popescu Vlad-Mihai, *LLVM Fork with Custom RISC-V Builtin DOT Instruction*, https://github.com/meetzaa/llvm-project, 2025.

[56] BeagleBoard, *BeagleV-Fire UART Debug Pins*, 2024. [Online]. Available: https://docs.beagleboard.org/_images/BeagleV-Fire-UART-Debug.webp.

[57] BeagleBoard Docs, *Flashing eMMC on BeagleV-Fire*, 2024. [Online]. Available: https://docs.beagleboard.org/boards/beaglev/fire/demos-and-tutorials/flashing-board.html#flashing-emmc.

Rezumat

Lucrarea de față a urmărit integrarea unui nou builtin în backend-ul compilatorului LLVM pentru arhitectura RISC-V, care realizează produsul scalar între doi vectori de întregi, optimizând astfel o operație frecvent întâlnită în calculele numerice și aplicațiile de procesare a semnalului. Motivația acestui demers a pornit din necesitatea de a înțelege în profunzime mecanismele de generare a codului într-un compilator modern și de a îmbunătăți performanțele unor operații fundamentale prin intervenții directe în procesul de selecție a instrucțiunilor.

Lucrarea a fost structurată într-o manieră top-down, începând cu analiza frontend-ului LLVM, continuând cu generarea reprezentărilor intermediare și culminând cu etapa de backend unde a fost introdus efectiv noul builtin. A fost detaliat întregul proces de implementare, de la definirea instrucțiunii în fișierele de tip .td și inserarea ei în DAG-ul de selecție, până la scrierea codului de expansiune manuală într-un context post-register allocation. S-a optat pentru o implementare directă în `RISCVExpandPseudo.cpp`, acolo unde codul este introdus în forma finală de asamblare, tocmai pentru a păstra controlul complet asupra structurii buclei și al registrilor utilizați.

Pe parcursul lucrării, au fost efectuate numeroase teste comparative între varianta scalară standard, implementările native și versiunea nouă optimizată. Testele, realizate pe placa BeagleV-Fire, au evidențiat îmbunătățiri semnificative de performanță, în special în cazul dimensiunilor mari ale vectorilor. Astfel, noul builtin a reușit să reducă timpul de execuție cu peste **5×** față de implementarea standard la nivel de optimizare O0, și a menținut performanțe competitive chiar și în regimuri O2 și O3.

Un aspect important al lucrării a fost documentarea dificultăților întâmpinate. Au fost descrise în detaliu obstacolele tehnice legate de setup-ul experimental, precum instabilitatea mediului de lucru sub Windows și soluționarea acestora printr-un dual-boot cu Linux Mint, dar și problemele hardware, cum ar fi brick-uirea plăcii BeagleV-Fire și metodele de remediere prin interfață UART cu ajutorul unui Arduino adaptat. De asemenea, au fost analizate provocările conceptuale privind alegerea locației corecte pentru expansiunea instrucțiunii în cadrul backend-ului LLVM.

Prin această lucrare s-a demonstrat nu doar o capacitate tehnică de a interveni în mod direct în pipeline-ul unui compilator complex, dar și o înțelegere aplicată a arhitecturii RISC-V, a modului de lucru cu sisteme embedded și a corelației dintre codul generat și performanța reală a aplicațiilor. Rezultatele obținute validează importanța intervențiilor la nivel de backend în obținerea unui control fin asupra codului rezultat, în special în contextul arhitecturilor emergente și al nevoii de eficiență energetică.

În perspectivă, această lucrare poate constitui baza extinderii către alte operații numerice frecvente, precum suma vectorială, multiplicarea matricială sau normele Euclidiene și chiar implementarea acestor operații direct în logica unui FPGA, în paralel cu procesorul, pentru accelerarea calculelor critice. Totodată, aceste rezultate pot fi valorificate în cadrul viitoarelor laboratoare educaționale de tip „Sisteme Încorporate", oferind studenților exemple concrete de modificare și testare a unui compilator open-source.

Abstract

This thesis aimed to integrate a new builtin function into the LLVM compiler backend for the RISC-V architecture, designed to perform a scalar dot product between two integer vectors. The motivation behind this work stemmed from the need to understand in depth the mechanisms of code generation in a modern compiler and to enhance the performance of fundamental operations by directly intervening in the instruction selection process.

The work followed a top-down structure, beginning with an analysis of the LLVM frontend, continuing with the generation of intermediate representations, and culminating in the backend phase where the builtin was effectively introduced. The entire implementation process was detailed, from defining the instruction in .td files and inserting it into the instruction selection DAG, to manually writing the expansion logic in a post-register allocation context. The expansion was implemented directly in `RISCVExpandPseudo.cpp`, allowing full control over the loop structure and register allocation strategy.

Throughout the project, several comparative tests were conducted between the standard scalar implementation, native loop unrolling, and the newly optimized builtin version. Benchmarks executed on the BeagleV-Fire board demonstrated significant performance improvements, particularly for large vector sizes. The builtin reduced execution time by over **5×** at optimization level O0 compared to traditional implementation, and maintained competitive performance even under higher optimization levels such as O2 and O3.

A key aspect of the thesis was the documentation of challenges encountered during the process. Technical obstacles related to cross-compiling under Windows, such as frequent out-of-memory errors or system crashes, were mitigated by switching to a Linux Mint dual-boot environment, which offered faster and more stable builds. Hardware-related issues, including soft-bricking of the BeagleV-Fire board due to excessive or rapid data transmission, were resolved by interfacing with the board through UART using an adapted Arduino circuit. Additionally, the conceptual challenge of choosing the correct expansion location in the LLVM backend was explored, resulting in a successful implementation.

This project not only demonstrated technical proficiency in modifying the internals of a complex open-source compiler, but also provided an applied understanding of RISC-V architecture, embedded systems integration, and the relationship between generated code and real-world performance. The results highlight the importance of backend-level interventions to achieve fine-grained control over instruction generation, particularly in the context of emerging open architectures and energy-efficient computing.

Looking ahead, this work can serve as a foundation for the optimization of other common numeric operations, such as vector summation, matrix multiplication, or Euclidean norm calculations, or even for offloading these operations onto an FPGA to perform computations in parallel with the processor. Furthermore, the implementations and insights gained can be leveraged in future academic laboratories, such as "Embedded Systems", providing students with concrete examples of real-world compiler modification and performance analysis.