

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: **CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI**
SPECIALIZAREA: **TEHNOLOGIA INFORMAȚIEI**

Implementarea unui procesor RISC-V în Verilog

LUCRARE DE DIPLOMĂ

Coordonator științific
Ş.I.dr.ing. Mircea Călin MONOR

Absolvent
Rareş-Andrei DANCĂU

Iași, 2024

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) DANCĂU RAREŞ-ANDREI,
legitimat(ă) cu CI seria _____ nr. _____, CNP _____
autorul lucrării IMPLEMENTAREA UNUI PROCESOR RISC-V ÎN VERILOG

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea IULIE 2024 a anului universitar 2023-2024, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data

Semnătura

Cuprins

Introducere.....	1
Capitolul 1. Fundamentarea teoretică și documentarea bibliografică.....	3
1.1. Domeniul și contextul abordării temei.....	3
1.2. Prezentare succintă și comparație cu realizările actuale.....	3
1.3. Analizarea tipurilor de produse existente.....	4
1.4. Limbajele utilizate în cadrul proiectului și motivația alegerii acestora.....	4
1.4.1. Verilog.....	4
1.4.2. C#.....	4
1.4.3. Python cu framework-ul Qt.....	5
1.4.4. MicroPython pentru Raspberry Pi Pico.....	5
1.5. Caracteristicile așteptate de la aplicație.....	5
1.5.1. Procesorul RISC-V pe FPGA.....	5
1.5.2. Codificatoarele.....	6
1.5.3. Emulatorul pe Raspberry Pi Pico.....	6
Capitolul 2. Proiectarea procesorului și a aplicațiilor anexe.....	8
2.1. Analiza Platformei Hardware.....	8
2.1.1. Descrierea Platformei FPGA Boolean Board.....	8
2.1.2. Implementarea Procesorului RISC-V în Verilog.....	8
2.1.2.1. Fișierul design.v.....	8
2.1.2.2. Fișierul constr.xdc.....	8
2.1.2.3. Fișierul data.mem.....	8
2.1.2.4. Fișierul instructions.mem.....	8
2.2. Proiectarea modulelor generale.....	9
2.2.1. Modulele Verilog.....	9
2.2.1.1. Modulul ClockDivider.....	9
2.2.1.2. Modulul RISCVProcessor.....	9
2.2.2. Codificator în varianta C#.....	10
2.2.2.1. Metoda buttonInstr_Click.....	11
2.2.2.2. Metoda buttonDel_click.....	11
2.2.2.3. Metoda buttonSal_Click.....	11
2.2.2.4. Metoda buttonIncFis_Click.....	11
2.2.2.5. Constructorul clasei CodificatorRISCV.....	11
2.2.2.6. Metoda buttonCod_Click.....	11
2.2.3. Codificator în varianta Python.....	11
2.2.3.1. Fișierul InterfataQt.ui.....	12
2.2.3.2. Fișierul main_window.py.....	12
2.2.3.3. Fișierul main.py.....	12
2.2.4. Emulator pe Raspberry Pi Pico.....	12
2.2.4.1. Fișierul RISC-Vpico.py.....	12
2.3. Analiza avantajelor și dezavantajelor implementării pe FPGA.....	12
2.3.1. Avantajele implementării.....	12
2.3.1.1. Flexibilitate și posibilitatea de reconfigurare.....	13
2.3.1.2. Paralelism.....	13
2.3.1.3. Consum redus de energie.....	13
2.3.1.4. Posibilitatea folosirii unei suite software solide.....	13

2.3.2. Dezavantajele implementării.....	13
2.3.2.1. Resurse limitate.....	13
2.3.2.2. Costuri ridicate pentru producția în masă.....	13
2.3.2.3. Consumul mare de energie comparativ cu integrate dedicate.....	13
2.3.2.4. Frecvența redusă a clock-ului.....	13
2.3.2.5. Necesitatea folosirii de software proprietar.....	13
2.4. Componete hardware.....	14
2.4.1. Componete hardware necesare.....	14
2.4.2. Scheme bloc.....	14
2.4.2.1. Schema bloc a modulului ClockDivider.....	14
2.4.2.2. Schema bloc simplificată a modulului RISC-VProcessor.....	14
2.4.3. Simularea componentelor hardware.....	17
2.4.3.1. Configurarea mediului de simulare.....	17
2.4.3.2. Scenarii de test pentru procesor.....	17
Capitolul 3. Implementarea procesorului și a aplicațiilor anexe.....	21
3.1. Implementarea procesorului în Verilog.....	21
3.1.1. Implementarea modulului ClockDivider.....	21
3.1.2. Implementarea modulului RISC-VProcessor.....	21
3.1.2.1. Implementarea etapei de inițializare.....	21
3.1.2.2. Instanțierea modulului ClockDivider.....	22
3.1.2.3. Verificarea frontului pozitiv pentru clk_div și a valorii semnalului reset.....	22
3.1.2.4. Pipeline verificarea valorilor pentru regiștrii bula și Jump.....	22
3.1.2.5. Pipeline fetch.....	23
3.1.2.6. Pipeline decode.....	23
3.1.2.7. Pipeline execute.....	23
3.1.2.8. Pipeline memory.....	24
3.1.2.9. Pipeline write-back.....	25
3.2. Implementarea codificatorului în varianta C#.....	26
3.2.1. Metoda buttonInstr_Click.....	26
3.2.2. Metoda buttonIncFis_Click.....	27
3.2.3. Metoda buttonDel_Click.....	28
3.2.4. Metoda buttonSal_Click.....	28
3.2.5. Constructorul clasei CodicatorRISCV.....	28
3.2.6. Metoda buttonCod_Click.....	29
3.2.7. Interfața grafică.....	32
3.3. Implementarea codificatorului în varianta Python.....	33
3.3.1. Funcția limit_32_bits.....	33
3.3.2. Constructorul clasei MainWindow.....	33
3.3.3. Metoda validare_registru.....	34
3.3.4. Metoda buttonDel_clicked.....	34
3.3.5. Metoda buttonIncFis_clicked.....	34
3.3.6. Metoda buttonSal_clicked.....	34
3.3.7. Metoda buttonInstr_clicked.....	35
3.3.8. Metoda buttonCod_clicked.....	35
3.3.9. Funcția main.....	37
3.3.10. Interfața grafică.....	37
3.4. Implementarea emulatorului pe Raspberry Pi Pico.....	38
3.4.1. Funcția sign_extend.....	38

3.4.2. Funcțiile limit_64_bits și limit_16_bits.....	38
3.4.3. Funcția set_pins_from_result.....	39
3.4.4. Constructorul clasei RISC-VProcessor.....	39
3.4.5. Metoda execute.....	39
3.4.6. Metoda run.....	41
3.4.7. Funcția main.....	42
3.5. Probleme întâmpinate și soluții de rezolvare.....	42
3.5.1. Probleme întâmpinate în timpul dezvoltării arhitecturii procesorului.....	42
3.5.1.1. Dificultatea ridicată a introducerii instrucțiunilor.....	42
3.5.1.2. Viteza de execuție a instrucțiunilor nu putea fi controlată ușor.....	42
3.5.1.3. Procesorul nu putea fi implementat din cauza dimensiunii memoriilor.....	42
3.5.1.4. Memoria RAM are celule de o dimensiune anormală.....	42
3.5.1.5. Lucrul cu memoriile se făcea în big-endian.....	42
3.5.1.6. Instrucțiunile nu erau executate în pipeline ci secvențial.....	43
3.5.2. Probleme întâmpinate în dezvoltarea codificatoarelor.....	43
3.5.2.1. Utilizatorul introduce o instrucțiune nesuportată de codificator.....	43
3.5.2.2. Utilizatorul introduce numele alternative pentru registri.....	43
3.5.2.3. Utilizatorul introduce o valoare eronată pentru registri.....	43
3.5.2.4. Utilizatorul introduce o valoare invalidă pentru valori imediate.....	43
3.5.2.5. La citirea unui fișier cu instrucțiuni nu se știe unde a apărut o problemă.....	43
Capitolul 4. Testarea procesorului și anexelor și rezultate experimentale.....	44
4.1. Punerea în funcțiune.....	44
4.1.1. Punerea în funcțiune a procesorului implementat în Verilog.....	44
4.1.2. Punerea în funcțiune pentru codificatorul implementat în C#.....	44
4.1.3. Punerea în funcțiune a codificatorului implementat în Python.....	44
4.1.4. Punerea în funcțiune a emulatorului pentru Raspberry Pi Pico.....	44
4.2. Testarea hardware și software.....	44
4.2.1. Testarea procesorului implementat în Verilog.....	44
4.2.2. Testarea codificatoarelor.....	45
4.2.3. Testarea emulatorului.....	45
4.3. Date de test.....	45
4.3.1. program_add_lw.s.....	45
4.3.2. program_counter.s.....	45
4.3.3. program_functie.s.....	45
4.3.4. program_mare.s.....	45
4.3.5. program_paritate.s.....	45
4.3.6. program_procedura.s.....	45
4.3.7. program_sll_xor.s.....	45
4.3.8. program_stocari_extrageri.s.....	46
4.3.9. program_suma.s.....	46
4.4. Rezultate experimentale.....	46
4.4.1. program_add_lw.s.....	46
4.4.2. program_counter.s.....	46
4.4.3. program_functie.s.....	46
4.4.4. program_mare.s.....	46
4.4.5. program_paritate.s.....	47
4.4.6. program_procedura.s.....	47

4.4.7. program_sll_xor.s.....	47
4.4.8. program_stocari_extrageri.s.....	47
4.4.9. program_suma.s.....	48
4.5. Utilizarea sistemului.....	48
4.5.1. Utilizarea procesorului.....	48
4.5.2. Utilizarea codificatoarelor.....	48
4.5.3. Utilizarea emulatorului de pe Raspberry Pi Pico.....	48
Concluzii.....	49
Bibliografie.....	51
Anexe.....	53
Anexa 1. Schema bloc completă a modulului RISC-VProcessor.....	53
Anexa 2. Programele de test în limbaj de asamblare RISC-V.....	63
Anexa 3. Codul complet al modulului RISC-VProcessor.....	67
Anexa 4. Codul sursă complet al clasei CodificatorRISCV.....	84
Anexa 5. Conținutul fișierului main.py al codificatorului Python.....	97
Anexa 6. Codul sursă al emulatorului pentru Raspberry Pi Pico.....	106

Lista figurilor

Figura 1.1. Boolean Board.....	6
Figura 1.2. Placa Raspberry Pi Pico.....	7
Figura 1.3. Explicarea pinilor plăci Raspberry Pi Pico.....	7
Figura 2.1: Schema bloc a modulului ClockDivider.....	14
Figura 2.2. Schema bloc simplificată a modului RISC-VProcessor.....	15
Figura 2.3. Gestionarea registrului PC.....	16
Figura 2.4. Gestionarea registrului bula.....	16
Figura 2.5. Gestionarea registrului Jump.....	17
Figura 2.6. Evoluția în cadrul pipeline-ului pentru opcode.....	17
Figura 2.7. Rezultatele pentru program_add_lw.s.....	18
Figura 2.8. Rezultatele pentru program_counter.s.....	18
Figura 2.9. Rezultatele pentru program_functie.s.....	18
Figura 2.10. Rezultatele pentru program_mare.s.....	18
Figura 2.11. Rezultatele pentru program_paritate.s.....	19
Figura 2.12. Rezultatele pentru program_procedura.s.....	19
Figura 2.13. Rezultatele pentru program_sll_xor.s.....	19
Figura 2.14. Rezultatele pentru program_stocari_extrageri.s.....	19
Figura 2.15. Rezultatele pentru program_suma.s.....	20
Figura 3.1. Codificator RISC-V C# aflat în execuție.....	32
Figura 3.2. Codificator C# cu elemente interactive etichetate.....	33
Figura 3.3. Execuție pe Linux Mint a codificatorului Python.....	37
Figura 3.4. Elemente interactive codificator Python etichetate.....	38
Figura A.1. Schema bloc a modulului RISC-VProcessor #a.....	53
Figura A.2. Schema bloc a modulului RISC-VProcessor #b.....	54
Figura A.3. Schema bloc a modulului RISC-VProcessor #c.....	55
Figura A.4. Schema bloc a modulului RISC-VProcessor #d.....	56
Figura A.5. Schema bloc a modulului RISC-VProcessor #e.....	57
Figura A.6. Schema bloc a modulului RISC-VProcessor #f.....	58
Figura A.7. Schema bloc a modulului RISC-VProcessor #g.....	59
Figura A.8. Schema bloc a modulului RISC-VProcessor #h.....	60
Figura A.9. Schema bloc a modulului RISC-VProcessor #i.....	61
Figura A.10. Schema bloc a modulului RISC-VProcessor #j.....	62

Implementarea unui procesor RISC-V în Verilog

Rareş-Andrei Dancău

Rezumat

Această lucrare descrie implementarea și testarea unui procesor RISC-V pe o platformă FPGA, dezvoltarea unor codificatoare de instrucțiuni în C# și Python, dar și realizarea unui emulator pe o placă Raspberry Pi Pico. Arhitectura RISC-V, cunoscută pentru simplitate și flexibilitate, este folosită din ce în ce mai mult atât în mediul academic, cât și în cel industrial mulțumită caracterului său bazat pe conceptul de sursă deschisă.

Proiectul a implicat utilizarea platformei FPGA Boolean Board, ce oferă un mediu robust și versatil pentru dezvoltarea și testarea circuitelor digitale. Implementarea procesorului a fost făcută în Verilog, un limbaj de descriere hardware popular, alături de instrumente de dezvoltare din cadrul suitei Vivado 2023.1 de la AMD. Codificatoarele de instrucțiuni au fost dezvoltate în C# cu WinForms și Python cu PyQt, oferind interfețe grafice intuitive pentru generarea de cod executabil pe procesor din fișiere cu instrucțiuni în asamblare RISC-V. Emulatorul, implementat pe un Raspberry Pi Pico folosind MicroPython, permite execuția și testarea programelor RISC-V într-un mediu diferit de FPGA.

Metodologia utilizată a inclus proiectarea și simularea componentelor hardware, dezvoltarea și testarea codificatoarelor de instrucțiuni și a emulatorului, dar și integrarea procesorului și a codificatoarelor într-un sistem coerent. Lucrarea prezintă avantajele și dezavantajele instrumentelor și metodelor alese, proiectarea procesorului, a codificatoarelor și a emulatorului, implementarea și testarea acestora, în urma cărora au fost obținute rezultate ce vor fi folosite la evaluarea îndeplinirii obiectivelor lucrării.

Principalele realizări ale proiectului includ o implementare funcțională a unui procesor RISC-V, codificatoare de instrucțiuni eficiente și un emulator capabil să execute programe RISC-V și să afișeze rezultatele acestora folosind leduri. Concluziile evidențiază succesul implementării, oferind și direcții de dezvoltare viitoare, cum ar fi extinderea setului de instrucțiuni ce pot fi executate, eficientizarea executării instrucțiunilor în pipeline și dezvoltarea unui sistem de operare minimal pentru procesorul implementat.

Lucrarea confirmă aplicabilitatea și avantajele utilizării arhitecturii RISC-V și a platformei FPGA în dezvoltarea de soluții academice, contribuind la extinderea cunoștințelor și abilităților în domeniul arhitecturii calculatoarelor și sistemelor încorporate. Implementarea pe FPGA permite studenților și cercetătorilor să poată experimenta, putând modifica proiectarea hardware a procesorului, dar și să înțeleagă principiile de funcționare a procesoarelor și să poată aplica optimizări și extensii ale setului de instrucțiuni.

Prin codificatoare și emulator, lucrarea demonstrează simplificarea procesului de dezvoltare și testare al programelor RISC-V, aceste instrumente putând fi utilizate de studenți ce vor să aprofundeze cunoștințe din domeniul arhitecturii calculatoarelor, fără a fi nevoie de acces la hardware specializat sau costisitor.

Introducere

În ultimii ani, procesul de dezvoltare și implementare al arhitecturilor de procesoare a evoluat într-un mod semnificativ, în special în contextul cerințelor de eficiență, flexibilitate și performanță necesare în cadrul diverselor domenii tehnologice. Una dintre arhitecturile ce doresc să respecte aceste principii, RISC-V, se distinge ca fiind o arhitectură deschisă, modulară și scalabilă oferind o alternativă la arhitecturile tradiționale de procesoare, bazate pe seturi complexe de instrucțiuni.

În cadrul proiectului curent se urmărește în principal implementarea unui procesor RISC-V, pe un FPGA, ce poate executa un număr considerabil de instrucțiuni din standardul RV64I[1], dar și dezvoltarea a două codificatoare de instrucțiuni, un codificator fiind implementat în C#, iar celălalt în limbajul Python. În cadrul lucrării se mai detaliază și implementarea unui emulator pe o placă Raspberry Pi Pico ce va putea executa aceleași instrucțiuni ca procesorul implementat.

Motivația alegerii temei

Principala motivație ce a dus la alegerea acestei teme a fost dorința de a analiza această arhitectură în curs de dezvoltare și de a putea implementa un procesor într-un mediu academic, folosind FPGA. Implementarea unui astfel de procesor pe un FPGA permite aprofundarea studiului arhitecturii și depășirea unor provocări ce pot apărea în cadrul procesului de dezvoltare.

Dezvoltarea codificatoarelor de instrucțiuni a susținut procesul de testare al procesorului, permitând transformarea instrucțiunilor din asamblare RISC-V direct în reprezentarea hexazecimală a acestora, ce poate fi executată direct pe procesor.

Dezvoltarea emulatorului a fost motivată de dezideratul investigării unei alte modalități de implementare a unui procesor, elaborându-se o implementare folosind un limbaj de programare de nivel înalt în comparație cu limbajului de descriere hardware Verilog.

Relevanța și Contextul Temei Alese

Arhitectura RISC-V a devenit din ce în ce mai populară nu doar din natura sa deschisă ci și din capacitatea de a o putea personaliza pentru diverse aplicații, de la sisteme încorporate la sisteme de calcul de înaltă performanță. În contextul cerinței de soluții flexibile și eficiente energetic, RISC-V este o alternativă bună la arhitecturi bazate pe standarde închise.

Proiectul urmărește aceste cerințe de flexibilitate și eficiență, implementând un procesor bazat pe RISC-V. De asemenea, au fost implementate aplicații ce pot contribui la dezvoltarea domeniului, familiarizând diverși utilizatori cu un nou mediu de dezvoltare, iar în urma acesteia, utilizatorii pot continua direcția de dezvoltare datorită modului de gestionare deschis al arhitecturii.

Obiectivele lucrării

Prezenta lucrare de licență își propune să implementeze câteva obiective semnificative:

- Implementarea unui procesor RISC-V în limbajul de descriere hardware Verilog, încărcarea acestuia pe un FPGA(Boolean Board) și executarea câtorva programe de test.
- Dezvoltarea unui codificator de instrucțiuni în C# pentru a facilita generarea de cod executabil pe procesor din fișiere cu instrucțiuni în asamblare RV64I.
- Realizarea unui codificator echivalent cu cel anterior folosind limbajul Python pentru a oferi o alternativă multiplatformă.
- Crearea unui emulator pentru procesorul RISC-V ce va fi executat pe un Raspberry Pi

Pico pentru a explora un nou mod de implementare al unui procesor și pentru a permite execuția și testarea programelor de test într-un mediu diferit de FPGA.

Metodologia și instrumentele utilizate

Pentru a implementa aceste obiective, în cadrul proiectului au fost folosite următoarele tehnologii:

- Verilog: Utilizat pentru descrierea hardware a procesorului RISC-V.
- Boolean Board: FPGA-ul utilizat pentru implementarea și testarea procesorului.
- Vivado 2023.1: Suite CAD de la AMD folosită pentru sinteză, implementare și programarea FPGA-ului.
- C#: Limbaj de programare folosit pentru prima variantă cu interfață grafică a codificatorului de instrucțiuni.
- Python și framework-ul Qt: Utilizate pentru a realiza codificatorul de instrucțiuni multiplatformă cu interfață grafică.
- Raspberry Pi Pico: Placa, pe care este executat emulatorul procesorului RISC-V, are ca circuit integrat principal un microcontroler RP2040.
- MicroPython: Limbajul utilizat pentru programarea emulatorului pe Raspberry Pi Pico.

Structura lucrării

Lucrarea se structurează pe mai multe capitoare, fiecare prezentând aspecte esențiale ale proiectului:

- Capitolul 1. Fundamentarea teoretică și documentarea bibliografică
 - Expune aspecte despre domeniul și abordarea temei, precum și o comparație a proiectului actual cu o serie de proiecte deja existente.
 - De asemenea, se detaliază limbajele utilizate și caracteristicile necesare ale proiectului.
- Capitolul 2. Proiectarea procesorului și a aplicațiilor anexe
 - Se detaliază platforma FPGA pe care va fi implementat procesorul, structurarea elementelor procesorului, a codificatoarelor și a emulatorului.
 - Se mai prezintă avantaje și dezavantaje ale implementării pe FPGA și structura hardware a proiectului.
- Capitolul 3. Implementarea procesorului și a aplicațiilor anexe
 - Sunt expuse detalii practice despre implementarea procesorului și a modulelor din care acesta este format. Sunt prezentate într-un mod similar codificatoarele și emulatorul.
 - În cadrul acestui capitol se mai ilustrează părțile fundamentale ale codului sursă pentru aplicațiile amintite, problemele întâmpinate în timpul dezvoltării aplicațiilor, alături de modul în care au fost rezolvate.
- Capitolul 4. Testarea procesorului și anexelor și rezultate experimentale
 - Se explică modul de punere în funcțiune și modul de folosire al procesorului, codificatoarelor și emulatorului, alături de prezentarea procesului de testare și a rezultatelor obținute în urma acestui proces.
- Concluzii
 - Se reflectă asupra îndeplinirii obiectivelor lucrării, dar se expun și posibile direcții de dezvoltare ale proiectului, cum ar fi extinderea setului de instrucțiuni suportate și implementarea unui sistem de operare minimal ce poate fi executat pe procesor.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

1.1. Domeniul și contextul abordării temei

Datorită caracterului său deschis, dar și flexibilității pe care o oferă, arhitectura RISC-V are un impact din ce în ce mai mare asupra viitorului implementării procesoarelor. Arhitectura RISC-V se bazează pe un set de instrucțiuni(ISA) de tip RISC ce se remarcă prin simplitate, eficiență, dar și prin extensibilitate. Inițial dezvoltată la Universitatea Berkeley din California, arhitectura RISC-V este acum susținută de o largă comunitate de organizații și dezvoltatori.

Contextul abordării temei este determinat de dorința de a implementa o platformă educațională care să permită cercetarea, experimentarea și dezvoltarea de noi tehnologii și aplicații. Implementarea procesorului, a codificatoarelor și a emulatorului au dus la înțelegerea și explorarea arhitecturii RISC-V, dar și a limbajelor folosite în implementarea aplicațiilor.

1.2. Prezentare succintă și comparație cu realizările actuale

Arhitectura RISC-V este explorată de mai multe proiecte[2] și prin urmare este folosită în mai multe aplicații. Printre aceste proiecte reprezentative, se disting:

- E203[3]: Un procesor cu două stagii de pipeline ce are ca scop consumul mic de putere și dimensiunea mică a implementării.
- BOOM[4]: Un procesor sintetizabil creat în limbajul Chisel la Universitatea Berkeley din California, creat în scopuri de cercetare.
- CVA6[5]: Un procesor cu șase stagii de pipeline, pe 64 de biți. Implementează extensiile I, M, A și C ale setului de instrucțiuni.
- Kronos[6]: Un procesor cu trei stagii de pipeline destinat implementărilor pe FPGA.
- PicoRV32[7]: Un procesor ce implementează setul de instrucțiuni RV32I, alături de extensiile M și C. Poate fi folosit ca procesor secundar în cadrul altor proiecte.
- Spike[8]: Un simulator RISC-V ce implementează un model funcțional al unor plăci ce folosesc procesoare RISC-V.
- FireSim[9]: O platformă de simulare hardware, accelerată folosind FPGA-uri, ce permite validarea și depanarea de hardware RTL.
- Ripes[10]: Este un simulator de arhitecturi de calculatoare ce permite editarea codurilor în asamblare executată pe procesor. Este bazat pe arhitectura RISC-V.
- RISC-V-Linux[11]: Proiect în cadrul căruia se face portarea sistemului de operare Fedora 29 GNOME pe placă HiFive Unleashed ce folosește un procesor bazat pe RISC-V.
- SiFive: O companie ce dezvoltă și comercializează procesoare RISC-V utilizate în diverse aplicații industriale.
- Parallel Ultra Low Power[12]: Un proiect ce explorează și dezvoltă arhitectura RISC-V. Scopul inițial a fost implementarea dispozitivelor de putere mică, dar s-a extins la dispozitive cu mai multe nuclee.

Comparativ, procesorul implementat se axează pe suportul instrucțiunilor RV64I, iar utilizarea unei platforme FPGA, oferă ideea de flexibilitate în experimentarea cu diverse optimizări și configurații hardware. Dezvoltarea codificatoarelor și a emulatorului duc la îmbunătățirea accesibilității pentru utilizatori, adăugând un nivel suplimentar de utilitate în procesul de dezvoltare și testare.

1.3. Analizarea tipurilor de produse existente

Pe piață sunt disponibile o varietate de produse și aplicații bazate pe RISC-V, fiecare utilizând diverse tehnologii în cadrul implementării:

- **Microprocesoare încorporate:** Sunt fabricate de companii cum ar fi SiFive, fiind utilizate în aplicații industriale, IoT și automotive. Acestea pot fi implementate folosind Verilog din punct de vedere al design-ului hardware, iar pentru programarea acestora se pot folosi diverse medii de dezvoltare.
- **Platforme educaționale:** În cadrul cursurilor de arhitectura calculatoarelor de la diverse universități și instituții de învățământ sunt folosite implementări RISC-V pentru a permite studierea conceptelor într-un mod practic.
- **Sisteme de dezvoltare:** Platforme cum ar fi PULP și BOOM oferă un mediu de dezvoltare flexibil, utilizând limbaje de descriere hardware precum Chisel, și suportă diverse fluxuri de proiectare și simulare.

Tehnologiile folosite pentru implementarea acestora pot varia de la limbaje de descriere hardware tradiționale cum ar fi VHDL și Verilog până la abordări bazate pe limbaje de nivel înalt, cum ar fi Chisel. Aceste tehnologii sunt susținute de suite software CAD, cum ar fi Vivado.

1.4. Limbajele utilizate în cadrul proiectului și motivația alegerii acestora

În cadrul proiectului au fost folosite mai multe limbaje de programare și limbajul de descriere hardware Verilog. În cadrul sub-capitolelor următoare vor fi prezentate motivația alegerii acestora alături de avantaje, dar și dezavantaje ale acestor decizii.

1.4.1. Verilog

Este unul dintre cele mai folosite limbaje de descriere hardware(HDL) ce descrie funcționalitatea circuitelor digitale. Un instrument de sinteză va traduce aceste descrieri hardware în implementarea de circuite combinaționale și secvențiale. Verilog este bazat pe limbajul C, comparativ cu limbajul VHDL implementat în Ada și Pascal.[13]

Acesta a fost ales pentru implementarea procesorului datorită capacitatea sale de a descrie circuite complexe la nivel structural și comportamental.

Verilog oferă mare control asupra design-ului hardware, dar și flexibilitate, putând fi utilizat de majoritatea uneltelor de simulare și sinteză.

Totuși, depanarea circuitelor implementate poate fi dificilă în comparație cu limbajele de nivel înalt, iar tratarea acestuia de evenimente la existența mai multor blocuri procedurale nu este intuitivă pentru programatorii ce folosesc limbaje de nivel înalt.

1.4.2. C#

Este cel mai popular limbaj de programare pentru platforma .NET. Programele scrise în C#, fără a folosi WinForms, pot fi executate pe diverse dispozitive, cum ar fi dispozitivele IoT, dar și telefoane mobile, laptop-uri și desktop-uri. Este bazat pe principiile programării orientate obiect, dar include și facilități din alte paradigme de programare, cum ar fi programarea funcțională.[14]

A fost ales pentru simplitatea și eficiența sa în dezvoltarea aplicațiilor desktop cu interfață grafică. Platforma .NET oferă un număr considerabil de biblioteci și unelte ce contribuie la dezvoltarea mai rapidă a aplicațiilor desktop.

Ca avantaje ale limbajului pot fi considerate suportul destul de bun pentru interfețe

grafice, utilizarea în mod mai facil comparativ cu alte limbaje de programare și performanța acceptabilă.

Ca dezavantaje ar fi limitarea utilizării codificatorului doar pe un sistem de operare, ca urmare a folosirii pachetul WinForms pentru a implementa interfață grafică, dar și utilizarea mai rară a limbajului într-un mediu academic, comparativ cu alte limbaje.

1.4.3. Python cu framework-ul Qt

Python este un limbaj orientat obiect comparabil cu Perl, Ruby, Scheme sau Java. Folosește o sintaxă elegantă ce îmbunătățește înțelegerea codului de către programator. Un limbaj ușor de folosit, este ideal pentru prototipare și alte cerințe ad-hoc de programare.[15]

Framework-ul Qt[16] conține o listă de clase intuitive și modulare din biblioteci C++ și conține API-uri pentru a simplifica dezvoltarea aplicațiilor cu interfață grafică. Dintre avantajele folosirii Qt, sunt considerate ușurința înțelegerei codului, mențenanța facilă, posibilitatea reutilizării codului și utilizarea pe mai multe platforme. PyQt, folosit la dezvoltarea codificatorului, permite utilizarea de către programe scrise în Python a framework-ului Qt. Folosind PyQt[17] pot fi create aplicații complexe cu interfață grafică și utilizate semnalele transmise de obiecte la interacțiunea acestora cu utilizatorul.

Printre avantajele limbajului Python se remarcă: ușurința în utilizare, sintaxa clară și concisă și suportul multiplatformă.

Dezavantajele constau în performanța mai redusă, datorită utilizării unui interpretor, comparativ cu programe compilate scrise în alte limbaje, iar gestiunea resurselor pentru aplicațiile intensive nu este optimă.

1.4.4. MicroPython pentru Raspberry Pi Pico

MicroPython[18] este o implementare eficientă și cu un consum mic de resurse a limbajului Python 3 ce include o parte din facilitățile bibliotecii standard Python și este destinat execuției pe microcontrolere sau în medii cu resurse limitate. MicroPython dorește să fie cât mai compatibil cu Python obișnuit pentru a facilita transferul codului de pe desktop pe microcontroler sau sistem încorporat.

Avantajele ce au dus la alegerea acestuia au fost: ușurința de utilizare, suportul excelent pentru Raspberry Pi Pico și posibilitatea dezvoltării rapide a emulatorului.

Dezavantajele includ performanța limitată comparativ cu C sau C++ și numărul mai redus de optimizări pentru aplicații din punct de vedere al performanței.

1.5. Caracteristicile așteptate de la aplicație

Pentru a îndeplini obiectivele lucrării, au fost stabilite următoarele specificații pentru fiecare componentă a proiectului:

1.5.1. Procesorul RISC-V pe FPGA

- Implementarea unui set de instrucțiuni de bază (RV64I) care să permită execuția programelor de test.
- Flexibilitatea adăugării de noi instrucțiuni.
- Folosirea eficientă a resurselor plăcii Boolean. Platforma FPGA poate fi reprezentată în Figura 1.1.

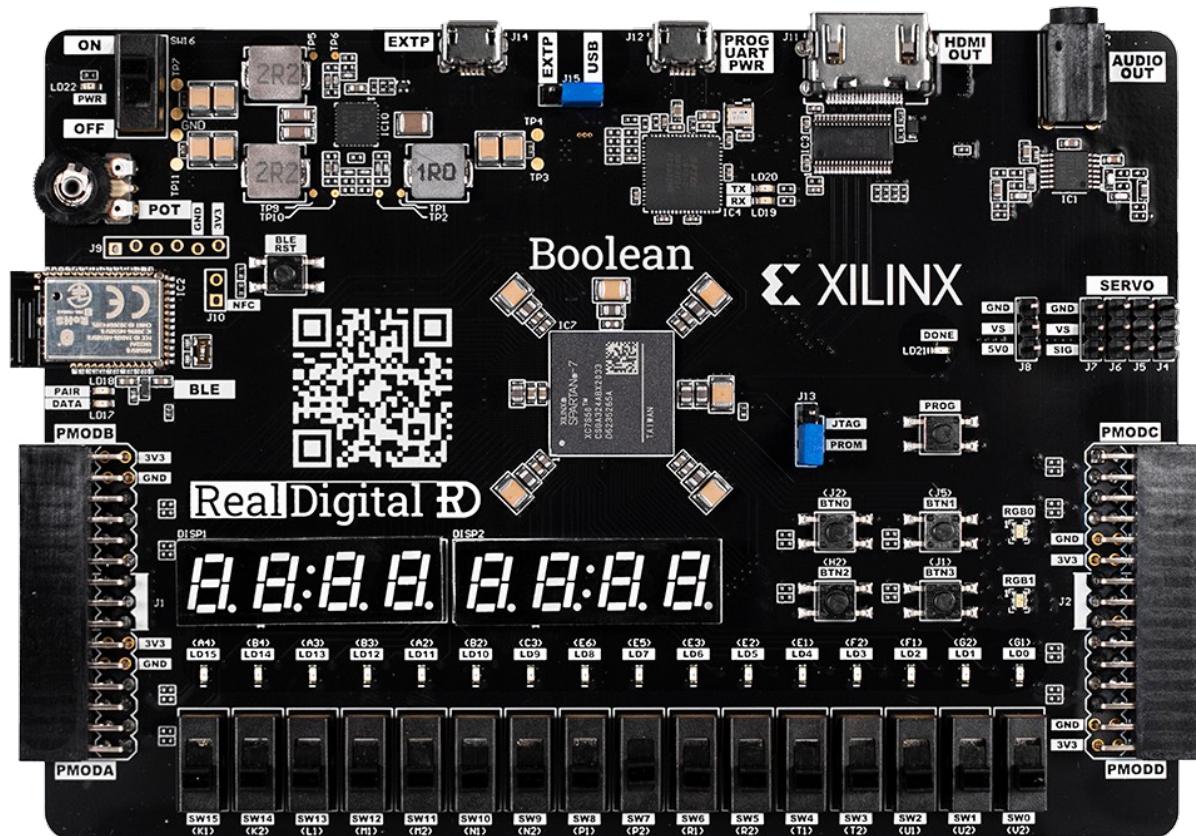


Figura 1.1. Boolean Board
[19]

1.5.2. Codificatoarele

- Codificatorul scris în C# trebuie să aibă o interfață intuitivă pentru a genera cod executabil din instrucțiuni în asamblare RISC-V.
- Codificatorul scris în Python, dezvoltat cu Qt, trebuie să poată fi executat pe mai multe sisteme de operare, să fie ușor de utilizat și să ofere aceleași funcționalități oferite de codificatorul C#.

1.5.3. Emulatorul pe Raspberry Pi Pico

- Emulatorul trebuie să poată executa și simula corect instrucțiunile RISC-V, oferind un mediu de testare facil.
- Performanța emulatorului trebuie să fie suficient de bună pentru a permite dezvoltarea și testarea eficientă a aplicațiilor.

Placa Raspberry Pi Pico poate fi identificată în Figura 1.2. Semnificația pinilor plăcii poate fi vizualizată în Figura 1.3.

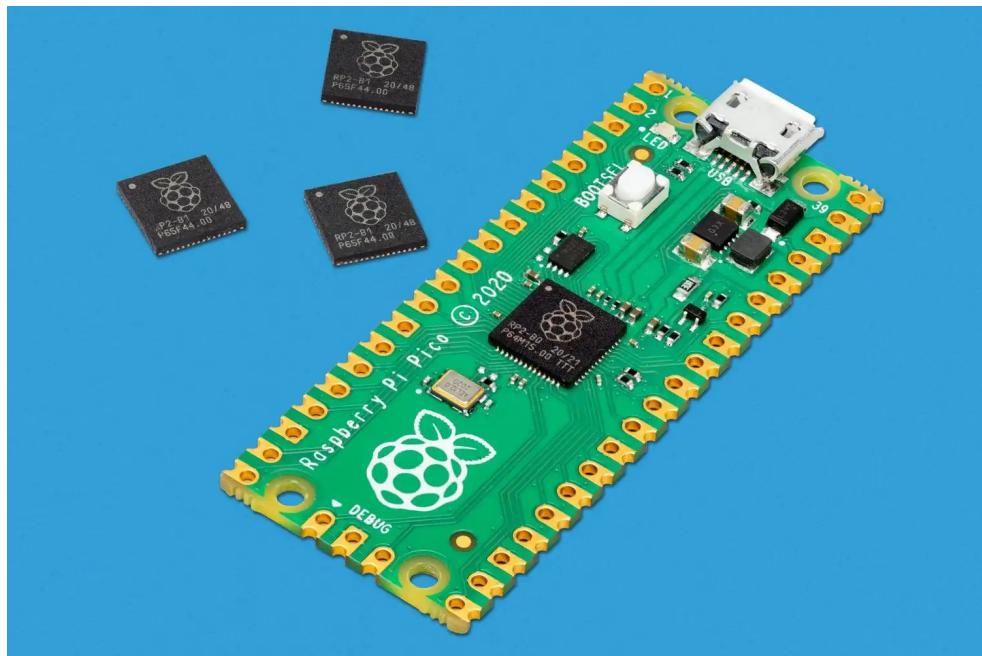


Figura 1.2. Placa Raspberry Pi Pico
[20]

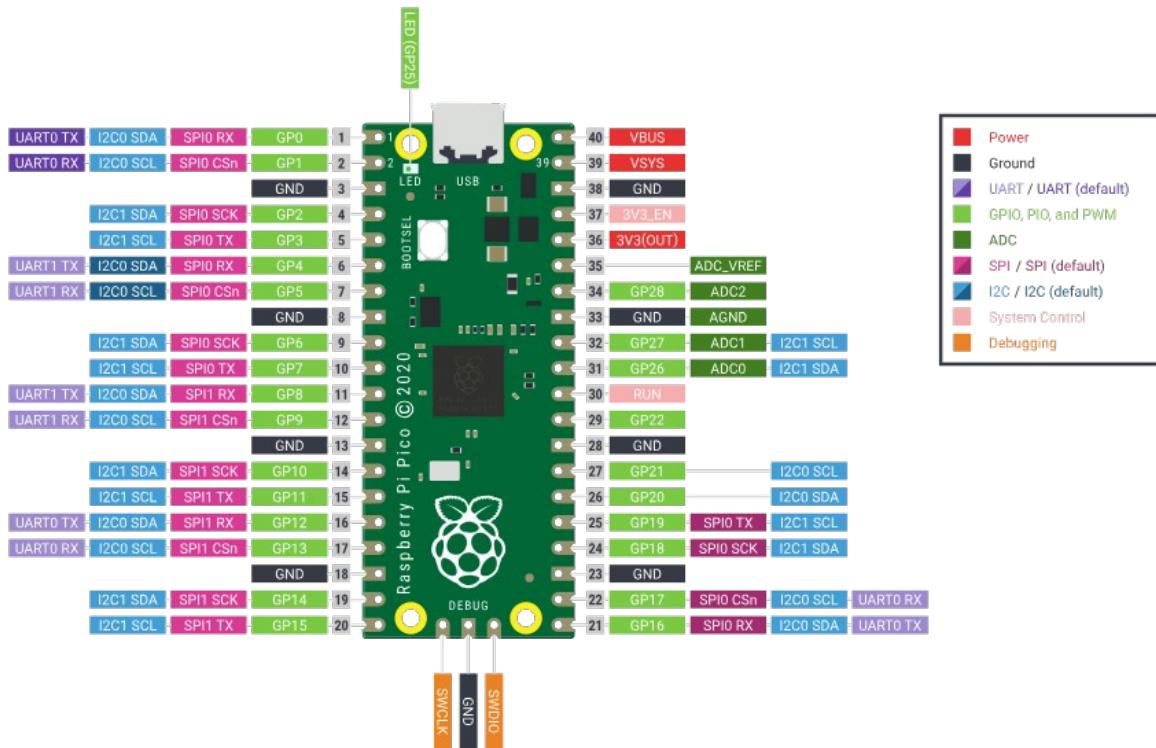


Figura 1.3. Explicarea pinilor plăci Raspberry Pi Pico
[21]

Aceste specificații au ghidat dezvoltarea și implementarea proiectului, ținând cont de faptul că rezultatele obținute respectă cerințele stabilite și oferă o platformă solidă pentru viitoare dezvoltări și cercetări.

Capitolul 2. Proiectarea procesorului și a aplicațiilor anexe

Pentru a implementa procesorul RISC-V s-a decis folosirea limbajului de descriere hardware Verilog, alegeră motivată de faptul că se abstractizează la nivel înalt proiectarea hardware și limbajul facilitează descrierea, simularea și executarea pe FPGA a circuitelor digitale complexe.

2.1. Analiza Platformei Hardware

2.1.1. Descrierea Platformei FPGA Boolean Board

Placa Boolean este o platformă bazată pe FPGA destinată proiectelor academice din domeniul calculatoarelor. Dispune de un număr mare de componente prin care se face interacțiunea cu utilizatorul uman, cum ar fi 16 manete, 16 leduri, patru butoane, ecran de opt cifre cu șapte segmente și două leduri RGB. Inima platformei este circuitul integrat Xilinx Spartan-7 XC7S50-CSGA324, ce face parte din familia Spartan-7 cunoscută pentru performanță excelentă pe watt[22], dar și dimensiunile fizice reduse. Placa poate fi programată folosind suita software Vivado 2023.1 de la AMD, ce permite utilizarea limbajelor hardware Verilog și VHDL. Conectarea la placă pentru programare se face printr-un cablu micro USB.

În cadrul proiectului sunt folosite mai multe facilități ale Boolean Board. Maneta V2 este folosită pentru a transmite semnalul reset, switch-ul J2 controlează semnalul CLK_BUTT din procesor, pinul F14 reprezintă semnalul clock intern al plăcii și este mapat la registrul clk din modulul RISC-VProcessor. Mai sunt folosite ledurile plăcuței, numite L0-L15, pentru a afișa rezultatele instrucțiunilor.

2.1.2. Implementarea Procesorului RISC-V în Verilog

Procesorul implementat se împarte pe mai multe fișiere, ce vor fi descrise în continuare.

2.1.2.1. Fișierul design.v

Conține implementarea efectivă a modulelor procesorului. Aceste module vor fi detaliate în 2.2. Proiectarea modulelor generale.

2.1.2.2. Fișierul constr.xdc

Conține instrucțiuni de mapare între semnalele din cadrul design.v și componentele plăcuței. În cadrul acestuia s-a făcut mapare semnalului reset la comutatorul V2, semnalul clk la pinul F14, semnalul CLK_BUTT la maneta J2 și biții semnalului result la ledurile L0-L15, numite intern G1 până la A4.

2.1.2.3. Fișierul data.mem

Conține valorile folosite la inițializarea memoriei RAM folosită de procesor. În mod implicit, conține 129 de valori 0 pe 64 de biți.

2.1.2.4. Fișierul instructions.mem

Conține octetii aferenți instrucțiunilor ce pot fi executate de către procesor și vor fi plasați în cadrul memoriei ROM. Acest fișier trebuie modificat în funcție de programul executat.

2.2. Proiectarea modulelor generale

2.2.1. Modulele Verilog

În cadrul fișierului design.v amintit în 2.1.2.1. Fișierul design.v, se află două module.

2.2.1.1. Modulul ClockDivider

Modulul ClockDivider are ca intrări semnalul clock și semnalul CLK_BUTT. Ca ieșire, modulul are semnalul clk_out, inițializat cu valoarea zero. În cadrul modulului se va stabili valoarea următoarea valoare a semnalului de ieșire clk_out, în funcție de starea anterioară și valorile pentru clk și CLK_BUTT. Schema bloc a acestui modul este disponibilă în 2.4.2.1. Schema bloc a modulului ClockDivider.

2.2.1.2. Modulul RISCVProcessor

Acest modul îl instanțiază pe precedentul și primește ca intrare semnalul clock de pe pinul F14, semnalul CLK_BUTT de la comutatorul J2 și semnalul reset de la maneta V2. Ca ieșire are registrul result ce va fi afișat la leduri. În cadrul acestui modul sunt definiți mai mulți registri și variabile.

Registrul bula este pe doi biți și stochează o valoare ce reprezintă numărul de perioade de ceas în care nu se continuă executarea etapelor fetch, decode și execute în urma apariției unui hazard de date în pipeline.

Registrul pc este pe 64 de biți și stochează adresa din cadrul memoriei ROM de la care se face fetch următoarei instrucțiuni.

Registrul pc_jump pe 64 de biți stochează o valoare ce va fi introdusă în registrul PC în urma unui salt.

Registrii reg_e și reg_m stochează valori intermediare executării instrucțiunilor.

Registrul jumpAddress_e pe 16 biți stochează ultimii 16 biți ai valorii din pc_jump.

Registrul jumpAddress_m pe 16 biți stochează valoarea primită de la registrul jumpAddress_e și în cazul unei instrucțiuni de salt, această valoare va fi transmisă în registrul de ieșire result.

Registrul Jump reprezintă un flag ce determină dacă instrucțiunea executată anterior în pipeline duce la un salt.

Vectorul de registrii registers conține cele 32 de registri specifici arhitecturii RISC-V.

Memoria dataMemory este memoria RAM folosită de procesor și este utilizată mai ales de instrucțiunile de stocare și extragere din memorie.

Memoria memory este memoria ROM utilizată de procesor și stochează octetii aferenți instrucțiunilor de executat.

Registrii instruction_f, instruction_d, instruction_e, instruction_m, instruction_wb pe 32 de biți stochează octetii instrucțiunii aflate la stagiul respectiv în pipeline.

Registrii opcode_d, opcode_e, opcode_m, opcode_wb pe șapte biți stochează valoarea opcode-ului instrucțiunii aflate la stagiul respectiv în pipeline.

Registrii funct3_d, funct3_e, funct3_m, funct3_wb pe trei biți stochează valoarea funct3-ului instrucțiunii aflate la stagiul respectiv în pipeline.

Registrii funct7_d, funct7_e, funct7_m, funct7_wb pe șapte biți stochează valoarea funct7-ului instrucțiunii aflate la stagiul respectiv în pipeline.

Registrii imm12_d, imm12_e, imm12_m pe 12 biți stochează o valoare numerică a instrucțiunii aflate la stagiul respectiv în pipeline.

Registrii shamt_d, shamt_e, shamt_m pe șase biți stochează o valoare numerică a

instrucțiunii aflate la stagiul respectiv în pipeline și este folosită de instrucțiunile de shiftare pe biți.

Regiștrii rs1_d, rs1_e, rs1_m pe cinci biți stochează indexul primului registru sursă al instrucțiunii aflate la stagiul respectiv în pipeline.

Regiștrii rs2_d, rs2_e, rs2_m pe cinci biți stochează indexul celui de-al doilea registru sursă al instrucțiunii aflate la stagiul respectiv în pipeline.

Regiștrii rd_d, rd_e, rd_m, rd_wb pe cinci biți stochează indexul registrului destinație al instrucțiunii aflate la stagiul respectiv în pipeline.

Variabila de tip wire clk_div conține semnalul clock de referință al procesorului și reprezintă ieșirea modulului ClockDivider.

În cadrul modulului au loc mai multe etape, cum ar fi etapa de inițializare, instanțierea modulului ClockDivider, blocul în care se testează frontul pozitiv al semnalului clk_div și etapa de pipeline propriu-zisă.

În etapa de inițializare se inițializează cu 0 registrul de ieșire result, dar și majoritatea regiștrilor interiori modulului. Singurele excepții de la inițializarea cu 0 sunt registrul x2(sp) ce este inițializat cu valoarea 128 pentru a lucra cu stiva în mod obișnuit, regiștrii instruction_f și instruction_d inițializați cu valori explicite pentru a facilita testarea procesorului și memoriile RAM și ROM inițializate din fișierele corespunzătoare.

După această etapă are loc instanțierea modulului ClockDivider și implicit, actualizarea valorii din clk_div cu ieșirea din modul.

Apoi se verifică dacă clk_div a ajuns într-un front pozitiv și dacă da, se verifică valoarea semnalului reset. Dacă semnalul reset este activ, atunci se reinicializează toți regiștrii, cu excepția memoriei ROM. Dacă semnalul reset nu este activ, atunci se trece mai departe în etapa de pipeline propriu-zisă.

În cadrul etapei de pipeline, întâi se verifică valorile semnalelor bula și Jump. Dacă semnalul bula este nenul și semnalul Jump este 0, atunci se decrementează semnalul bula. Altfel, dacă semnalul Jump este activ, atunci se dezactivează acest semnal și valoarea din pc_jump este transmisă în pc la următorul ciclu de clock. Dacă aceste două semnale nu sunt active, se trece în sub-etapa fetch și se extrage în instruction_f valoarea din memoria ROM aferentă instrucțiunii curente. În cadrul sub-etapei decode se decodifică instrucțiunea extrasă în fetch și se modifică valoarea registrului bula în cazul în care apar hazarduri de date. Apoi, dacă semnalele bula și Jump nu sunt active, se intră în sub-etapa execute ce are ca scop executarea instrucțiunilor ce nu folosesc memoria și plasarea rezultatului în registru reg_e. În cadrul sub-etapei memory se lucrează cu dataMemory pentru instrucțiunile de stocare și extragere, iar rezultatul operației este plasat în registru reg_m. Pentru toate celelalte instrucțiuni, doar se transmite în reg_m valoarea din reg_e. În ultima sub-etapă a pipeline-ului se scrie valoarea din reg_m în registru destinație pentru instrucțiunile aferente și valoarea din reg_m este trunchiată la 16 biți și scrisă în registru de ieșire result. Dacă se introduc operații invalide pentru procesor, valoarea lui result va fi 0.

O schemă bloc simplificată a acestui modul poate fi vizualizată în 2.4.2.2. Schema bloc simplificată a modulului RISC-VProcessor.

2.2.2. Codificator în varianta C#

În cadrul evoluției procesorului, a fost necesară crearea unui codificator pentru instrucțiuni cu scopul de a facilita procesul de creare a fișierelor executabile din fișierele cu instrucțiuni în asamblare RISC-V. De aceea, a fost creat acest codificator cu interfață grafică, bazat pe clasa Form din System.Windows.Forms. Programul conține o clasă CodificatorRISCV ce moștenește Form și conține mai multe variabile și metode. Interfața grafică cu utilizatorul va fi descrisă în 3.2.7. Interfața grafică.

Din punct de vedere al variabilelor, clasa CodicatorRISCV conține mai multe liste cu numele instrucțiunilor și un dicționar ce stocă nume alternative pentru regiștri. Metodele clasei sunt prezentate în următoarele subcapitole.

2.2.2.1. Metoda buttonInstr_Click

Este apelată la apăsarea butonului de salvare a instrucțiunilor în fișier instruction. Are ca rol salvarea instrucțiunilor din textBoxInput în format little endian hexazecimal. Va deschide un dialog de interacțiune cu utilizatorul și va cere un nume pentru fișierul în care va salva rezultatul. Va parcurge fiecare instrucțiune codificată și vizibilă în textBoxInput și va salva cei patru octeți ai instrucțiunii curente câte unul pe linie în ordine inversă. După parcurgerea tuturor instrucțiunilor, rezultatul este scris în fișierul introdus de utilizator.

2.2.2.2. Metoda buttonDel_click

Este metoda apelată atunci când se dorește eliminarea instrucțiunilor existente în textBoxInput. Are ca efect eliminarea conținutului din textBoxInput.

2.2.2.3. Metoda buttonSal_Click

Este apelată atunci când utilizatorul apasă pe butonul de salvare în fișier input. Fișierul se numește input pentru că în cadrul procesului de dezvoltare erau salvate codificările instrucțiunilor în stânga liniilor și în dreapta era forma în asamblare a instrucțiunii, iar folosind un script Python erau extrași octeții și salvați în fișier instruction. Instrucțiunile sunt salvate în formatul în care apar în textBoxInput, după ce utilizatorul a închis cu succes dialogul de salvare al fișierului.

2.2.2.4. Metoda buttonIncFis_Click

Este apelată atunci când utilizatorul dorește să încarce un fișier cu instrucțiuni în asamblare în locul introducerii manuale pentru fiecare instrucțiune din fișier. Se parcurge linie cu linie fișierul de intrare și se apelează metoda buttonCod_Click.

2.2.2.5. Constructorul clasei CodicatorRISCV

În cadrul acestuia se inițializează listele cu nume de instrucțiuni și dicționarul cu numele alternativ al regiștrilor.

2.2.2.6. Metoda buttonCod_Click

Are ca scop codificarea instrucțiunii curente din textBoxInstr. În funcție de numărul de componente al instrucțiunii se identifică tipul instrucțiunii și valoarea parametrilor ce vor fi codificați. După determinarea codificării se afișează în textBoxInput codificarea în format hexazecimal big endian alături de instrucțiunea originală.

2.2.3. Codicator în varianta Python

Pentru a facilita execuția cross-platform a codificatorului, s-a decis implementarea acestuia și într-o variantă folosind Python și framework-ul Qt. Această variantă a codificatorului este și ea formată din mai multe fișiere. Interfața grafică va fi detaliată în 3.3.10. Interfața grafică.

2.2.3.1. Fişierul InterfataQt.ui

Este un fișier creat în Qt Creator și descrie componentele interfeței programului și poziția acestora în cadrul ferestrei.

2.2.3.2. Fişierul main_window.py

Conține codul Python ce implementează interfața din InterfataQt.ui.

2.2.3.3. Fişierul main.py

Conține logica codificatorului și include elemente asemănătoare implementării în C#, la care se mai adaugă două funcții:

- Funcția limit_32_bits trunchiază un număr la ultimii săi 32 de biți.
- Funcția validare_registru primește ca parametru un registru și verifică faptul că este un registru valid pentru arhitectura RISC-V. În cazul în care nu este valid, se afișează un mesaj într-un MessageBox.

2.2.4. Emulator pe Raspberry Pi Pico

Pentru a putea vizualiza într-un alt mod efectul execuției instrucțiunilor dar și pentru a testa buna interpretare a programelor codificate, s-a dezvoltat un emulator pentru procesorul implementat. Emulatorul este executat pe o placă Raspberry Pi Pico conectată la 16 leduri, numărul acestora fiind echivalentul numărului ledurilor de pe Boolean Board. Acest emulator este format din mai multe fișiere.

2.2.4.1. Fişierul RISC-Vpico.py

Conține implementarea emulatorului și este format din mai multe funcții și o clasă.

Funcția sign_extend are rolul de a extinde semnul unui întreg dat ca prim parametru la 64 de biți, dimensiunea inițială a numărului este dată ca al doilea parametru.

Funcția limit64bits trunchiază numărul dat ca parametru la 64 de biți.

Funcția limit16bits are un efect asemănător cu anterioara, dar trunchiază la 16 biți.

Funcția set_pins_from_result va controla valoarea ledurilor în funcție de rezultatul unei instrucțiuni.

Clasa RISCVProcessor este o implementare la nivel înalt a unui procesor fără pipeline și conține trei metode:

- Constructorul clasei inițializează variabilele de lucru ale procesorului și completează memoria ROM cu octeții din fișierul instructions.txt.
- Metoda execute va citi din memoria ROM instrucțiunea aferentă variabilei PC a clasei, va decodifica instrucțiunea și o va executa.
- Metoda run are rolul de a executa metoda execute cât timp registrul PC indică o locație validă în memoria ROM.

În cadrul funcției main a fișierului se instanțiază clasa RISCVProcessor și se apelează metoda run a obiectului rezultat.

2.3. Analiza avantajelor și dezavantajelor implementării pe FPGA

2.3.1. Avantajele implementării

Implementarea procesorului pe FPGA a adus o serie de avantaje.

2.3.1.1. Flexibilitate și posibilitatea de reconfigurare

Implementarea procesorului a putut fi modificată, nu a fost necesară înlocuirea Boolean Board în situația în care arhitectura a fost modificată în mod substanțial.

2.3.1.2. Paralelism

Paralelismul dat de arhitectură a influențat într-un mod pozitiv performanța procesorului ce lucrează în pipeline.

2.3.1.3. Consum redus de energie

Familia Spartan-7, din care face parte integratul principal al plăcii FPGA, este eficientă energetic, permitând utilizarea procesorului în sisteme incorporate, unde eficiența energetică este esențială.

2.3.1.4. Posibilitatea folosirii unei suite software solide

Placa FPGA a fost programată folosind suita Vivado de la AMD, folosind limbajul de descriere hardware Verilog. Această suită oferă și instrumente de simulare, utile atunci când se dorește testarea procesorului înainte de programarea pe FPGA.

2.3.2. Dezavantajele implementării

2.3.2.1. Resurse limitate

FPGA-ul Spartan-7 XC7S50 are resurse limitate comparativ cu alte FPGA-uri sau integrate din industrie, arhitectura procesorului a trebuit să fie schimbată în urma numărului prea mare de LUT-uri utilizate. Asignările blocante au dus la un consum ridicat de resurse în primele etape ale proiectului, dar după micșorarea mărimii memoriei RAM s-a redus numărul de LUT-uri folosite.

2.3.2.2. Costuri ridicate pentru producția în masă

Procesorul implementat nu ar putea fi produs în masă la costuri eficiente dacă ar rămâne implementat doar pe FPGA din cauza complexității plăcilor, acest lucru rezultând la prețul ridicat al acestora.

2.3.2.3. Consumul mare de energie comparativ cu integrate dedicate

Pentru același nivel de performanță, FPGA-urile consumă mai multă energie comparativ cu integratele dedicate unui scop, chiar dacă familia Spartan-7 este eficientă energetic.

2.3.2.4. Frecvența redusă a clock-ului

Frecvența de lucru a clock-ului plăcii Boolean este limitată la 100 MHz, o valoare destul de mică comparativ cu procesoarele din industrie al căror semnal ceas poate ajunge în domeniul GHz.

2.3.2.5. Necesitatea folosirii de software proprietar

Deși Vivado are o variantă gratuită, un programator nu are acces la codul sursă al suitei software. Acest lucru ar putea limita numărul de dispozitive ce pot executa acest software și implicit, numărul de dispozitive ce pot programa o placă FPGA.

2.4. Componente hardware

2.4.1. Componente hardware necesare

Pentru a putea executa procesorul, este necesară ca resursă hardware placă Boolean Board. Este suficientă varianta fără Bluetooth pentru că nu se folosește această facilitate în cadrul proiectului.

Pentru a implementa emulatorul pe Raspberry Pi Pico, este necesară prezența plăcii, alături de trei circuite ULN2003A sau echivalente și 16 leduri. Circuitele ULN sunt folosite pe post de releu, evitând supraîncărcarea sursei de alimentare internă a plăcii Raspberry Pi Pico.

2.4.2. Scheme bloc

2.4.2.1. Schema bloc a modulului ClockDivider

În Figura 2.1 poate fi vizualizată schema bloc a modulului ClockDivider.

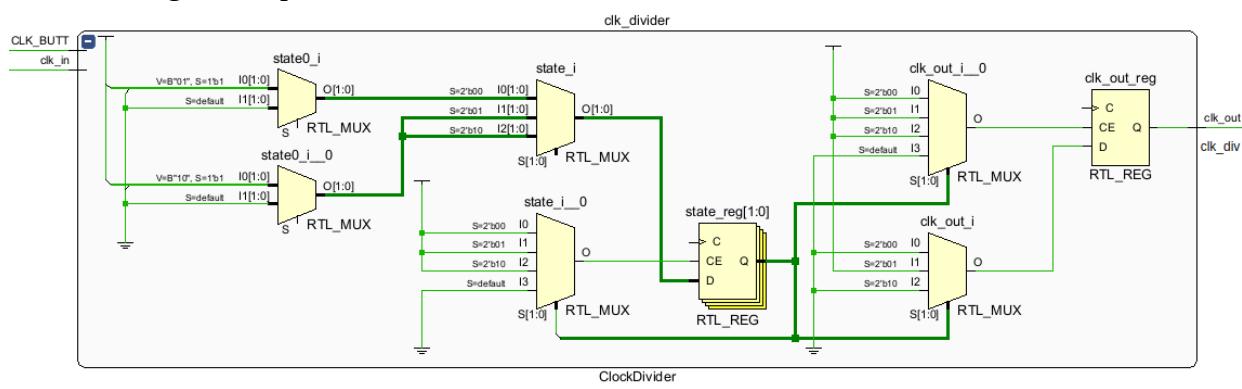


Figura 2.1: Schema bloc a modulului ClockDivider.

Se observă că acest modul a fost implementat folosind multiple multiplexoare și regiștri de tip D.

2.4.2.2. Schema bloc simplificată a modulului RISC-VProcessor

În Figura 2.2 poate fi vizualizată o schemă bloc simplificată a modulului RISC-VProcessor. Schema bloc completă este disponibilă în Anexa 1. Schema bloc completă a modulului RISC-VProcessor.

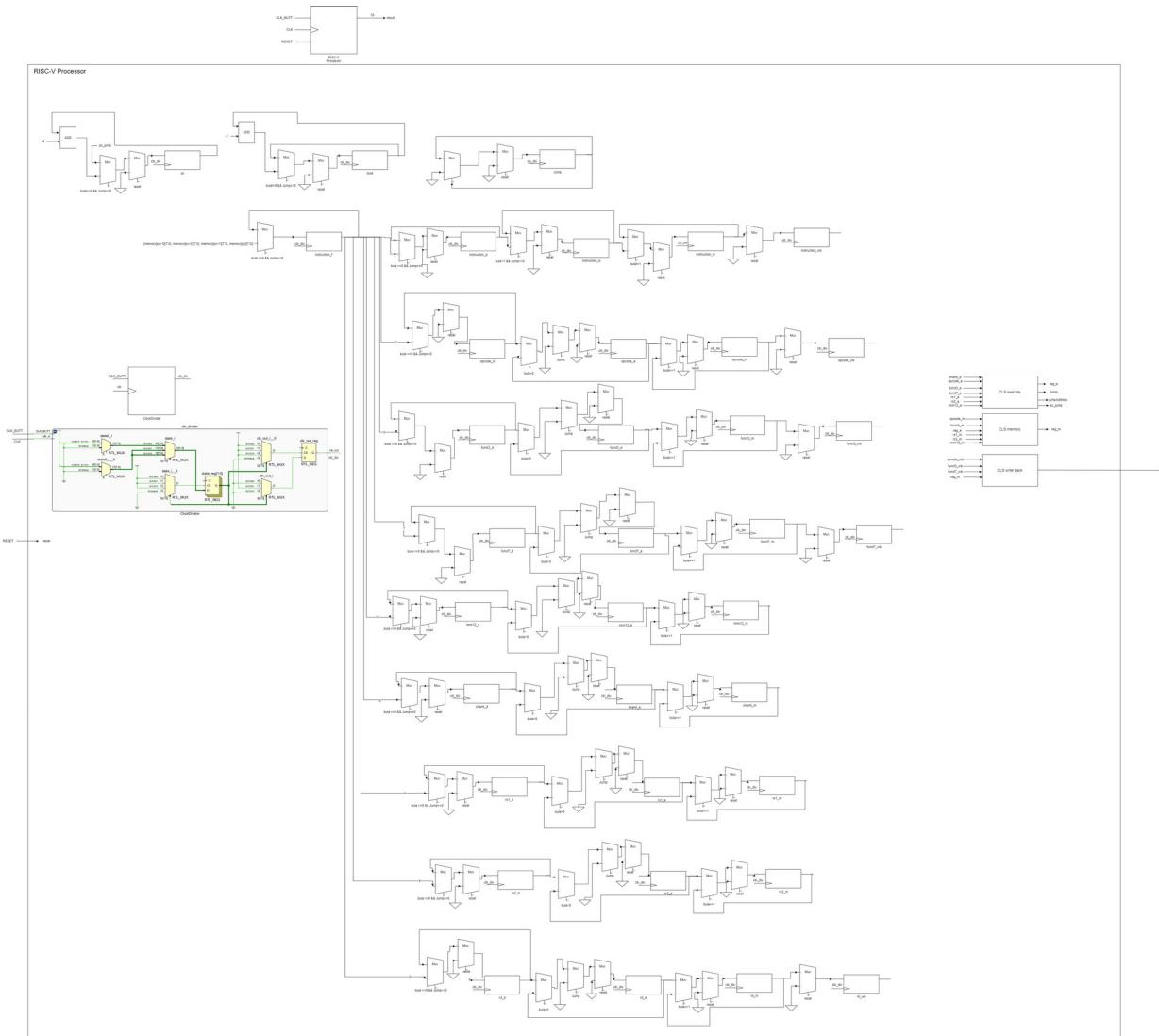


Figura 2.2. Schema bloc simplificată a modului RISC-VProcessor.

Din această schemă bloc se pot identifica câteva părți semnificative, în figurile 2.3, 2.4, 2.5 și 2.6.

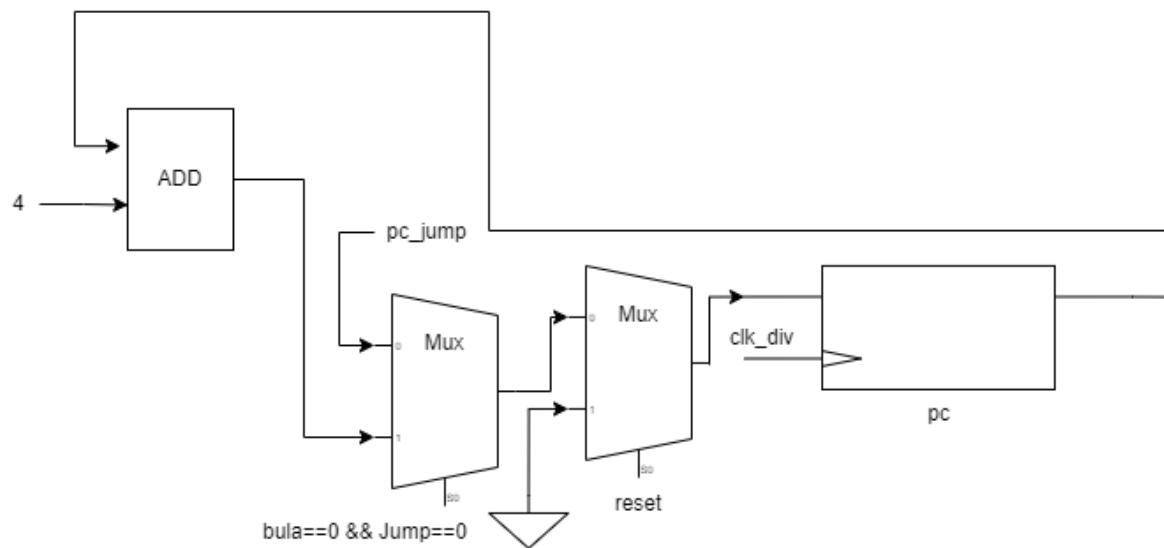


Figura 2.3. Gestionarea registrului PC.

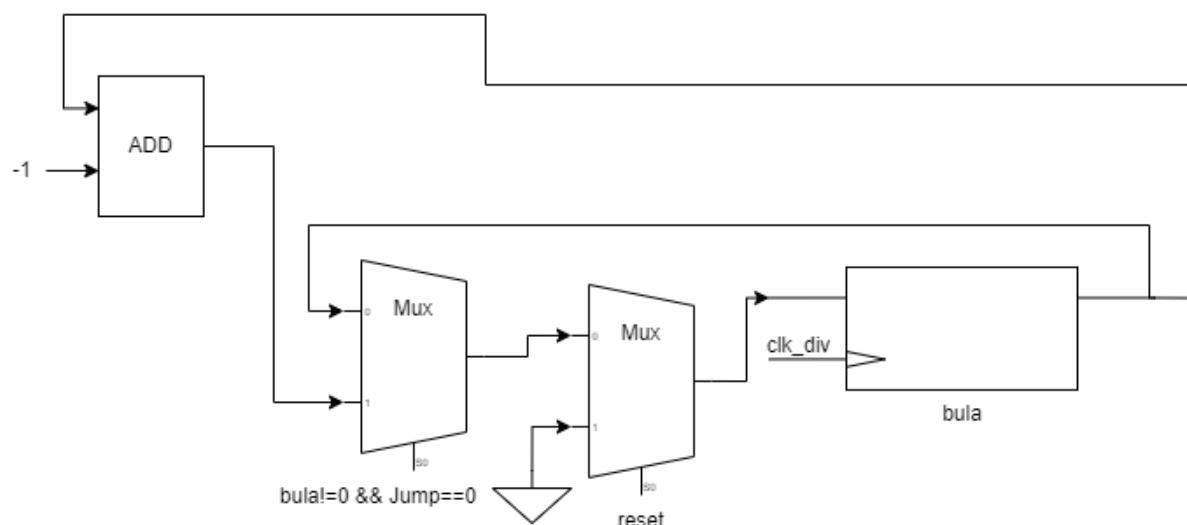


Figura 2.4. Gestionarea registrului bula.

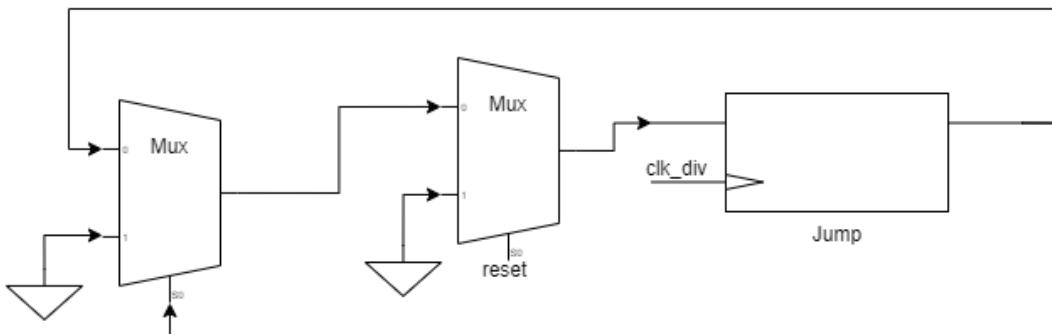


Figura 2.5. Gestionarea registrului Jump.

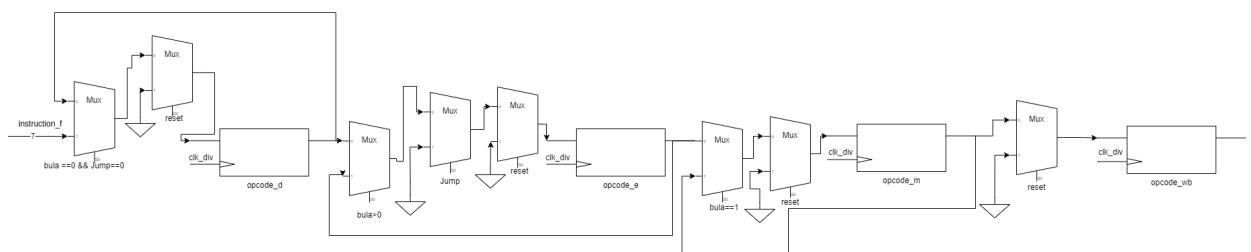


Figura 2.6. Evoluția în cadrul pipeline-ului pentru opcode.

2.4.3. Simularea componentelor hardware

Simularea este esențială în procesul de verificare și validare a design-ului. Aceasta a permis testarea interpretării și executării instrucțiunilor fără a mai programa placă Boolean.

2.4.3.1. Configurarea mediului de simulare

Pentru a configura mediul de simulare de pe Eda Playground, din secțiunea Tools & Simulators se alege din meniu Icarus Verilog 12.0 și se apasă pe opțiunea Open EPWave after run. De asemenea, trebuie scris un script de tip testbench în care se inițializează semnalele și se instanțiază modulul. Se completează fișierul design.v cu implementarea procesorului și mai trebuie create și completate fișierele instructions.mem și data.mem.

2.4.3.2. Scenarii de test pentru procesor

Procesorul a fost testat cu nouă programe al căror cod în asamblare RISC-V este disponibil în Anexa 2. Programele de test în limbaj de asamblare RISC-V. În figurile 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14 și 2.15 sunt disponibile valorile regiștrilor result, pc și clk_div pentru programele de test. În urma analizei figurilor se poate observa execuția corectă a instrucțiunilor.

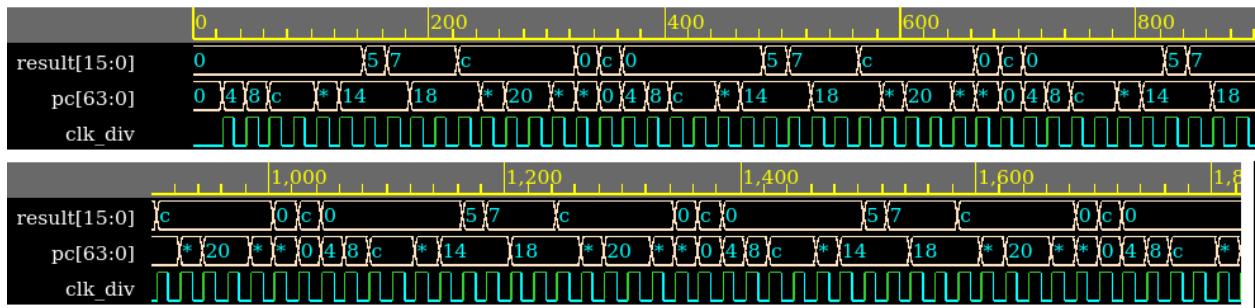


Figura 2.7. Rezultatele pentru program_add_lw.s.

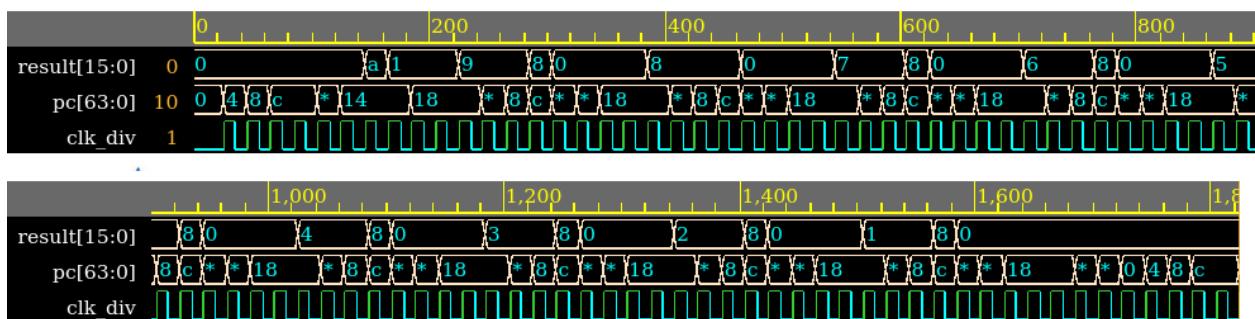


Figura 2.8. Rezultatele pentru program_counter.s.

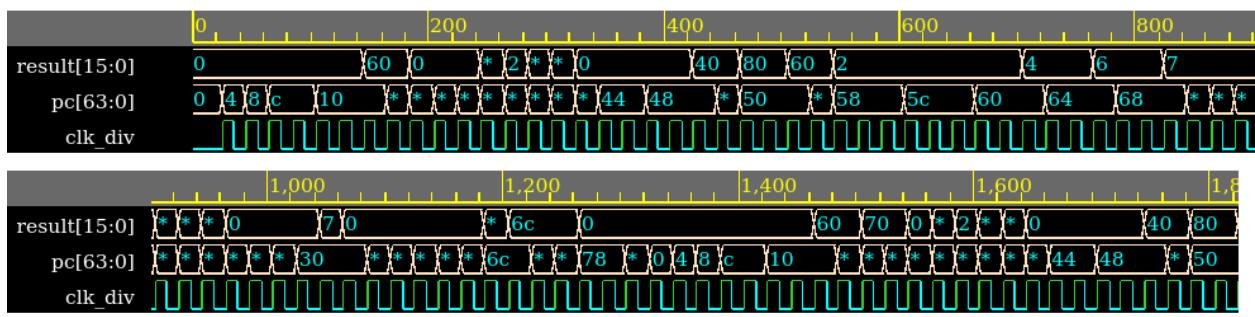


Figura 2.9. Rezultatele pentru program_functie.s.

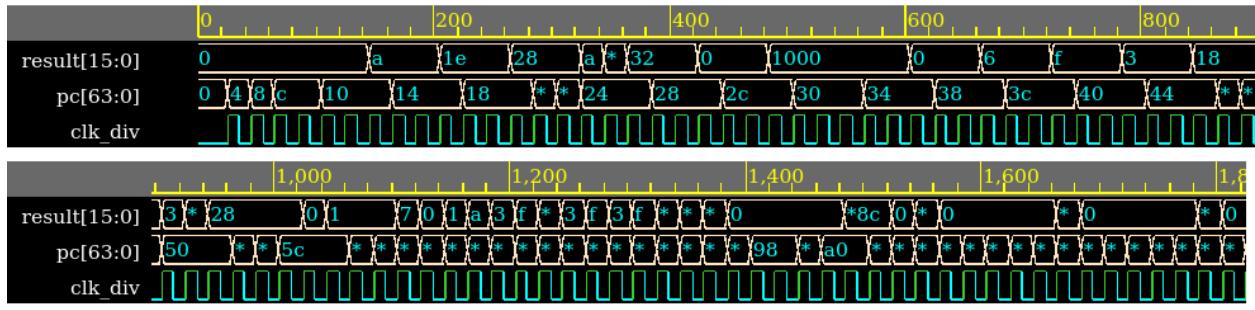


Figura 2.10. Rezultatele pentru program_mare.s.

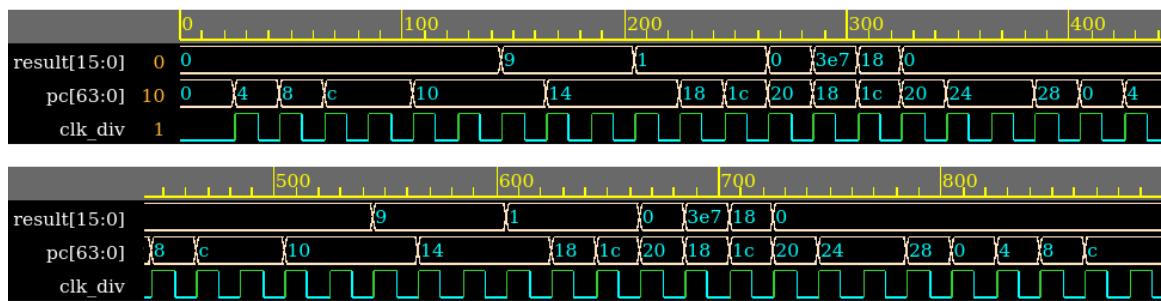


Figura 2.11. Rezultatele pentru program_paritate.s.

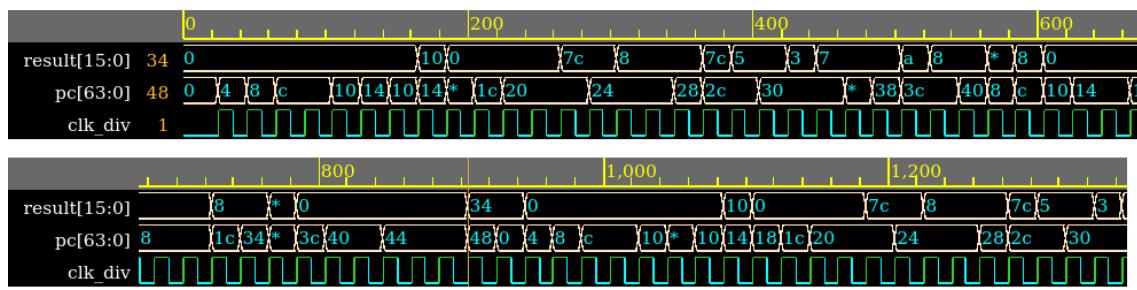


Figura 2.12. Rezultatele pentru program_procedura.s.

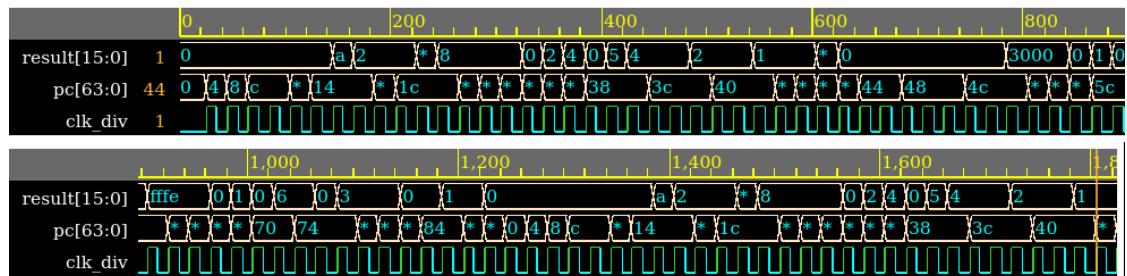


Figura 2.13. Rezultatele pentru program_sll_xor.s.

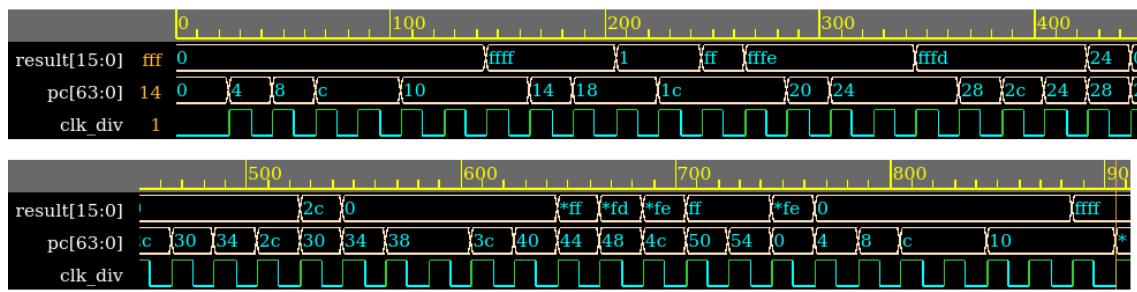


Figura 2.14. Rezultatele pentru program_stocari_extrageri.s.

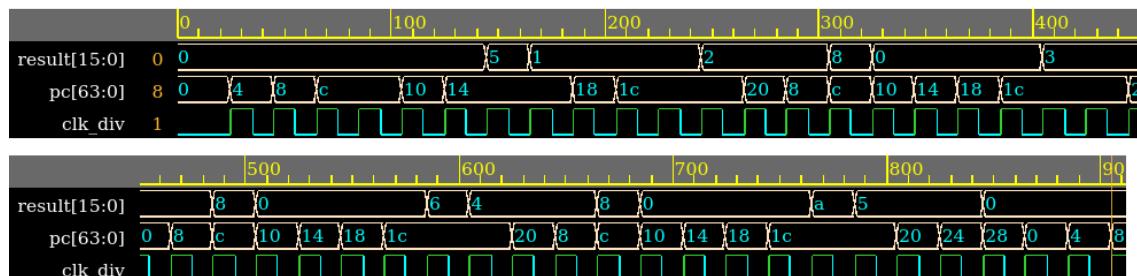


Figura 2.15. Rezultatele pentru program_suma.s.

Capitolul 3. Implementarea procesorului și a aplicațiilor anexe

3.1. Implementarea procesorului în Verilog

3.1.1. Implementarea modulului ClockDivider

```
//Clock controlat de buton
module ClockDivider (
    input wire clk_in,
    input wire CLK_BUTT,
    output reg clk_out
);
    reg [1:0] state; // Starea automatului pe 2 biti pentru cele trei stări
    initial begin
        state=2'b00;
        clk_out=0;
    end
    always @(posedge clk_in) begin
        // Automatul Moore
        case (state)
            2'b00: state <= (CLK_BUTT) ? 2'b01 : 2'b00; // Starea 1
            2'b01: state <= (CLK_BUTT) ? 2'b10 : 2'b00; // Starea 2
            2'b10: state <= (CLK_BUTT) ? 2'b10 : 2'b00; // Starea 3
        endcase
        // Comutare clock in functie de starea automatului
        case (state)
            2'b00: clk_out <= 1'b0; // Starea 1, iesire 0
            2'b01: clk_out <= 1'b1; // Starea 2, iesire 1
            2'b10: clk_out <= 1'b0; // Starea 3, iesire 0
        endcase
    end
endmodule
```

Implementarea modulului ClockDivider se bazează pe un automat Moore cu rol de sincronizare între clk_in și CLK_BUTT. Folosește un registru de stare și în funcție de valoarea anterioară a acestui registru și de valoarea curentă a CLK_BUTT se determină noua valoare a registrului de stare. În funcție de valoarea registrului state, se calculează valoarea pentru clk_out.

3.1.2. Implementarea modulului RISC-VProcessor

Codul complet al modulului este disponibil în Anexa 3. Codul complet al modulului RISC-VProcessor. În continuare vor fi prezentate secțiunile semnificative pentru fiecare etapă a modulului.

3.1.2.1. Implementarea etapei de inițializare

```
initial begin
    //Citirea din fisier a continutului memoriei ROM
    $readmemh("instructions.mem", memory);
    pc = 0; // Initializare cu 0
```

```

//Initializare cu 0 banc de registre
for (i = 0; i <= 31; i = i + 1)
    registers[i] = (i==2)?128:0;
//Initializare cu 0 memorie de date
$readmemh("data.mem", dataMemory);
[...]
end

```

În cadrul etapei de inițializare se introduce valoarea implicită pentru regiștrii și memoriile de lucru.

3.1.2.2. Instanțierea modulului ClockDivider

```

//Instantiere modul de control al clock-ului
ClockDivider clk_divider (
    .clk_in(clk),
    .clk_out(clk_div),
    .CLK_BUTT(CLK_BUTT)
);

```

Se instanțiază modulul ClockDivider cu semnalele de intrare clk și CLK_BUTT, iar ieșirea modulului este plasată în clk_div.

3.1.2.3. Verificarea frontului pozitiv pentru clk_div și a valorii semnalului reset

```

always @(posedge clk_div) begin
    //Daca semnalul reset nu este activ, atunci continuu operarea in pipeline.
    if(!reset) begin
        //Etapa de pipeline
        [...]
    end
    else
        //Daca reset este activ, reinitializez valorile componentelor(mai putin ROM).
        begin
            pc <= 64'h0;
            result<=0;
            Jump <= 0;
            jumpAddress_e<=0;
            jumpAddress_m<=0;
            bula<=0;
            for (i = 0; i <= 31; i = i + 1)
                registers[i] <= (i==2)?128:0;
            $readmemh("data.mem", dataMemory);
            [...]
        end

```

La frontul pozitiv al clk_div se intră în blocul always@ și se verifică valoarea din reset. În cazul în care reset este activ, atunci se face reinițializarea variabilelor. Altfel, se intră în etapa de pipeline.

3.1.2.4. Pipeline verificarea valorilor pentru regiștrii bula și Jump

```
//Control al semnalului bula care se ocupă de gestiunea hazardurilor de date în
pipeline.
    if(bula!=0 && Jump==0)
        bula=bula-1;
    if (Jump==1) begin
        Jump<=0;
        pc<=pc_jump;
    end
```

3.1.2.5. Pipeline fetch

```
//Etapa fetch
if(bula==0 && Jump==0)begin
    instruction_f <= {memory[pc+3][7:0], memory[pc+2][7:0], memory[pc+1][7:0],
    memory[pc][7:0]};
    pc<=pc+4;
end
```

Dacă regiștrii bula și Jump au valori nule, atunci se va introduce în instruction_f valoarea din memoria ROM corespunzătoare registrului PC.

3.1.2.6. Pipeline decode

```
//Etapa decode
if(bula==0)begin
    if(Jump==0)begin
        instruction_d<=instruction_f;
        opcode_d <= instruction_f[6:0];
        funct3_d <= instruction_f[14:12];
        funct7_d <= instruction_f[31:25];
        imm12_d <= instruction_f[31:20];
        shamt_d <= instruction_f[25:20];
        rs1_d <= instruction_f[19:15];
        rs2_d <= instruction_f[24:20];
        rd_d <= instruction_f[11:7];
        //Tratare hazard date
        if(rs1_d == rd_e || rs2_d == rd_e)
            bula<=3'b010;
        else if(rs1_d == rd_m || rs2_d == rd_m)
            bula<=3'b001;
    end
end
```

În cadrul acestei etape instrucțiunea este decodificată și este setată valoarea din registrul bula în cazul în care apar hazarduri de date.

3.1.2.7. Pipeline execute

```
//Etapa execute
if(bula<1 && Jump==0) begin
```

```

opcode_e <= opcode_d;
funct3_e <= funct3_d;
funct7_e <= funct7_d;
imm12_e <= imm12_d;
shamt_e <= shamt_d;
rs1_e <= rs1_d;
rs2_e <= rs2_d;
rd_e <= rd_d;
instruction_e<=instruction_d;
//Stabilirea operatiei de executat in functie de parametrii decodificati.
case (opcode_e)
  7'b0110011: begin
    // Instructiuni de tip R
    case (funct3_e)
      3'b000: begin // ADD, SUB
        if (funct7_e == 7'b0000000)
          reg_e<= registers[rs1_e] + registers[rs2_e];
        else if(funct7_e == 7'b0100000)
          reg_e<= registers[rs1_e] - registers[rs2_e];
      end
      3'b001: // SLL pentru RV64I
      reg_e <= registers[rs1_e] << registers[rs2_e][5:0];
      [...]
    endcase
  end
  7'b0110111: begin // LUI
    reg_e <= {{32{instruction_e[31]}},instruction_e[31:12], 12'b0};
  end
  7'b0010111: begin // AUIPC
    reg_e <= pc - 12 + {{32{instruction_e[31]}},instruction_e[31:12],
12'b0};//Valoare PC de la fetch
  end
endcase
end

```

În cadrul acestei etape, în funcție de valorile decodificate se execută instrucțiunile.

3.1.2.8. Pipeline memory

```

//Etapa memory
jumpAddress_m<=jumpAddress_e;
if(bula!=1) begin
  opcode_m <= opcode_e;
  funct3_m <= funct3_e;
  funct7_m <= funct7_e;
  imm12_m <= imm12_e;
  shamt_m <= shamt_e;
  rs1_m <= rs1_e;
  rs2_m <= rs2_e;
  rd_m <= rd_e;
  instruction_m<=instruction_e;

```

```

//Pentru aceasta etapa, se executa operatii specifice pentru instructiunile
de load si store in memorie. Restul instructiunilor doar vor primi in reg_m valoarea
anterioara din reg_e.
case (opcode_m)
  7'b0000011: begin
    // Instructiuni load
    case (funct3_m)
      3'b000: begin // LB
        reg_m <= {{56{dataMemory[registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7]}},
                    {dataMemory[registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0]}};
      end
      [...]
    endcase
  end
  7'b0100011: begin
    // Instructiuni store
    case (funct3_m)
      3'b000: begin // SB
        dataMemory[registers[rs1_m]+{{52{instruction_m[31]}},
                                    {instruction_m[31:25],instruction_m[11:7]}]} <= registers[rs2_m][7:0];
        reg_m<=registers[rs2_m][7:0];
      end
      [...]
    endcase
  end
  default: reg_m<=reg_e;
endcase
end

```

În cadrul etapei memory instrucțiunile de lucru cu memoria sunt executate, restul instructiunilor doar trimit mai departe valoarea din reg_e în reg_m.

3.1.2.9. Pipeline write-back

```

//Etapa write back
opcode_wb<=opcode_m;
funct3_wb <= funct3_m;
funct7_wb <= funct7_m;
rd_wb <= rd_m;
instruction_wb<=instruction_m;
  //Este etapa in care se scrie rezultatul operatiei in result si in registrul
destinatie.
case (opcode_wb)
  7'b0110011: begin
    // Instructiuni de tip R
    case (funct3_wb)
      3'b000: begin // ADD, SUB
        if (funct7_wb == 7'b0000000)
          begin
            if (rd_wb!=0) begin
              registers[rd_wb]<= reg_m;
            end
          end
      end
    endcase
  end

```

```

        result<=reg_m;
    end
    else
        result<=0;
    end
    else if (funct7_wb == 7'b0100000)
    begin
        if (rd_wb!=0) begin
            registers[rd_wb]<= reg_m;
            result<=reg_m;
        end
        else
            result<=0;
    end
    end
    [...]
    endcase
end
[...]
default: result<= 0;
endcase

```

În această etapă se scrie în registrul destinație valoarea din reg_m dacă este cazul. Pentru toate operațiile va avea loc actualizarea registrului de ieșire result.

3.2. Implementarea codificatorului în varianta C#

În următoarele subcapitole vor fi prezentate fragmentele semnificative din metodele clasei CodificatorRISCV și la final descrierea interfeței grafice a codificatorului. Codul complet al clasei CodificatorRISCV se află în Anexa 4. Codul sursă complet al clasei CodificatorRISCV.

3.2.1. Metoda buttonInstr_Click

```

string continutInput = textBoxInput.Text;
if (continutInput.Length == 0) {
    MessageBox.Show(
        "Nu pot salva un fișier dacă nu au fost procesate instrucțiuni!");
    return;
}
if (saveFileDialog.ShowDialog() == DialogResult.OK) {
    string caleFisier = saveFileDialog.FileName;

    try {
        string rezultat = "";
        // Se extrag octetii din instructiune în ordine inversă și se afisează câte
        // un octet pe linie, pentru a putea fi cititi de funcția readmemh din
        // Verilog.
        List<string> instructions = continutInput.Split('\r')
            .Select(x => x.Replace("\n", ""))
            .Where(s => !string.IsNullOrEmpty(s))
            .ToList();
    }
}

```

```

foreach (string instruction in instructions) {
    string[] componente = instruction.Split(' ');
    rezultat += string.Join(Environment.NewLine,
        componente [0]
            .Substring(6, 2),
        componente [0]
            .Substring(4, 2),
        componente [0]
            .Substring(2, 2),
        componente [0]
            .Substring(0, 2),
        "");
}
rezultat = rezultat.Substring(0, rezultat.Length - 2);
File.WriteAllText(caleFisier, rezultat);

MessageBox.Show("Instrucţiunile au fost salvate în format mem!");
} catch (Exception ex) {
    MessageBox.Show("A apărut o eroare la salvarea fișierului: " + ex.Message);
}
}

```

În cazul în care utilizatorul a parcurs cu succes dialogul de salvare a fișierului, se parcurge conținutul din textBoxInput linie cu linie și se extrag octeții în ordine little endian care sunt introdusi în fișierul de ieșire.

3.2.2. Metoda buttonIncFis_Click

```

// Se deschide fisierul si se apeleaza functia de codificare pentru fiecare
// instructiune
openFileDialog.Filter = "Assembly Files (*.s)|*.s";

if (openFileDialog.ShowDialog() == DialogResult.OK) {
    string caleFisier = openFileDialog.FileName;
    try {
        using(StreamReader reader = new StreamReader(caleFisier)) {
            while (!reader.EndOfStream) {
                string linie = reader.ReadLine();
                textBoxInstr.Text = linie.Trim(' ').Replace("\r\n", "");
                this.buttonCod_Click(sender, e);
            }
        }

        MessageBox.Show("Fișier încărcat și interpretat cu succes!");
    } catch (Exception ex) {
        MessageBox.Show(
            "A apărut o eroare la citirea și interpretarea fișierului: " +
            ex.Message);
    }
}

```

După deschiderea cu succes a fișierului cu instrucțiuni în asamblare se apelează metoda buttonCod_Click pentru fiecare linie.

3.2.3. Metoda buttonDel_Click

```
//Se elibera continutul textBox-ului aferent
textBoxInput.Text = "";
```

Doar se elimină conținutul din textBoxInput.

3.2.4. Metoda buttonSal_Click

```
if (textBoxInput.Text.Length == 0) {
    MessageBox.Show(
        "Nu pot salva un fișier dacă nu au fost procesate instrucțiuni!");
    return;
}
if (saveFileDialog.ShowDialog() == DialogResult.OK) {
    string caleFisier = saveFileDialog.FileName;

    try {
        // Instrucțiunile afisate în textBox corespund cu formatul
        // în care vreau să afisez instrucțiunile în fisier.
        File.WriteAllText(caleFisier, textBoxInput.Text.Substring(
            0, textBoxInput.Text.Length - 2));

        MessageBox.Show("Instrucțiunile au fost salvate în format input!");
    } catch (Exception ex) {
        MessageBox.Show("A apărut o eroare la salvarea fișierului: " + ex.Message);
    }
}
```

Se salvează conținutul din textBoxInput în fișierul indicat fără a salva și ultimul Enter.

3.2.5. Constructorul clasei CodificatorRISCV

```
// Imi initializez liste de instrucțiuni și numele alternative pentru
// registri.
_loads = new List<string>{"lb", "lh", "lw", "ld", "lbu", "lhu", "lwu"};
_stores = new List<string>{"sb", "sh", "sw", "sd"};
_jumps = new List<string>{"jal", "jalr"};
_branches = new List<string>{"beq", "bne", "blt", "bge", "bltu", "bgeu"};
_others = new List<string>{"lui", "auipc"};
_registerAlternateName = new Dictionary<string, string>();
_registerAlternateName.Add("zero", "x0");
_registerAlternateName.Add("ra", "x1");
_registerAlternateName.Add("sp", "x2");
_registerAlternateName.Add("gp", "x3");
_registerAlternateName.Add("tp", "x4");
_registerAlternateName.Add("t0", "x5");
[...]
```

În această funcție se initializează listele de nume pentru instrucțiuni și dicționarul de nume alternative pentru regiștri.

3.2.6. Metoda buttonCod_Click

```
// Citirea instructiunii
string instructiune = textBoxInstr.Text;
// Impartirea instructiunii pe componente
string[] componente = instructiune.Split(' ');
// În funcție de numarul de componente și de valoarea acestora, se va calcula
// codificarea hexazecimală a instructiunii. Dacă apar erori în parsarea
// instructiunii, utilizatorul va fi notificat printr-un MessageBox.
if (componente.Length == 4)
{
    string rd = componente[1]
        .Replace(",", "");
    string rs1 = componente[2]
        .Replace(",", "");
    if (!rd.StartsWith("x"))
    {
        _registerAlternateName.TryGetValue(rd, out rd);
        if (rd == null)
        {
            MessageBox.Show($"Nu am găsit echivalent pentru rd! {instructiune}");
            return;
        }
    }
    if (!rs1.StartsWith("x"))
    {
        _registerAlternateName.TryGetValue(rs1, out rs1);
        if (rs1 == null)
        {
            MessageBox.Show($"Nu am găsit echivalent pentru rs1! {instructiune}");
            return;
        }
    }
    int rdInt = int.Parse(rd.Substring(1));
    int rs1Int = int.Parse(rs1.Substring(1));
    if (rdInt > 32 || rs1Int > 32 || rs1Int < 0 || rdInt < 0)
    {
        MessageBox.Show($"Registrii în afara ariei de lucru! {instructiune}");
        return;
    }
    if (componente[0]
        .EndsWith("i"))
    {
        string imm = componente[3];
        int immInt = int.Parse(imm);
        if (immInt > 2047 || immInt < -2048)
        {
            MessageBox.Show($"Valoare prea mare sau mică pentru imm!");
        }
    }
}
```

```

{instructiune}");
    return;
}
if ((immInt > 63 || immInt < 0) &
    (componente[0] == "slli" || componente[0] == "srli" ||
     componente[0] == "srai"))
{
    MessageBox.Show($"Valoare prea mare sau mică pentru imm!
{instructiune}");
    return;
}
int funct3, funct7;
int opcode;
switch (componente[0])
{
    case "addi":
        opcode = 0b0010011;
        funct7 = 0;
        funct3 = 0;
        break;
    case "slli":
        opcode = 0b0010011;
        funct7 = 0;
        funct3 = 0b001;
        break;
    [...]
    default:
        MessageBox.Show($"Operație nesuportată! {instructiune}");
        return;
}
int instrCod = opcode | (rdInt << 7) | (funct3 << 12) | (rs1Int << 15) |
    (immInt << 20) | (funct7 << 25);
textBoxInput.AppendText($"{instrCod.ToString("X8")}) {instructiune}" +
    Environment.NewLine);
[...]
}
else if (componente.Length == 3)
{
    if (_others.Contains(componente[0]))
    {
        string rd = componente[1]
            .Replace(",", "");
        if (!rd.StartsWith("x"))
        {
            _registerAlternateName.TryGetValue(rd, out rd);
            if (rd == null)
            {
                MessageBox.Show($"Nu am găsit echivalent pentru rd!
{instructiune}");
                return;
            }
        }
    }
}

```

```

        }
        int rdInt = int.Parse(rd.Substring(1));
        if (rdInt < 0 || rdInt > 31)
        {
            MessageBox.Show($"Registru rd înafara ariei de lucru!
{instructiune}");
            return;
        }
        string imm = componente[2];
        int immInt = int.Parse(imm);
        if (immInt > 1048575 || immInt < -1048576)
        {
            MessageBox.Show(
                $"Valoare prea mare sau mică pentru imm! {instructiune}");
            return;
        }
        int opcode;
        switch (componente[0])
        {
            case "lui":
                opcode = 0b0110111;
                break;
            case "auipc":
                opcode = 0b0010111;
                break;
            default:
                MessageBox.Show($"Operatie nesuportata! {instructiune}");
                return;
        }
        int instrCod = opcode | (rdInt << 7) | (immInt << 12);
        textBoxInput.AppendText($"{instrCod.ToString(" X8 ")} {instructiune}"
+
                                         Environment.NewLine);
    }
    else if (_jumps.Contains(componente[0]))
    { // Jump-uri
        if (componente[0] == "jal")
        {
            [...]
            int instrCod = opcode | (rdInt << 7) | (((immInt >> 20) & 1) << 31) |
                (((immInt >> 1) & 0b1111111111) << 21) |
                (((immInt >> 11) & 1) << 20) |
                (((immInt >> 12) & 0b11111111) << 12);
            textBoxInput.AppendText($"{instrCod.ToString(" X8 ")}
{instructiune}" +
                                         Environment.NewLine);
        }
        [...]
    }
    else
    {
        MessageBox.Show(

```

```
$"Format incorrect sau instructiune nesuportată!
{instructiune}");
}
```

În funcție de numărul de componente al instrucțiunii se codifică și se concatenează codificarea fiecărei instrucțiuni la textBoxInput, alături de instrucțiunea originală.

3.2.7. Interfața grafică

În Figura 1.1 se poate remarca o captură de ecran din timpul execuției codificatorului.

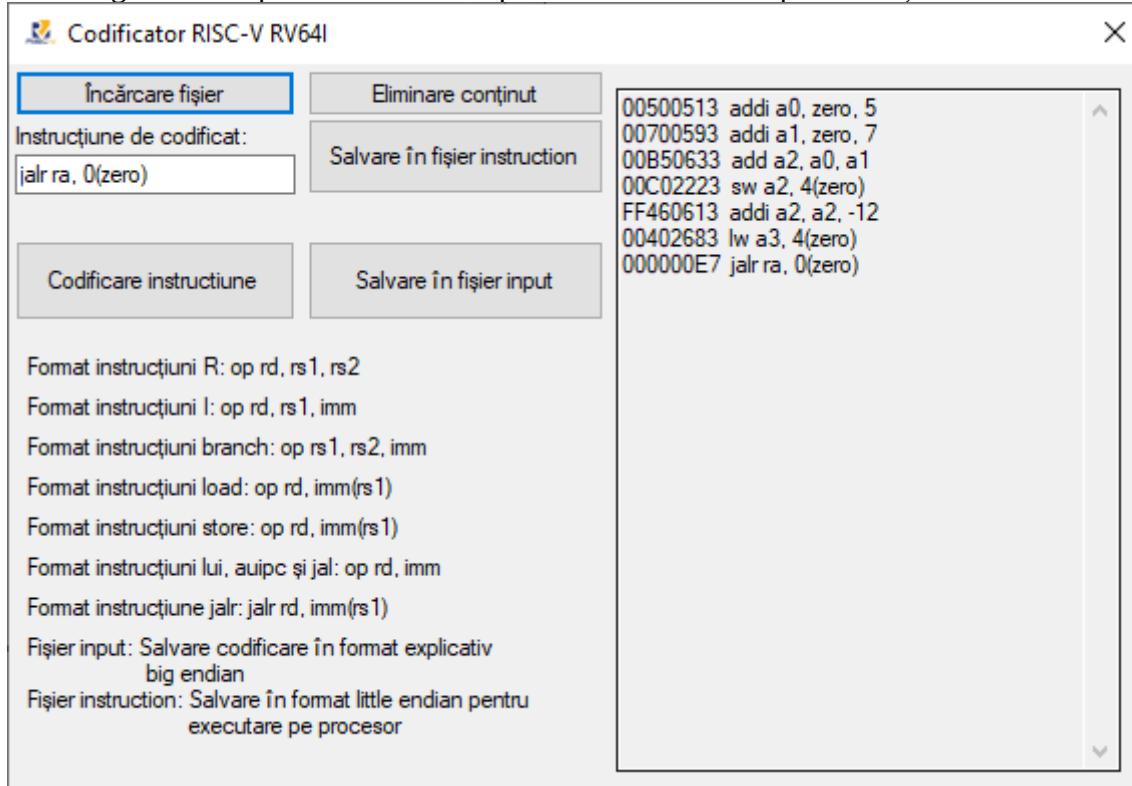


Figura 3.1. Codificator RISC-V C# aflat în execuție.

Interfața este compusă din mai multe elemente grafice ce permit interacțiunea cu utilizatorul, etichetate în Figura 3.2.

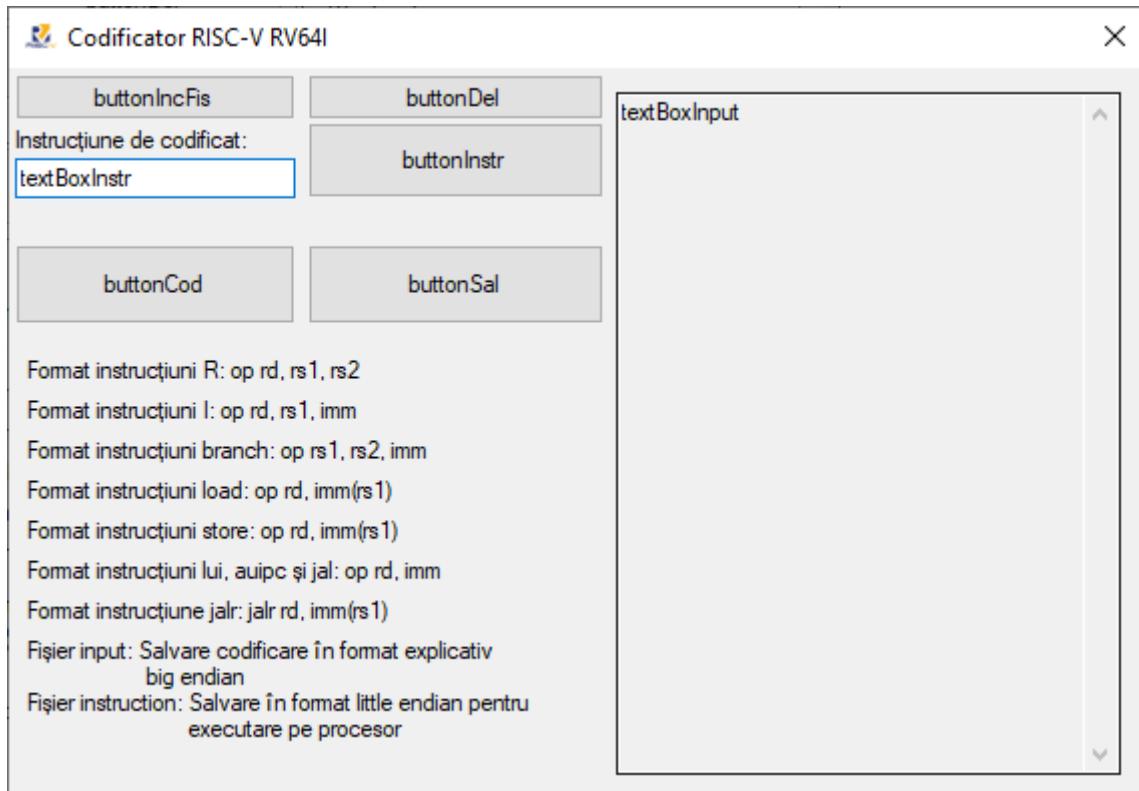


Figura 3.2. Codificator C# cu elemente interactive etichetate.

Interacțiunea cu aceste elemente este tratată de metodele clasei CodificatorRISC-V.

3.3. Implementarea codificatorului în varianta Python

În continuare va fi detaliată implementarea funcțiilor și a clasei MainWindow din fișierul main.py și prezentarea interfeței grafice pentru acest codificator. Conținutul complet al fișierului main.py este disponibil în Anexa 5. Conținutul fișierului main.py al codificatorului Python.

3.3.1. Funcția limit_32_bits

```
#Functie care trunchiaza un numar la 32 de biti
def limit_32_bits(number):
    return number & 0xFFFFFFFF
```

3.3.2. Constructorul clasei MainWindow

```
[...]
self.ui.buttonDel.clicked.connect(self.buttonDel_clicked)
self.ui.buttonSal.clicked.connect(self.buttonSal_clicked)
self.ui.buttonIncFis.clicked.connect(self.buttonIncFis_clicked)
self.ui.buttonCod.clicked.connect(self.buttonCod_clicked)
self.ui.buttonInstr.clicked.connect(self.buttonInstr_clicked)
```

Constructorul face legătura dintre evenimentele de apăsare și celelalte metode ale clasei.

3.3.3. Metoda *validare_registru*

```
def validare_registru(self, rd, nume_reg):
    if rd[0] != "x":
        try:
            rd = registerAlternateName[rd]
        except KeyError:
            QMessageBox.warning(self, "Avertisment", f"Nu am găsit echivalent
pentru {nume_reg}! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,
                     return 100
    rd = int(rd[1:])
    if rd > 32 or rd < 0:
        QMessageBox.warning(self, "Avertisment", f"Registru {nume_reg} în afara
ariei de lucru! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,
                     return 100
    return rd
```

Se verifică validitatea registrului dat ca parametru.

3.3.4. Metoda *buttonDel_clicked*

```
#Functia de eliminare a continutului textBox-ului cu instructiuni din dreapta
def buttonDel_clicked(self):
    self.ui.listInput.clear()
```

3.3.5. Metoda *buttonIncFis_clicked*

```
#Functia de incarcare a unui fisier cu instructiuni in asamblare
def buttonIncFis_clicked(self):
    file_path, _ = QFileDialog.getOpenFileName(self, "Citire fișier asamblare",
                                              "", "Assembly Files (*.s)")
    if file_path:
        with open(file_path, "r") as file:
            for line in file:
                self.ui.textInstr.setText(line.strip())
                self.buttonCod_clicked()
        QMessageBox.information(self, "Parcursare completă", "Fișierul a fost
parcurs!", QMessageBox.Ok)
```

3.3.6. Metoda *buttonSal_clicked*

```
#Functia de salvare a instructiunilor din textBox in format explicativ
def buttonSal_clicked(self):
    file_path, _ = QFileDialog.getSaveFileName(self, "Salvare în fișier input",
                                                "input_.txt", "Text Files (*.txt)")
    if file_path:
        with open(file_path, "w") as file:
            for index in range(self.ui.listInput.count()):
                item = self.ui.listInput.item(index)
                file.write(f"{item.text()}\n")
```

```
QMessageBox.information(self,"Salvare completă","Instrucțiunile au fost salvate în format input!",QMessageBox.Ok,)
```

3.3.7. Metoda buttonInstr_clicked

```
#Functia de salvare a instructiunilor in format executabil pe procesor
def buttonInstr_clicked(self):
    file_path, _ = QFileDialog.getSaveFileName(self,"Salvare în fișier
instruction","instruction_.mem","Memory Files (*.mem"),)
    if file_path:
        with open(file_path, "w") as file:
            for index in range(self.ui.listInput.count()):
                instruction = self.ui.listInput.item(index).text()
                componente = instruction.split(" ")
                rezultat = (componente[0][6:8]+ os.linesep + componente[0][4:6] +
os.linesep + componente[0][2:4] + os.linesep + componente[0][0:2] + os.linesep)
                file.write(f"{rezultat}")
        QMessageBox.information(self,"Salvare completă","Instrucțiunile au fost salvate în format mem!",QMessageBox.Ok,)
```

3.3.8. Metoda buttonCod_clicked

```
#Functia de codificare a unei instructiuni
def buttonCod_clicked(self):
    try:
        #Impartirea instructiunii pe componente
        componente = self.ui.textInstr.toPlainText().split(" ")
        #In functie de numarul de componente si valoarea acestora, se formeaza
        instructiunea codificata in format hexazecimal
        #In cazul in care apar probleme pe parcursul parsarii instructiunii,
        utilizatorul va fi notificat printr-un MessageBox
        if len(componente) == 4:
            rd = componente[1].strip(",")
            rs1 = componente[2].strip(",")
            rd = self.validare_registru(rd, "rd")
            if rd == 100:
                return
            rs1 = self.validare_registru(rs1, "rs1")
            if rs1 == 100:
                return
            if componente[0][-1] == "i" or componente[0] == "sliu":
                imm = int(componente[3])
                if imm > 2047 or imm < -2048:
                    QMessageBox.warning(self,"Avertisment",f"Valoare prea mică
sau mare pentru imm! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)
                    return
                if (imm > 63 or imm < 0) and (componente[0] == "slli" or
componente[0] == "srli" or componente[0] == "srai"):
                    QMessageBox.warning(self,"Avertisment",f"Valoare prea mică
```

```

sau mare pentru imm! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)

        return
    if componente[0] == "addi":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0
    elif componente[0] == "slli":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b001
    [...]
    else:
        QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)

        return
    instrCod = limit_32_bits(opcode | (rd << 7) | (funct3 << 12) |
(rs1 << 15) | (imm << 20) | (funct7 << 25))
    self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
    [...]
else:
    rs2 = componente[3].strip(",")
    rs2 = self.validare_registru(rs2, "rs2")
    if rs2 == 100:
        return
    opcode = 0b0110011
    if componente[0] == "add":
        funct3 = 0
        funct7 = 0
    elif componente[0] == "sub":
        funct7 = 0b0100000
        funct3 = 0
    elif componente[0] == "sll":
        funct7 = 0
        funct3 = 0b001
    [...]
    else:
        QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)

        return
    instrCod = limit_32_bits(opcode | (rd << 7) | (funct3 << 12) |
(rs1 << 15) | (rs2 << 20) | (funct7 << 25))
    self.ui.listInput.addItem("{:08X}".format(instrCod)+ " " +
self.ui.textInstr.toPlainText())
    elif len(componente) == 3:
        if componente[0] in others:
            rd = componente[1].strip(",")
            rd = self.validare_registru(rd, "rd")
            if rd == 100:
                return
            imm = int(componente[2])

```

```

        if imm > 1048575 or imm < -1048576:
            QMessageBox.warning(self, "Avertisment", f"Valoare prea
            mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        [...]
    else:
        QMessageBox.warning(self, "Avertisment", f"Operătie nesuportată!
        {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
        return
except Exception as e:
    QMessageBox.warning(self, "Avertisment", f"A apărut o eroare la tratarea
    instrucțiunii: {self.ui.textInstr.toPlainText()} {str(e)}", QMessageBox.Ok,)
    return

```

3.3.9. Funcția main

```

#Functia principala, in care se instantiaza clasa MainWindow.
def main():
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())

```

3.3.10. Interfața grafică

În Figura 3.3 este prezentată o captură de ecran din timpul execuției codificatorului.

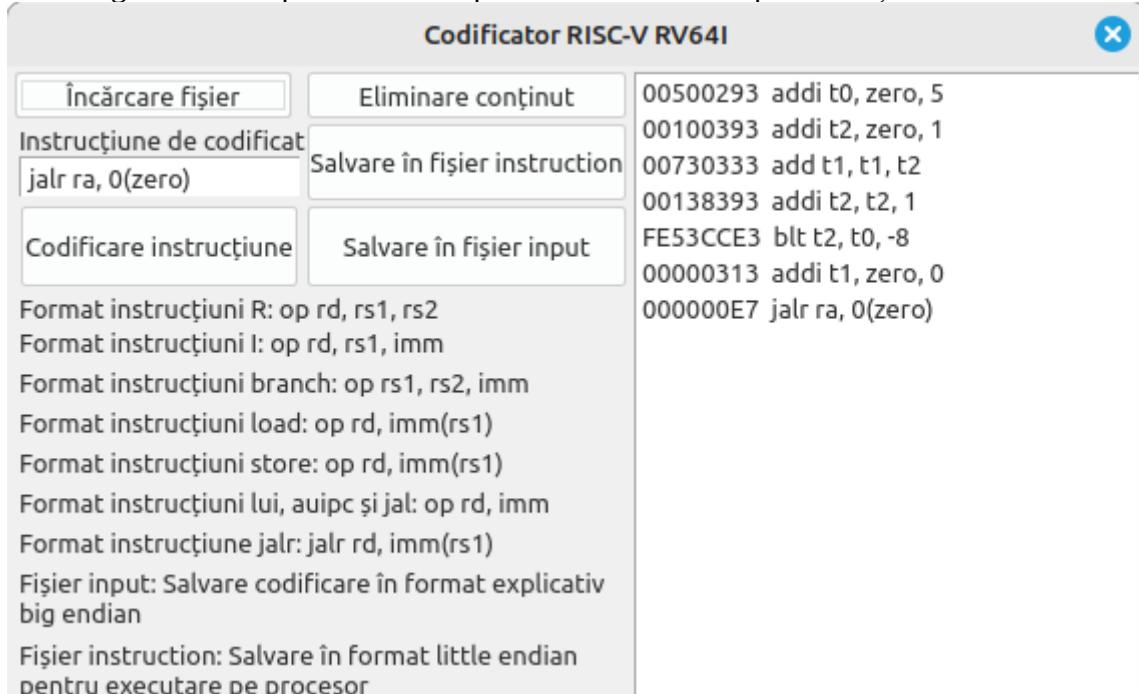


Figura 3.3. Execuție pe Linux Mint a codificatorului Python.

În Figura 3.4 sunt etichetate elementele grafice cu care utilizatorul poate interacționa.

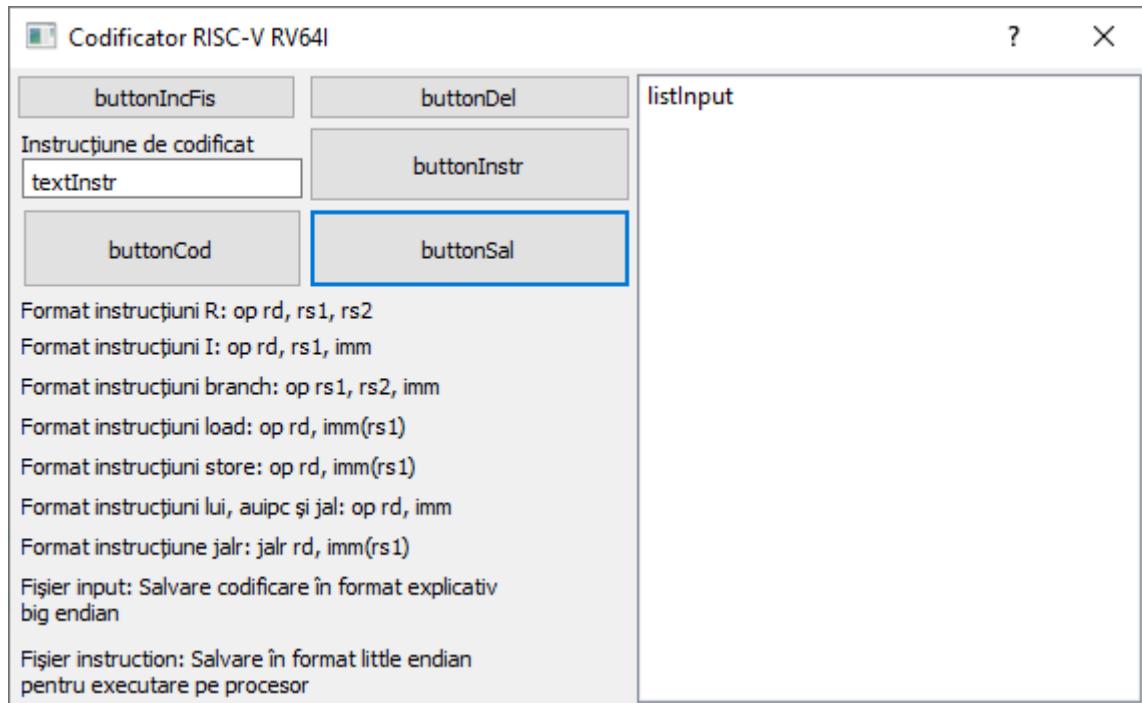


Figura 3.4. Elemente interactive codificator Python etichetate.

3.4. Implementarea emulatorului pe Raspberry Pi Pico

În continuare vor fi prezentate principalele funcții și metode ale clasei RISCVProcessor din fișierul RISC-Vpico.py. Codul complet al fișierului este disponibil în Anexa 6. Codul sursă al emulatorului pentru Raspberry Pi Pico.

3.4.1. Funcția sign_extend

```
#Functie de extindere a semnului unui numar la 64 de biti
def sign_extend(number, bits):
    mask = (1 << bits) - 1

    sign_bit = number & (1 << (bits - 1))

    if sign_bit:
        return number | (~mask & 0xFFFFFFFFFFFFFF)
    else:
        return number
```

3.4.2. Funcțiile limit_64_bits și limit_16_bits

```
#Functie de trunchiere a unui numar la 64 de biti
def limit64bits(number):
    return number&0xFFFFFFFFFFFFFF

#Functie de trunchiere a unui numar la 16 de biti
def limit16bits(number):
    return number&0xFFFF
```

3.4.3. Funcția `set_pins_from_result`

```
#Functia de control a LED-urilor, in functie de rezultatul unei instructiuni
def set_pins_from_result(result):
    gpio_pins = [machine.Pin(pin, machine.Pin.OUT) for pin in range(16)]

    for i in range(16):
        if result & (1 << i):
            gpio_pins[i].high()
        else:
            gpio_pins[i].low()
```

3.4.4. Constructorul clasei `RISC-VProcessor`

```
def __init__(self):
    self.pc = 0
    self.regdif = 0
    self.Jump = False
    self.jumpAddress = 0
    self.registers = [0] * 32
    self.dataMemory = [0] * 100
    self.memory = [0] * 176
    self.instruction = 0
    self.opcode = 0
    self.funct3 = 0
    self.funct7 = 0
    self.imm12 = 0
    self.shamt = 0
    self.rs1 = 0
    self.rs2 = 0
    self.rd = 0
    self.result = 0
    #Initializare memorie ROM
    with open("instructions.txt", "r") as f:
        hex_instructions = f.read().split()
        self.memory= hex_instructions
```

3.4.5. Metoda `execute`

```
#Functia de preluare din memorie si de executare a unei instructiuni
def execute(self):
    #Pentru a face o distinctie intre reinceperea programului, se afiseaza la
    #consola un spatiu, informatie folosita la debug
    if(self.pc==0):
        print()
    #Preluarea instructiunii din ROM, in functie de valoarea registrului PC
    instruction = (int(self.memory[self.pc+3],16)<<24)+
    (int(self.memory[self.pc+2],16)<<16)+(int(self.memory[self.pc+1],16)<<8)+
    (int(self.memory[self.pc],16))
    #Decodificarea instructiunii in componente
```

```

self.opcode = instruction & 0b1111111
self.funct3 = (instruction >> 12) & 0b111
self.funct7 = (instruction >> 25) & 0b1111111
self.imm12 = (instruction >> 20) & 0b111111111111
self.shamt = (instruction >> 20) & 0b1111
self.rs1 = (instruction >> 15) & 0b11111
self.rs2 = (instruction >> 20) & 0b11111
self.rd = (instruction >> 7) & 0b11111
#Executarea propriu-zisa a instructiunii, in functie de valorile decodificate
if self.opcode == 0b0110011: # Instructiuni de tip R

    if self.funct3 == 0b000: # ADD, SUB
        if self.funct7 == 0b0000000: #ADD
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] +
self.registers[self.rs2])
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct7 == 0b0100000: #SUB
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] -
self.registers[self.rs2])
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b001: # SLL
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] <<
(self.registers[self.rs2]&0b11111))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        [...]
    else:
        self.result=0
elif self.opcode == 0b0010011: # Instructiuni de tip I
    self.funct3 = self.funct3
    if self.funct3 == 0b000: # ADDI
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] +
sign_extend(self.imm12,12))
        self.result=limit16bits(self.registers[self.rd])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    [...]
    else:
        self.result=0

elif self.opcode == 0b0000011: # Instructiuni Load

    if self.funct3 == 0b000: # LB
        self.registers[self.rd]=sign_extend(self.dataMemory[limit64bits(self.registers[se
lf.rs1]+sign_extend(self.imm12,12))]&0xff,8)
        self.result=limit16bits(self.registers[self.rd])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    [...]
    else:

```

```

        self.result=0
    elif self.opcode == 0b0100011: # Instructiuni Store; Pentru aceste
instructiuni, imm12 se calculeaza intr-un mod diferit
        if self.funct3== 0b000: # SB
            self.imm12=((instruction>>25)<<7)|((instruction>>7)&0b11111)
self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]=self.registers[self.rs2]&0xff

    self.result=limit16bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign
_extend(self.imm12,12))])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    [...]
    else:
        self.result=0
    [...]
    elif self.opcode == 0b1100011:#Instructiuni branch; pentru aceste tipuri de
instructiuni, imm12 se calculeaza intr-un mod diferit
        self.imm12=((instruction>>31)<<12)|(((instruction>>7)&1)<<11)|((instruction>>25)&0b11111)<<5)|((instruction>>8)&0xf)<<1
        if self.funct3 == 0b000: #BEQ
            self.Jump=(self.registers[self.rs1]==self.registers[self.rs2])
            self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
            self.result=limit16bits(self.jumpAddress if self.Jump else 0)
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    [...]
    else:
        self.result=0
    else:
        self.result=0
#Verificarea flag-ului de salt
    if self.Jump:
        self.pc = self.jumpAddress
        self.Jump = False
    else:
        self.pc += 4
#Afisarea pe LED-uri a rezultatului
    set_pins_from_result(self.result)

```

În cadrul acestei metode se extrage din memoria ROM o instrucțiune și se execută, urmând un număr de pași asemănători, dar simplificați, cu varianta implementată în Verilog.

3.4.6. Metoda run

```

#Functia de executare a programului din memoria ROM
def run(self):
    while self.pc<len(self.memory):
        self.execute()
        time.sleep(0.3)

```

Metodă folosită pentru a executa instrucțiuni cât timp variabila PC indică o adresă validă din memorie.

3.4.7. Funcția main

```
def main():
    #Instantierea clasei si executarea functiei run
    processor = RISCVProcessor()
    processor.run()
```

Funcție în care se instanțiază clasa și se apelează funcția run a obiectului instanțiat.

3.5. Probleme întâmpinate și soluții de rezolvare

3.5.1. Probleme întâmpinate în timpul dezvoltării arhitecturii procesorului

În cursul dezvoltării arhitecturii procesorului au intervenit o serie de probleme ce au fost rezolvate.

3.5.1.1. Dificultatea ridicată a introducerii instrucțiunilor

La începutul dezvoltării procesorului, instrucțiunile nu erau citite din memorie ci erau un semnal de intrare pentru modulul RISC-VProcessor. Acest lucru făcea dificilă execuția procesorului pe Boolean Board pentru că nu pot fi introdusei cei 32 de biți în cadrul unui singur ciclu de ceas, placa având doar 16 comutatoare. Această problemă a fost rezolvată folosind memorii interne în care sunt stocate instrucțiunile.

3.5.1.2. Viteza de execuție a instrucțiunilor nu putea fi controlată ușor

Inițial modulul ClockDivider diviza frecvența clock a plăcii FPGA și nu putea fi controlată viteza cu care erau executate instrucțiunile decât dacă se reprograma placa. Pentru a putea vizualiza într-un mod mai controlat rezultatele instrucțiunilor a fost necesară introducerea unui nou semnal, CLK_BUTT, alături de automatul Moore pentru a putea controla din buton execuția instrucțiunilor.

3.5.1.3. Procesorul nu putea fi implementat din cauza dimensiunii memorii

La un moment dat numărul de componente LUT folosite de către procesor a depășit numărul de astfel de componente disponibile pe Boolean Board. Acest lucru împiedica implementarea procesorului pe placă. Pentru a rezolva această problemă a fost necesară reducerea dimensiunii pentru memoria RAM la un număr destul de mic de celule de memorie, dar și numărul de LUT utilizate a scăzut într-un mod acceptabil.

3.5.1.4. Memoria RAM are celule de o dimensiune anormală

Pe parcursul implementării memoria RAM avea celule de 64 biți pentru a facilita lucrul cu instrucțiunile de încărcare și stocare în memorie. Dar pentru a respecta standardele s-a trecut la o arhitectură pe 8 biți a celulelor de memorie și acest lucru a dus la o creștere a compatibilității cu alte procesoare RISC-V existente.

3.5.1.5. Lucrul cu memoriile se făcea în big-endian

Inițial, instrucțiunile erau citite din memoria ROM în format big endian și lucrul cu memoria RAM se făcea tot big endian. Procesorul a fost modificat pentru a lucra în format little endian din același motiv, pentru a crește compatibilitatea cu alte procesoare existente.

3.5.1.6. Instrucțiunile nu erau executate în pipeline ci secvențial

Procesorul RISC-V implementat inițial nu a fost creat folosind conceptele pipeline și execuția instrucțiunilor în mod secvențial. Acest lucru nu este de dorit deoarece execuția în pipeline a instrucțiunilor ajută la paraleлизarea unor operații ce trebuie execuțiate de procesor. Problema a fost rezolvată introducând cele cinci nivele de pipeline și tratarea hazardurilor ce puteau apărea în execuția în acest mod a instrucțiunilor.

3.5.2. Probleme întâmpinate în dezvoltarea codificatoarelor

Codificatoarele au urmat un proces similar de dezvoltare, la fiecare pas s-a adăugat suport pentru o nouă categorie de instrucțiuni. Problemele apărute pentru aceste programe sunt mai mult influențate de erorile pe care utilizatorii le pot face în timpul folosirii codificatoarelor.

3.5.2.1. Utilizatorul introduce o instrucțiune nesuportată de codificator

Inițial utilizatorul nu primea un feedback dacă apărea o problemă în codificarea instrucțiunilor. Pentru a transmite faptul că a apărut o problemă la codificarea instrucțiunii curente s-a introdus afișarea în MessageBox-uri a unui mesaj care descrie problema apărută.

3.5.2.2. Utilizatorul introduce numele alternative pentru regiștri

Codificatoarele acceptau în mod inițial doar notația cu x a regiștrilor. Acest lucru a fost rezolvat folosind un dicționar ce face legătura între numele oficial și notația alternativă.

3.5.2.3. Utilizatorul introduce o valoare eronată pentru regiștri

În acest caz o valoare eronată poate fi ori o valoare total separată de numele registrului ori o notație cu x urmată de un număr mai mare ca 31 sau mai mic ca 0. Pentru a semnaliza această problemă, utilizatorului îi va fi afișat faptul că valoarea introdusă pentru un registru este invalidă.

3.5.2.4. Utilizatorul introduce o valoare invalidă pentru valori imediate

Utilizatorul putea introduce valori imediate ce depășeau posibilitatea de codificare conform instrucțiunii, dar aceste valori erau trunchiate la dimensiunea maximă de biți disponibili. Pentru a atenționa utilizatorul că o valoare imediată nu este validă pentru instrucțiunea curentă se va afișa un MessageBox cu un mesaj ce descrie această situație.

3.5.2.5. La citirea unui fișier cu instrucțiuni nu se știe unde a apărut o problemă

Când utilizatorul încarcă un fișier cu instrucțiuni în asamblare nu se știe la ce instrucțiuni a apărut problema, iar restul instrucțiunilor erau codificate corect. Pentru a informa utilizatorul la ce instrucțiune a apărut o problemă se afișează un MessageBox cu instrucțiunea ce nu a putut fi codificată.

Capitolul 4. Testarea procesorului și anexelor și rezultate experimentale

4.1. Punerea în funcțiune

4.1.1. Punerea în funcțiune a procesorului implementat în Verilog

După ce au fost obținute resursele hardware descrise în 2.4.1. Componente hardware necesare, trebuie obținută suita software Vivado 2023.1 sau o versiune compatibilă cu aceasta. Se creează un proiect specific Boolean Board și a circuitului integrat al acestiei. Se introduce ca sursă de constrângeri fișierul constr.xdc. Ca surse de design se introduc fișierele design.v, data.mem și instructions.mem cu octetii instrucțiunilor de executat. Se trece prin procesul de Sintetizare, Implementare și Generare de bitstream și se programează placa FPGA conectată la calculator printr-un cablu microUSB folosind fișierul bitstream generat. Dacă programarea s-a făcut cu succes atunci poate începe execuția pe Boolean Board.

4.1.2. Punerea în funcțiune pentru codificatorul implementat în C#

Pentru a putea executa codificatorul implementat în C# este necesar IDE-ul Visual Studio 2019 sau o versiune compatibilă cu acesta, crearea unui proiect de tip Windows Forms App(.NET Framework) și modificarea fișierelor implicite pentru clasa moștenită din Form și clasa Designer. Mai trebuie introdusă icoana cropped-riscv-favicon-32x32.ico pentru a fi afișată de fereastră.

4.1.3. Punerea în funcțiune a codificatorului implementat în Python

Pentru a putea pune în funcțiune codificatorul implementat în Python, este necesară instalarea interpretorului Python și a modulelor sys, os și PyQt5. Se plasează fișierele main.py și main_window.py într-un director și se execută din interpretorul Python fișierul main.py.

4.1.4. Punerea în funcțiune a emulatorului pentru Raspberry Pi Pico

După procurarea resurselor hardware aferente emulatorului, descrise în 2.4.1. Componente hardware necesare, este necesară instalarea pe calculatorului a IDE-ului Thonny și instalarea mediului MicroPython pe Raspberry Pi Pico. Se conectează cele 16 leduri în ordinea dorită la pinii GPIO0-GPIO15 prin circuitele integrate ULN2003A și se salvează pe Pico fișierele RISC-Vpico.py redenumit în main.py (pentru a fi executat automat după deconectarea de la calculator) și fișierul instructions.txt completat cu octetii instrucțiunilor ce se doresc a fi executate. După încărcarea cu succes a fișierelor pe Raspberry Pi Pico se începe execuția programului încărcat și afișarea la leduri a rezultatelor instrucțiunilor.

4.2. Testarea hardware și software

4.2.1. Testarea procesorului implementat în Verilog

Pentru a testa procesorului implementat în Verilog s-a verificat valoarea registrului de ieșire result afișat pe leduri pentru fiecare instrucțiune din cele nouă programe al căror cod în asamblare RISC-V este disponibil în Anexa 2. Programele de test în limbaj de asamblare RISC-V.

4.2.2. Testarea codificatoarelor

Pentru testarea codificatoarelor a fost comparată codificarea generată de acestea cu o codificare obținută dintr-un codificator online.[23]

4.2.3. Testarea emulatorului

Pentru a testa emulatorul s-a încărcat un program pentru care se cunoaște secvența de rezultate pentru instrucțiuni și a fost verificată aprinderea ledurilor corespunzătoare rezultatelor corecte.

4.3. Date de test

Datele de test ale proiectului sunt reprezentate de nouă programare în asamblare RISC-V. Rolul acestora este de a verifica execuția corectă pe procesor a instrucțiunilor din care sunt formate. Aceste programe vor fi prezentate în următoarele sub-capitole, dar codul lor complet este disponibil în Anexa 2. Programele de test în limbaj de asamblare RISC-V.

4.3.1. program_add_lw.s

Este un program în care sunt folosite instrucțiunile addi, add, sw, lw și jalr. Se testează în acest mod operații simple și lucrul cu memoria RAM.

4.3.2. program_counter.s

În cadrul acestui program se execută o buclă în care registrul t1 joacă rolul unui counter descrescător, numărând de la 10 la 0. Sunt folosite instrucțiunile addi, sub, bne și jalr.

4.3.3. program_functie.s

Acest program prezintă un exemplu de funcție ce poate fi executată pe procesor. Folosește o multitudine de instrucțiuni, din care fac parte addi, sd, auipc și ld.

4.3.4. program_mare.s

Rolul acestui program este de a testa toate instrucțiunile procesorului, aşadar include toate instrucțiunile implementate. Este primul program dezvoltat pentru testare.

4.3.5. program_paritate.s

În cadrul acestui program se verifică paritatea valorii stocate într-un registru și în funcție de paritate se introduce o valoare în alt registru. Instrucțiunile utilizate sunt addi, andi, beq, jal și jalr.

4.3.6. program_procedura.s

Programul conține o procedură ce este executată. Conține instrucțiunile auipc, jal, addi, sw, lw și jalr.

4.3.7. program_sll_xor.s

Are rolul principal de a testa operațiile pe lucru cu biți. Conține un număr considerabil de instrucțiuni, cum ar fi addi, sll, xor, ori, bge, lui și jalr.

4.3.8. program_stocari_extrageri.s

Se testează în principal instrucțiunile de lucru cu memoria. Sunt folosite instrucțiunile addi, sb, sub, sh, sd, bltu, bgeu, add, lb, ld, lh, lbu, lhu, lwu și jalr.

4.3.9. program_suma.s

Conține o buclă în care într-un registru este calculată suma numerelor de la unu la patru. Instrucțiunile utilizate sunt addi, add, blt și jalr.

4.4. Rezultate experimentale

O reprezentare a semnalelor pentru rezultate obținute în urma execuției programelor a fost prezentată în 2.4.3.2. Scenarii de test pentru procesor, dar în continuare se vor explica câteva dintre valorile hexazecimale obținute în registrul result. Aceste valori au o frecvență diferită de apariție în funcție de hazardurile apărute în pipeline. Pentru codificatoare, rezultatele au fost reprezentate de codificarea acestor programe, iar pentru emulator rezultatul a fost secvența de leduri corespunzătoare valorii registrului result. Toate programele se încheie cu o instrucțiune jalr care va duce la valoarea 0 în result și va relua de la început execuția programului. De asemenea, la începutul execuției valoarea registrului result este 0 până când ajunge o instrucțiune cu un alt rezultat în etapa write-back.

4.4.1. program_add_lw.s

Se afișează valoarea 0x5 corespunzătoare primei instrucțiuni addi și valoarea 0x7 corespunzătoare celei de-a doua instrucțiune addi.

Pentru instrucțiunea add se face adunarea în hexazecimal a valorilor anterioare și rezultatul este 0xc, rezultat menținut și de instrucțiunea sw. Apoi rezultatul instrucțiunii addi este zero și rezultatul instrucțiunii lw este tot 0xc.

4.4.2. program_counters.s

După inițializarea din pipeline, apar următoarele rezultate:

- Valoarea hexazecimală 0xa este rezultatul instrucțiunii addi t1, zero, 10.
- Valoarea hexazecimală 0x1 este rezultatul instrucțiunii addi t2, zero, 1.
- Se intră în buclă, în care rezultatele sunt date de registrul t1, adresa de revenire la instrucțiunea de scădere(valoarea 0x8) și valoarea 0x0 generată de efectuarea saltului.

4.4.3. program_functie.s

Valoarea 0x60 este dată de diferența dintre valoarea inițială a registrului sp și 60 în baza 10.

Apoi se afișează valorile inițiale pentru regiștri ra și s0 atunci când sunt stocați în stivă. Urmează valoarea 0x80 reprezentată de revenirea registrului sp la valoarea inițială.

Valoarea 0x2 este dată de stocarea valorii 0x2 în registrul a0.

Valoarea 0x14 a rezultat în urma instrucțiunii auipc.

0x38 este valoarea adresei la care s-a făcut saltul, urmată de valoare 0x0 generată de încărcarea regiștrilor ra și s0 etc.

4.4.4. program_mare.s

Acest program are o execuție liniară, nu au loc salturi ce duc la bucle.

Valoarea 0xa este dată de instrucțiunea addi ra, zero, 10.

Valoarea 0x1e este dată de instrucțiunea addi sp, ra, 20.

Valoarea 0x28 este dată de instrucțiunea add gp, ra, sp.

Valoarea 0xa este generată de sub tp, gp, sp.

Instrucțiunea lui t0, 1 generează valoarea 0x1000.

Apoi din execuția instrucțiunii addi t1, zero, 50 rezultă 0x32.

Din and t2, t0, t1 rezultă valoarea 0x0.

xor t4, t3, t2 duce la valoarea 0x1000.

andi t5, t4, 3 va duce la 0x0 etc.

4.4.5. program_paritate.s

Execuția acestui program produce câteva valori utile.

Valoarea 0x9 este dată de instrucțiunea addi a0, zero, 9.

Valoarea 0x1 este dată de instrucțiunea andi t0, a0, 1.

Valoarea 0x0 este dată de instrucțiunea beq t0, zero, 12 pentru că nu se execută saltul.

Valoarea 0x3e7 este dată de instrucțiunea addi t0, zero, 999.

Valoarea 0x18 este dată de instrucțiunea jal t1, 8.

4.4.6. program_procedura.s

Inițial se execută auipc ra, 0 de unde rezultă valoarea 0x0, apoi instrucțiunea jal ra, 12 generează valoarea 0x10(0xc+0x4, unde 0x4 este valoarea anterioară a lui pc).

Valoarea 0x0 este generată de executarea unui salt, urmată de stocarea registrului ra pe stivă ce dă valoarea 0x7c.

0x5 este dat de addi a0, zero, 5 și 0x3 rezultă în urma addi a1, zero, 3.

După adunarea valorii 0x2 din instrucțiunea addi a0, a0, 2 la a0 rezultă 0x7.

Valoarea 0xa este dată de add a0, a0, a1, după rezolvarea hazardului de date.

Se încarcă valoarea stocată în stivă în ra, și rezultă valoarea 0x8.

Apoi se reface valoarea originală a lui sp și rezultatul este 0x80.

Se revine la instrucțiunea addi zero, zero, 0 aflată la adresa 0x8 etc.

4.4.7. program_sll_xors.s

Valoarea 0xa este generată de instrucțiunea addi gp, zero, 10, valoarea 0x2 de addi sp, zero, 2.

Valoarea 0x28 este rezultatul din instrucțiunea sll t0, gp, sp și reprezintă valoarea 0xa shiftată cu două poziții.

Se face xor gp, gp, sp și rezultă 0x8, echivalentă și cu valoarea generată de and t1, t0, gp.

Se obține 0x0 în urma xor gp, gp, gp și se obține 0x2 în urma or t2, zero, sp.

xori a0, zero, 4 duce la obținerea valorii 0x4, urmată de eliminarea valorii din t1 prin xor t1, t1, t1 ce dă rezultatul 0x0.

Apoi se obține valoarea 0x5 făcând ori a1, zero, 5.

Se introduce valoarea 0x4 în sp, shiftând la stânga. Apoi se shiftează la dreapta cu o poziție prima dată logic și apoi aritmetic rezultând 0x2 și 0x1 etc.

4.4.8. program_stocari_extrageri.s

Valoarea 0xffff este generată de instrucțiunea addi ra, zero, -1, valoarea 0x1 este rezultatul instrucțiunii addi gp, zero, 1.

0xff a fost generat în urma instrucțiunii sb ra, 0(zero) pentru că se stochează doar un

octet.

0xffffe a fost generat în urma instrucțiunii sub ra, ra, gp, reprezentând valoarea -2 în complement față de doi și este menținut de stocarea acestei valori determinate de instrucțiunea sh ra, 4(zero).

Se mai decrementează registrul ra și rezultă valoarea 0xffffd, ce este stocată folosind sd în memorie.

Se face un salt în urma căruia rezultă valoarea 0x24 și se ajunge la bltu gp, ra, 8 ce generează valoarea 0x2c. Apoi se citesc valorile din memorie etc.

4.4.9. program_suma.s

În urma executării instrucțiunii addi t0, zero, 5 rezultă valoarea 0x5, pentru addi t2, zero, 1 va fi valoarea 0x1, pentru add t1, t1, t2 valoarea 0x2 și se intră în buclă în care se afișează valorile adresei instrucțiunii add t1, t1, t2 și se afișează până când se iese din buclă, când se afișează valoarea finală a lui t1 în urma instrucțiunii addi t1, zero, 0.

4.5. Utilizarea sistemului

4.5.1. Utilizarea procesorului

După ce placa FPGA a fost programată folosind fișierul bitstream, utilizatorul poate vizualiza pe ledurile LD0-LD15 rezultatul instrucțiunilor în urma comutării manetei corespunzătoare semnalului CLK_BUTT.

4.5.2. Utilizarea codificatoarelor

Utilizatorul poate să codifice instrucțiunile introduse în textBoxInstr respectiv textInstr apăsând butonul etichetat „Codificare instrucțiune”.

Pentru a codifica un fișier cu instrucțiuni în asamblare RISC-V, utilizatorul poate apăsa butonul etichetat „Încărcare fișier” și se va parurge fișierul ales, linie cu linie.

Pentru a salva instrucțiunile codificate în format executabil pe procesor se apasă butonul etichetat „Salvare în fișier instruction” și se selectează locația unde se dorește salvarea fișierului.

Pentru a salva instrucțiunile în format explicativ, numit și format input, se apasă butonul numit „Salvare în fișier input” și se indică locația unde va fi salvat fișierul.

Pentru a elimina instrucțiunile deja codificate se apasă butonul „Eliminare conținut”.

4.5.3. Utilizarea emulatorului de pe Raspberry Pi Pico

După programarea plăcii Raspberry Pi Pico nu mai este nevoie de interacțiunea cu utilizatorul, ci se poate observa secvența de leduri corespunzătoare rezultatelor instrucțiunilor.

Concluzii

Prezenta lucrare de licență și-a propus implementarea unui procesor RISC-V folosind un FPGA, dezvoltarea codificatoarelor în C# și Python, dar și crearea emulatorului executat pe Raspberry Pi Pico. Obiectivele inițiale au fost îndeplinite, fiecare componentă fiind proiectată, implementată și testată conform ideilor inițiale.

Realizarea obiectivelor propuse

- Implementarea unui procesor RISC-V și executarea pe FPGA.
 - Procesorul a fost implementat în Verilog și încărcat cu succes pe Boolean Board. Verificarea execuției corecte a programelor de test a confirmat implementarea corectă a procesorului din punct de vedere al instrucțiunilor utilizate.
 - Utilizarea EDA Playground pentru simulare și Vivado pentru sinteză, implementare și programare a permis o evaluare detaliată a comportamentului hardware.
- Codificatorul de instrucțiuni în C#.
 - A fost dezvoltat pentru a facilita generarea de cod executabil pe procesor din instrucțiuni în asamblare RV64I.
 - Folosirea acestuia l-a dovedit a fi eficient și intuitiv, simplificând procesul de codificare pentru procesorul implementat.
- Codificatorul implementat în Python cu interfață grafică dezvoltată folosind Qt.
 - A oferit suport multiplatformă și a permis codificarea instrucțiunilor pe sisteme de operare cum ar fi Linux Mint.
- Emulatorul pentru procesorul RISC-V executat pe Raspberry Pi Pico.
 - A permis execuția și testarea codului într-un mediu diferit de FPGA, demonstrând o simulare corectă a comportamentului procesorului implementat.

Opinia personală și compararea cu alte proiecte

În opinia mea, rezultatele proiectului sunt remarcabile, demonstrând posibilitatea implementării unui procesor RISC-V cu pipeline pe un FPGA, argumentând fezabilitatea implementării unor codificatoare în C# și Python, și realizării unui emulator pe Raspberry Pi Pico.

Comparativ cu alte proiecte, procesorul are o structură mai compactă, fiind implementat în cadrul unui singur modul principal, dar oferă suport și pentru operații executate pe un număr mai mare de biți.

Utilizarea unui FPGA și a unui Raspberry Pi Pico a contribuit la utilizarea unui mediu optim pentru dezvoltare și testare, aceste dispozitive putând fi reprogramate în funcție de modificările suferite de arhitectură.

Posibile direcții viitoare de dezvoltare

Continuarea proiectului se poate face în mai multe direcții:

- Adăugarea de suport pentru noi instrucțiuni.
 - Arhitectura RISC-V prezintă o serie de extensii la standardul RV64I, cum ar fi extensia M pentru operații de înmulțire și împărțire, extensiile F și D pentru numere cu virgulă și extensia C pentru instrucțiuni comprimate.
 - Această direcție va duce în mod natural și la extinderea aplicațiilor anexe pentru a

adăuga suport pentru noile instrucțiuni implementate în procesorul de bază.

- Dezvoltarea unui sistem de operare minimal și executarea acestuia pe procesor.
 - Ar putea fi o direcție interesantă care să ofere suport pentru multitasking și gestionarea resurselor.
- Integrarea cu alte dispozitive hardware.
 - Extinderea compatibilității emulatorului și procesorului cu alte dispozitive hardware ar putea duce la includerea acestora în aplicații de o complexitate mai mare.
- Îmbunătățirea performanței.
 - Tratarea hazardurilor de date și control ce pot apărea în pipeline poate fi îmbunătățită comparativ cu varianta actuală în care se oprește execuția noilor instrucțiuni pentru hazardurile de date, iar instrucțiunile de salt duc la reluarea din fetch a următoarei instrucțiuni. Pentru hazardurile de date ar putea fi folosită o soluție de tip data forwarding, iar pentru hazardurile de control ar putea fi utilizată o soluție de predicție a ramurii.

Contribuții personale

Prin această lucrare am demonstrat posibilitatea implementării unui procesor RISC-V pe Boolean Board într-un mediu academic. Implementarea procesorului, dezvoltarea codificatoarelor și crearea emulatorului sunt contribuții semnificative ce pot constitui o bază a unor proiecte viitoare.

În concluzie, proiectul oferă o platformă robustă pentru explorarea și folosirea arhitecturii RISC-V, subliniind potențialul său de folosire în diverse domenii tehnologice. Implementarea pe FPGA împreună cu folosirea codificatoarelor și a emulatorului demonstrează flexibilitatea și modularitatea arhitecturii RISC-V în aplicații de cercetare și industriale. Această platformă permite studierea aprofundată a arhitecturii unui procesor RISC-V, facilitând experimentarea practică și oferind o bază solidă pentru dezvoltarea și testarea de noi instrucțiuni și funcționalități.

Bibliografie

- [1] Andrew Waterman, Krste Asanovic, SiFive Inc., CS Division, EECS Department, University of California, Berkeley, The RISC-V Instruction Set Manual Document Version 20191213 [Online], Disponibil la adresa: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZaurFLLww7ou5/view, Accesat: 2024.
- [2] suryakantamangaraj, Awesome RISC-V Resources [Online], Disponibil la adresa: <https://github.com/suryakantamangaraj/AwesomeRISC-VResources/blob/master/README.md>, Accesat: 2024.
- [3] SI-RISCV, Hummingbird E203 Opensource Processor Core [Online], Disponibil la adresa: https://github.com/SI-RISCV/e200_opensource/blob/master/README.md, Accesat: 2024.
- [4] jerryz123, The Berkeley Out-of-Order RISC-V Processor [Online], Disponibil la adresa: <https://github.com/riscv-boom/riscv-boom/blob/master/README.md>, Accesat: 2024.
- [5] JeanRochCoulon, CVA6 RISC-V CPU [Online], Disponibil la adresa: <https://github.com/openhwgroup/cva6/blob/master/README.md>, Accesat: .
- [6] SonalPinto, Kronos RISC-V [Online], Disponibil la adresa: <https://github.com/SonalPinto/kronos/blob/master/README.md>, Accesat: 2024.
- [7] mattvenn, PicoRV32 - A Size-Optimized RISC-V CPU [Online], Disponibil la adresa: <https://github.com/YosysHQ/picorv32/blob/main/README.md>, Accesat: 2024.
- [8] Du-Chao, Spike RISC-V ISA Simulator [Online], Disponibil la adresa: <https://github.com/riscv-software-src/riscv-isa-sim/blob/master/README.md>, Accesat: 2024.
- [9] sagark, FireSim: Fast and Effortless FPGA-accelerated Hardware Simulation with On-Prem and Cloud Flexibility [Online], Disponibil la adresa: <https://github.com/firesim/firesim/blob/main/README.md>, Accesat: 2024.
- [10] mortbopet, Ripes [Online], Disponibil la adresa: <https://github.com/mortbopet/Ripes/blob/master/README.md>, Accesat: 2024.
- [11] atishp04, Build Fedora Gnome Desktop on RISC-V!! [Online], Disponibil la adresa: <https://github.com/westerndigitalcorporation/RISC-V-Linux/blob/master/README.md>, Accesat: 2024.
- [12] --, About PULP [Online], Disponibil la adresa: <https://pulp-platform.org/projectinfo.html>, Accesat: 2024.
- [13] --, Verilog HDL [Online], Disponibil la adresa: <https://vlsiverify.com/verilog/>, Accesat: 2024.
- [14] BillWagner, A tour of the C# language [Online], Disponibil la adresa: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview>, Accesat: 2024.
- [15] AtmanAn, BeginnersGuide Overview [Online], Disponibil la adresa: <https://wiki.python.org/moin/BeginnersGuide/Overview>, Accesat: 2024.
- [16] --, Qt Framework [Online], Disponibil la adresa: <https://www.qt.io/product/framework>, Accesat: 2024.
- [17] --, A Brief Introduction to PyQt [Online], Disponibil la adresa: <https://www.python-me.org/a-brief-introduction-to-pyqt>, Accesat: 2024.
- [18] --, MicroPython [Online], Disponibil la adresa: <https://micropython.org/>, Accesat: 2024.
- [19] --, Boolean Board [Online], Disponibil la adresa: <https://www.realdigital.org/hardware/boolean>, Accesat: 2024.
- [20] --, Raspberry Pi Pico [Online], Disponibil la adresa: <https://www.raspberrypi.com/products/raspberry-pi-pico/>, Accesat: 2024.

- [21] --, Raspberry Pi Pico and Pico W [Online], Disponibil la adresa: <https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html>, Accesat: 2024.
- [22] --, Spartan 7 FPGAs [Online], Disponibil la adresa: <https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/spartan-7.html>, Accesat: 2024.
- [23] LupLab, RISC-V Instruction Encoder/Decoder [Online], Disponibil la adresa: <https://luplab.gitlab.io/rvcodecjs/>, Accesat: 2024.

Anexe.

Anexa 1. Schema bloc completă a modulului RISC-VProcessor

În figurile A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9 și A.10 este disponibilă schema bloc completă a modulului, generată de Vivado 2023.1.

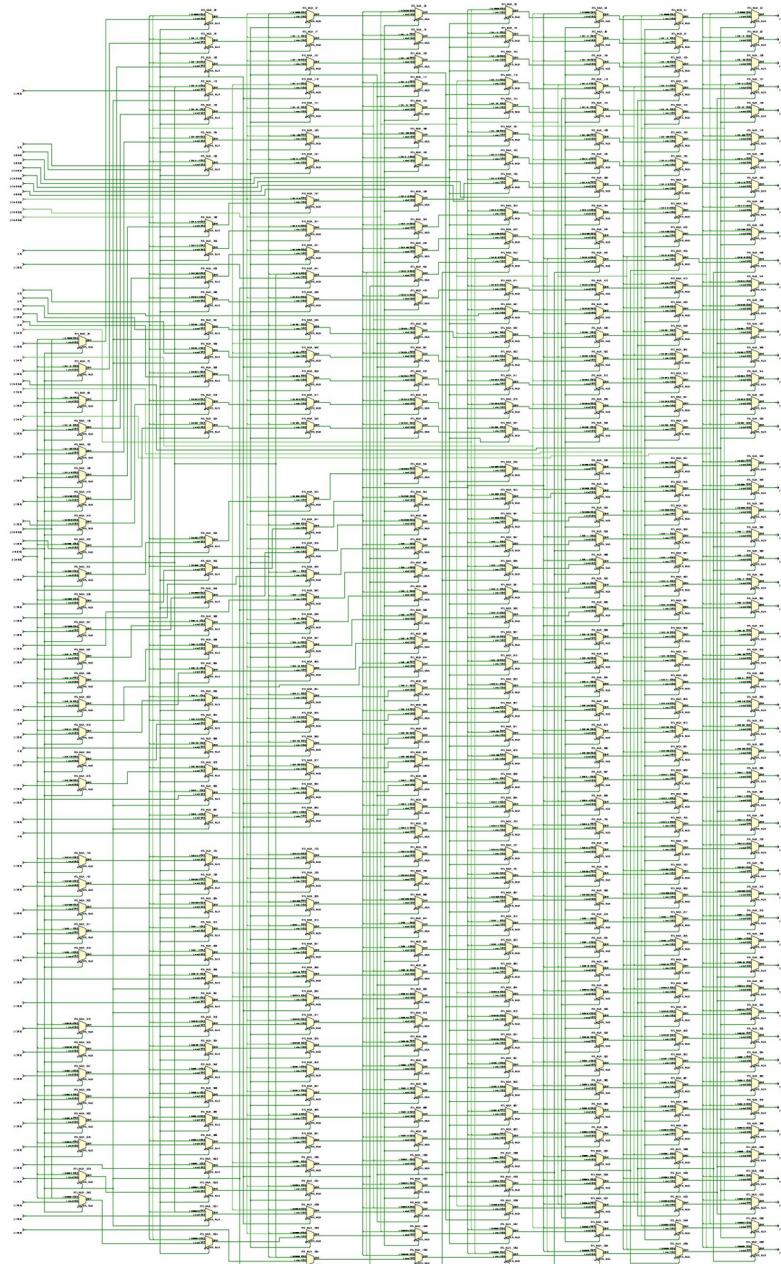


Figura A.1. Schema bloc a modulului RISC-VProcessor #a.

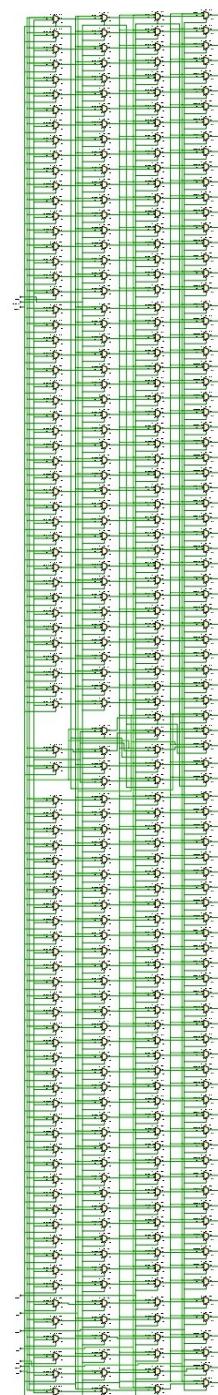


Figura A.2. Schema bloc a modulului RISC-VProcessor #b.

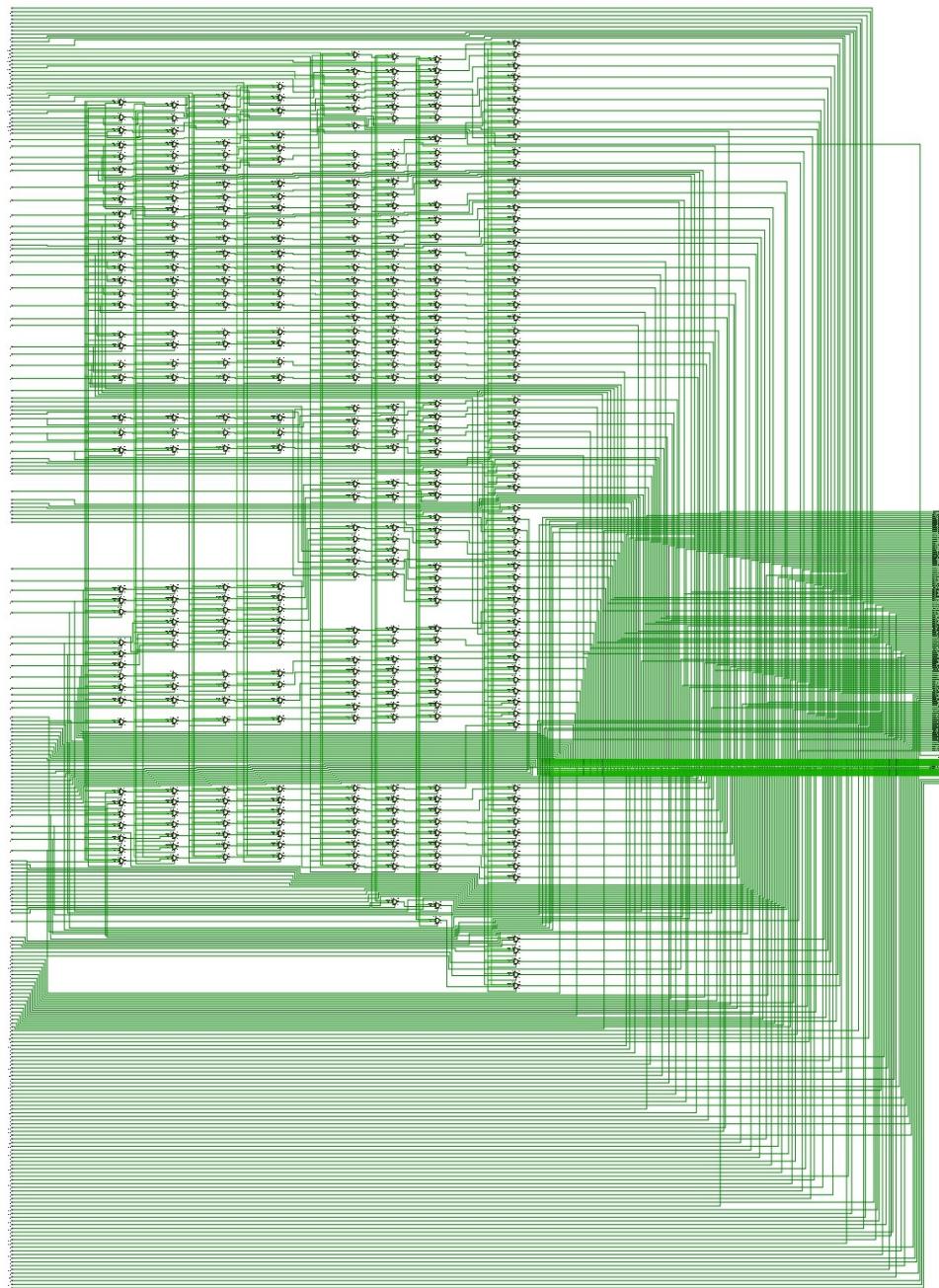


Figura A.3. Schema bloc a modulului RISC-VProcessor #c.

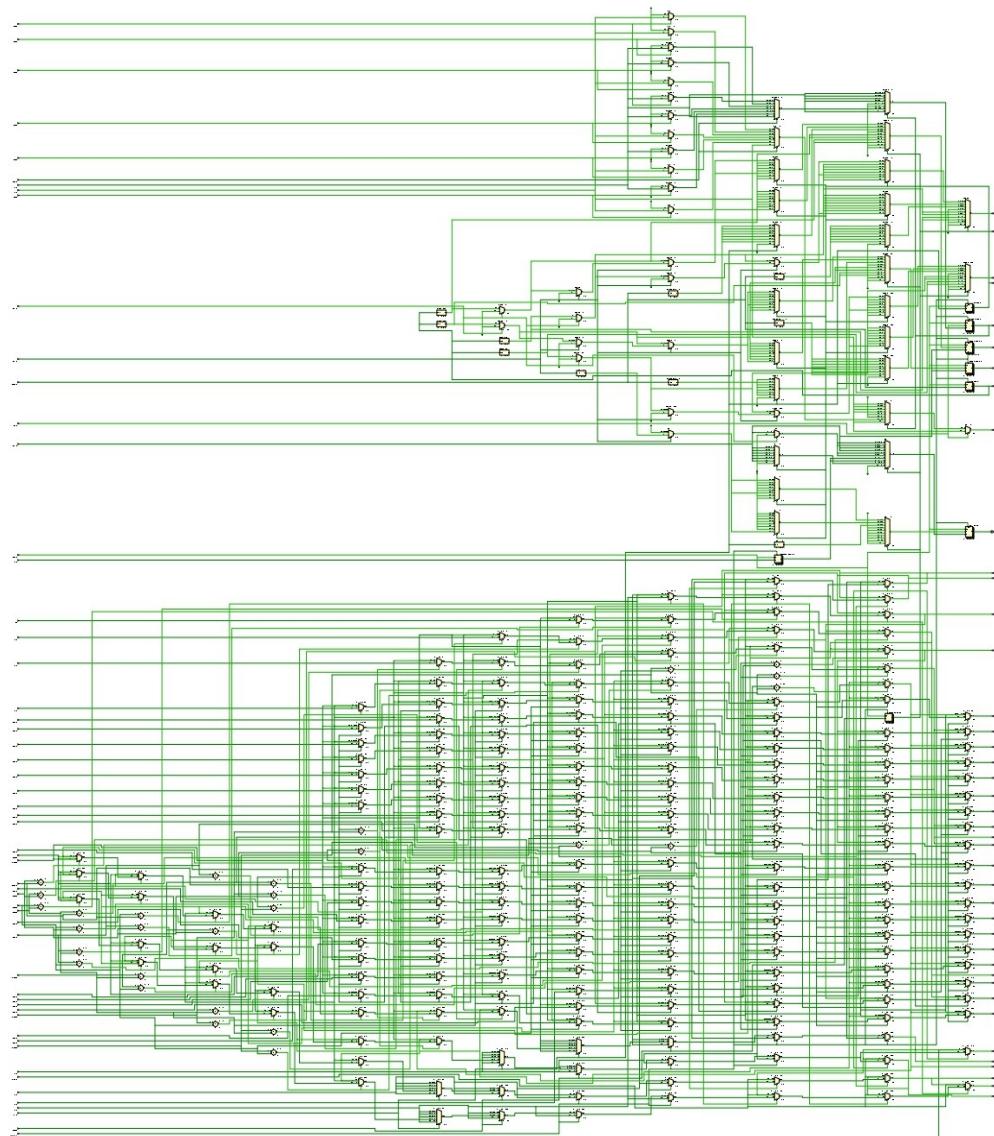


Figura A.4. Schema bloc a modulului RISC-VProcessor #d.

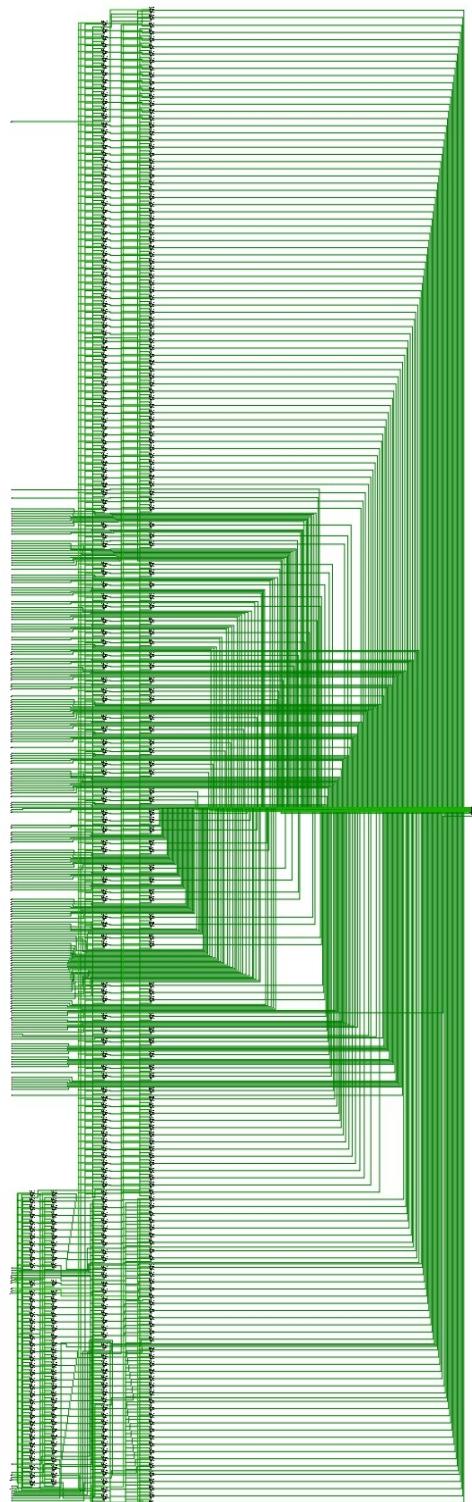


Figura A.5. Schema bloc a modulului RISC-VProcessor #e.

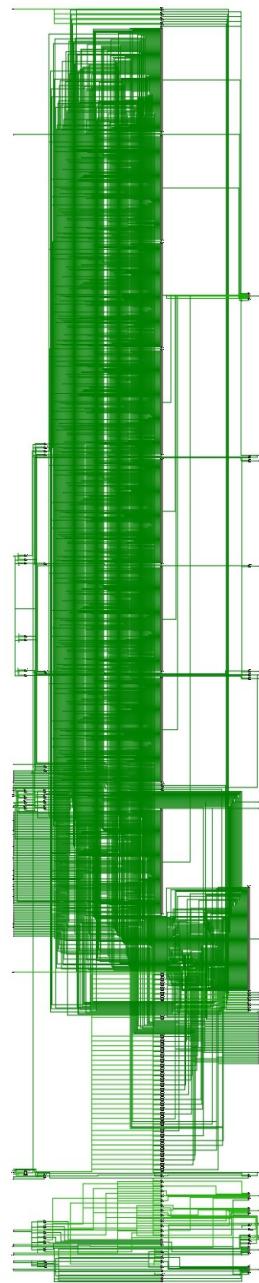


Figura A.6. Schema bloc a modulului RISC-VProcessor #f.

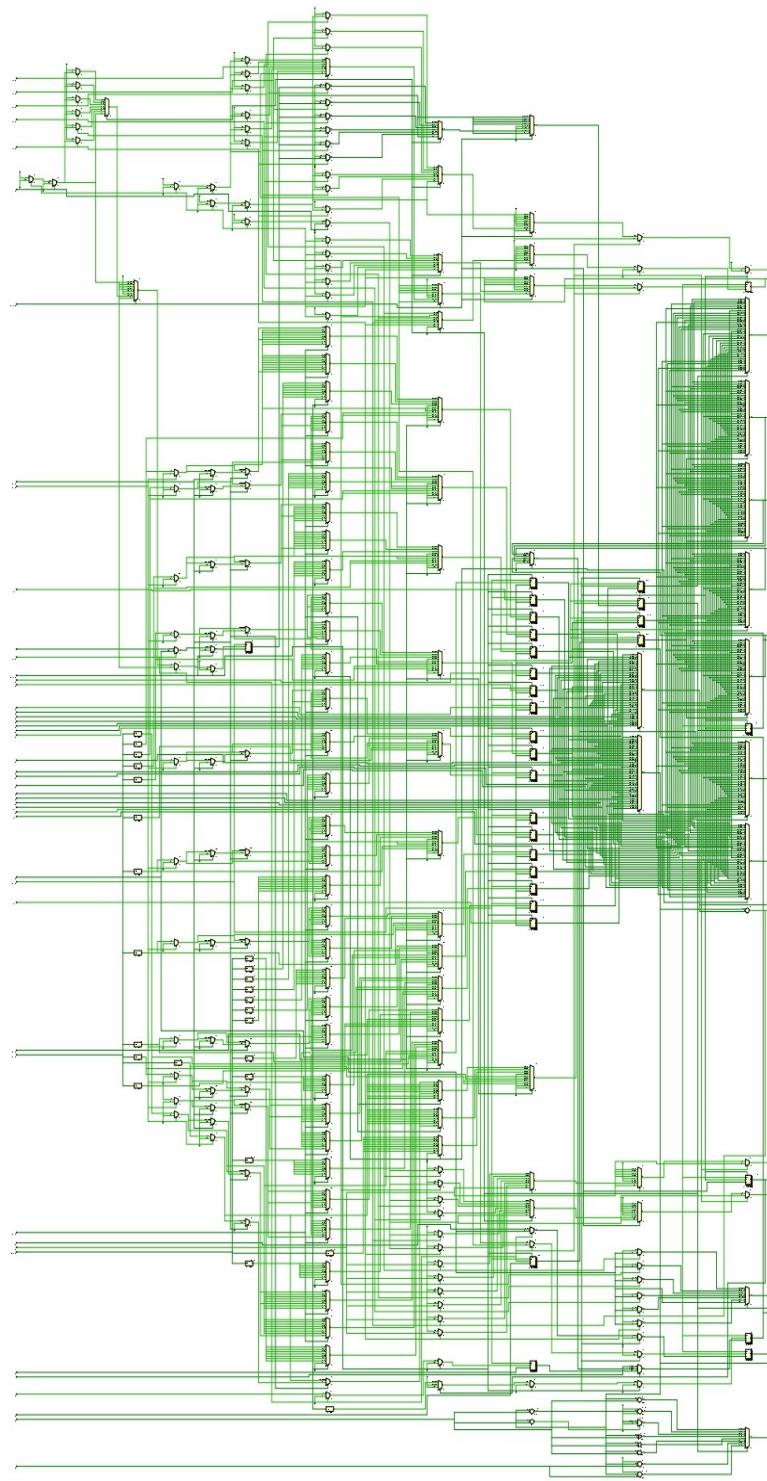


Figura A.7. Schema bloc a modulului RISC-VProcessor #g.

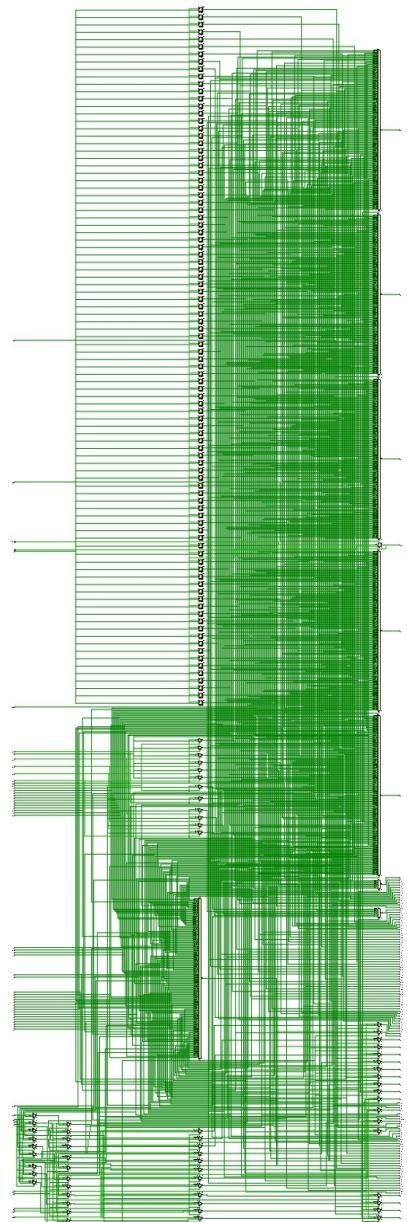


Figura A.8. Schema bloc a modulului RISC-VProcessor #h.

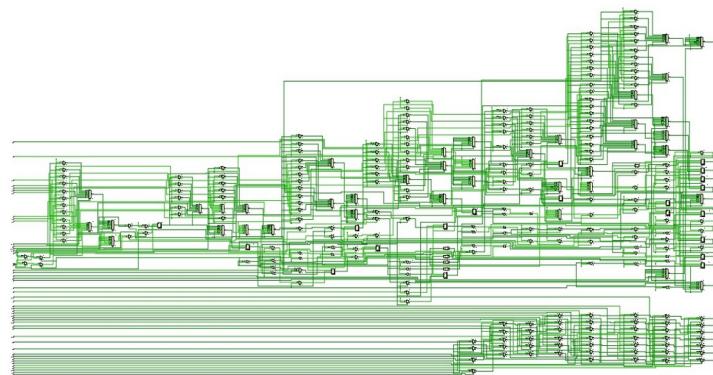


Figura A.9. Schema bloc a modulului RISC-VProcessor #i.

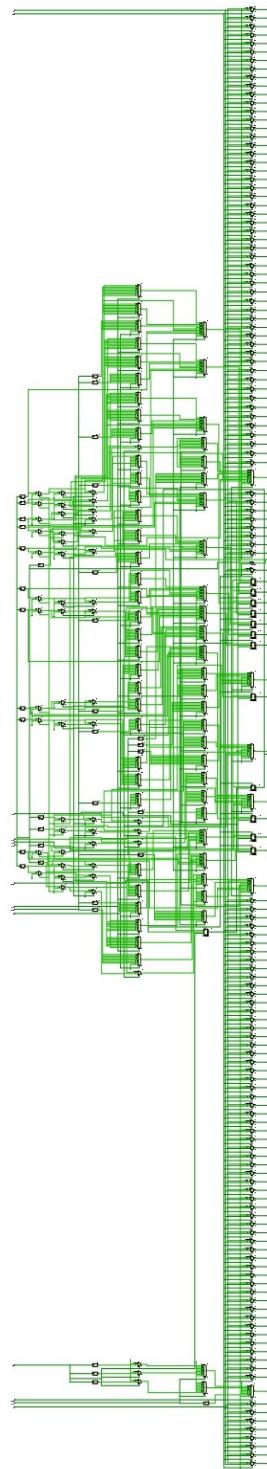


Figura A.10. Schema bloc a modulului RISC-VProcessor #j.

Revenire la secțiunea 2.4.2.2. Schema bloc simplificată a modulului RISC-VProcessor.

Anexa 2. Programele de test în limbaj de asamblare RISC-V

- program_add_lw.s

```
addi a0, zero, 5
addi a1, zero, 7
add a2, a0, a1
sw a2, 4(zero)
addi a2, a2, -12
lw a3, 4(zero)
jalr ra, 0(zero)
```

- program_counter.s

```
addi t1, zero, 10
addi t2, zero, 1
sub t1, t1, t2
bne t1, zero, -4
jalr ra, 0(zero)
```

- program_functie.s

```
addi sp, sp, -32
sd ra, 24(sp)
sd s0, 16(sp)
addi s0, sp, 32
addi a0, zero, 2
auipc x1, 0
jal x1, 32
sd a0, -24(s0)
addi a5, zero, 0
add a0, zero, a5
ld ra, 24(sp)
ld s0, 16(sp)
addi sp, sp, 32
jal ra, 56
addi sp, sp, -32
sd s0, 24(sp)
addi s0, sp, 32
sd a0, -24(s0)
add a4, zero, a0
add a5, zero, a4
slli a5, a5, 1
add a5, a5, a4
addi a5, a5, 1
add a0, zero, a5
ld s0, 24(sp)
addi sp, sp, 32
```

```
jal ra, -76
jalr ra, 0(zero)
```

- program_mare.s

```
addi ra, zero, 10
addi sp, ra, 20
add gp, ra, sp
sub tp, gp, sp
lui t0, 1
addi t1, zero, 50
and t2, t0, t1
or t3, t2, t0
xor t4, t3, t2
andi t5, t4, 3
ori t6, t5, 6
xori s0, t6, 9
srli s1, s0, 2
sll s2, s1, s1
srl s3, s2, s1
slli s1, s0, 2
sw gp, 0(zero)
lw a0, 0(zero)
slt t0, sp, ra
sltu t1, gp, s1
sra t2, s0, t1
slti a3, s1, 45
sltiu s2, s3, 4
srai a5, a0, 2
sb s3, 4(zero)
sh s0, 8(zero)
sd t3, 12(zero)
lb s9, 4(zero)
lh s7, 8(zero)
lbu t5, 4(zero)
lhu s7, 8(zero)
lwu a0, 0(zero)
ld t3, 12(zero)
jal s7, 8
jalr s10, 140(a5)
auipc s1, 20
beq sp, gp, 100
bne s1, a0, 4
blt s1, a0, 8
bge gp, a5, 4
bltu s1, a0, 10
bgeu gp, a5, 4
jalr ra, 0(zero)
addi zero, zero, 0
```

- program_paritate.s

```
addi a0, zero, 9
andi t0, a0, 1
beq t0, zero, 12
addi t0, zero, 999
jal t1, 8
addi t0, zero, 888
jalr ra, 0(zero)
```

- program_procedura.s

```
auipc ra, 0
jal ra, 12
addi zero, zero, 0
jal ra, 40
addi sp, sp, -4
sw ra, 0(sp)
addi a0, zero, 5
addi a1, zero, 3
addi a0, a0, 2
add a0, a0, a1
lw ra, 0(sp)
addi sp, sp, 4
jalr t0, 0(ra)
addi zero, zero, 0
jalr ra, 0(zero)
```

- program_sll_xor.s

```
addi gp, zero, 10
addi sp, zero, 2
sll t0, gp, sp
xor gp, gp, sp
and t1, t0, gp
xor gp, gp, gp
or t2, zero, sp
xori a0, zero, 4
xor t1, t1, t1
ori a1, zero, 5
slli sp, sp, 1
srli sp, sp, 1
srai sp, sp, 1
bge sp, zero, 4
add zero, zero, zero
lui sp, 3
slti a0, sp, -1
sltiu a1, sp, -1
```

```

xor sp, sp, sp
addi tp, zero, -2
slt a3, t0, tp
sltu a4, t0, tp
xor tp, tp, tp
addi a1, zero, 6
xor t0, t0, t0
srl a5, a1, a4
xor a5, a5, a5
xor a4, a4, a4
sra a1, a1, t2
xor t2, t2, t2
xor a1, a1, a1
jalr ra, 0(zero)

```

- program_stocari_extrageri.s

```

addi ra, zero, -1
addi gp, zero, 1
sb ra, 0(zero)
sub ra, ra, gp
sh ra, 4(zero)
sub ra, ra, gp
sd ra, 8(zero)
bltu gp, ra, 8
add zero, zero, zero
bgeu ra, gp, 8
add zero, zero, zero
lb a0, 0(zero)
ld a1, 8(zero)
lh a2, 4(zero)
lbu a3, 15(zero)
lhu a4, 0(zero)
lwu a5, 4(zero)
jalr ra, 0(zero)

```

- program_suma.s

```

addi t0, zero, 5
addi t2, zero, 1
add t1, t1, t2
addi t2, t2, 1
blt t2, t0, -8
addi t1, zero, 0
jalr ra, 0(zero)

```

Revenire la secțiunile 2.4.3.2. Scenarii de test pentru procesor, 4.2.1. Testarea procesorului implementat în Verilog și 4.3. Date de test.

Anexa 3. Codul complet al modulului RISC-VProcessor

```

module RISCVProcessor (
    input clk,
    input reset,
    input wire CLK_BUTT,
    output reg [15:0] result
);
    reg [1:0] bula;
    reg[63:0] pc,pc_jump;
    reg[63:0] reg_e,reg_m;
    reg[15:0] jumpAddress_e,jumpAddress_m;
    reg Jump;
    reg [63:0] registers [0:31];
    reg [8:0] dataMemory[0:128];
    reg [8:0] memory[0:175];
    reg [31:0] instruction_f, instruction_d,
instruction_e,instruction_m,instruction_wb;
    reg [6:0] opcode_d,opcode_e,opcode_m,opcode_wb;
    reg [2:0] funct3_d,funct3_e,funct3_m,funct3_wb;
    reg [6:0] funct7_d,funct7_e,funct7_m,funct7_wb;
    reg [11:0] imm12_d,imm12_e,imm12_m;
    reg [5:0] shamt_d,shamt_e,shamt_m;//Pentru RV64I, shamt are 6 biti
    reg [4:0] rs1_d,rs1_e,rs1_m;
    reg [4:0] rs2_d,rs2_e,rs2_m;
    reg [4:0] rd_d,rd_e,rd_m,rd_wb;
    integer i;
    initial begin
        //Citirea din fisier a continutului memoriei
        $readmemh("instructions.mem", memory);
        pc = 0; // Initializare cu 0
        //Initializare cu 0 banc de registre
        for (i = 0; i <= 31; i = i + 1)
            registers[i] = (i==2)?128:0;
        //Initializare cu 0 memorie de date
        $readmemh("data.mem", dataMemory);
        //Initializare registrii de lucru
        Jump=0;
        result=0;
        bula=0;
        jumpAddress_e=0;
        jumpAddress_m=0;
        instruction_f=32'hAAAAAAFF;
        instruction_d=32'hFFFFFFF;
        instruction_e=0;
        instruction_m=0;
        instruction_wb=0;
        opcode_d=0;
        funct3_d=0;
        funct7_d=0;
        imm12_d=0;

```

```

shamt_d=0;
rs1_d=0;
rs2_d=0;
rd_d=0;
opcode_e=0;
funct3_e=0;
funct7_e=0;
imm12_e=0;
shamt_e=0;
rs1_e=1;
rs2_e=2;
rd_e=3;
opcode_m=0;
funct3_m=0;
funct7_m=0;
imm12_m=0;
shamt_m=0;
rs1_m=4;
rs2_m=5;
rd_m=6;
opcode_wb=0;
funct3_wb=0;
funct7_wb=0;
rd_wb=0;
end
wire clk_div;
//Instantiere modul de control al clock-ului
ClockDivider clk_divider (
    .clk_in(clk),
    .clk_out(clk_div),
    .CLK_BUTT(CLK_BUTT)
);
always @(posedge clk_div) begin
    //Daca semnalul reset nu este activ, atunci continui operarea in pipeline.
    if(!reset) begin
        //Control al semnalului bula care se ocupă de gestiunea hazardurilor de date in
        pipeline.
        if(bula!=0 && Jump==0)
            bula<=bula-1;
        if (Jump==1) begin
            Jump<=0;
            pc<=pc_jump;
            end
        //Etapa fetch
        if(bula==0 && Jump==0)begin
            instruction_f <= {memory[pc+3][7:0], memory[pc+2][7:0], memory[pc+1][7:0],
            memory[pc][7:0]};
            pc<=pc+4;
            end
        //Etapa decode
        if(bula==0)begin

```

```

    if(Jump==0)begin
instruction_d<=instruction_f;
opcode_d <= instruction_f[6:0];
funct3_d <= instruction_f[14:12];
funct7_d <= instruction_f[31:25];
imm12_d <= instruction_f[31:20];
shamt_d <= instruction_f[25:20];
rs1_d <= instruction_f[19:15];
rs2_d <= instruction_f[24:20];
rd_d <= instruction_f[11:7];
    //Tratare hazard date
    if(rs1_d == rd_e || rs2_d == rd_e)
        bula<=3'b010;
    else if(rs1_d == rd_m || rs2_d == rd_m)
        bula<=3'b001;
    end
end
    //Etapa execute
if(bula<1 && Jump==0) begin
opcode_e <= opcode_d;
funct3_e <= funct3_d;
funct7_e <= funct7_d;
imm12_e <= imm12_d;
shamt_e <= shamt_d;
rs1_e <= rs1_d;
rs2_e <= rs2_d;
rd_e <= rd_d;
instruction_e<=instruction_d;
    //Stabilirea operatiei de executat in functie de parametrii decodificati.
case (opcode_e)
    7'b0110011: begin
        // Instructiuni de tip R
        case (funct3_e)
            3'b000: begin // ADD, SUB
                if (funct7_e == 7'b0000000)
                    reg_e<= registers[rs1_e] + registers[rs2_e];
                else if(funct7_e == 7'b0100000)
                    reg_e<= registers[rs1_e] - registers[rs2_e];
            end
            3'b001: // SLL pentru RV64I
                reg_e <= registers[rs1_e] << registers[rs2_e][5:0];
            3'b010: begin // SLT
                reg_e <= (registers[rs1_e] - registers[rs2_e])>>63;
            end
            3'b011: begin // SLTU
                reg_e <= (registers[rs1_e] < registers[rs2_e]) ? 1 : 0;
            end
            3'b100: // XOR
                reg_e<= registers[rs1_e] ^ registers[rs2_e];
            3'b101: begin
                if (funct7_e == 7'b0000000) // SRL pentru RV64I

```

```

        reg_e <= registers[rs1_e] >> registers[rs2_e][5:0];
else if (funct7_e == 7'b0100000) // SRA pentru RV64I
    reg_e <= registers[rs1_e] >>> registers[rs2_e][5:0];
end
3'b110: // OR
    reg_e <= registers[rs1_e] | registers[rs2_e];
3'b111: // AND
    reg_e <= registers[rs1_e] & registers[rs2_e];
endcase
end
7'b0010011: begin
    // Instructiuni de tip I
    case (funct3_e)
        3'b000: // ADDI
            reg_e <= registers[rs1_e] + {{52{imm12_e[11]}},imm12_e};
        3'b010: begin // SLTI
            reg_e <= (registers[rs1_e]-{{52{imm12_e[11]}},imm12_e})>>63;
        end
        3'b011: begin // SLTIU
            reg_e <=(registers[rs1_e] < {{52{imm12_e[11]}},imm12_e}) ? 1 : 0;
        end
        3'b100: begin // XORI
            reg_e <= registers[rs1_e] ^ {{52{imm12_e[11]}},imm12_e};
        end
        3'b110: begin // ORI
            reg_e <= registers[rs1_e] | {{52{imm12_e[11]}},imm12_e};
        end
        3'b111: begin // ANDI
            reg_e <= registers[rs1_e] & {{52{imm12_e[11]}},imm12_e};
        end
        3'b001: begin // SLLI
            reg_e <= registers[rs1_e] << shamt_e;
        end
        3'b101: begin
            if (funct7_e == 7'b0000000)
                begin // SRLI
                    reg_e <= registers[rs1_e] >> shamt_e;
                end
            else if (funct7_e == 7'b0100000)
                begin // SRAI
                    reg_e <= registers[rs1_e] >>> shamt_e;
                end
            end
        endcase
    end
    7'b0110111: begin // LUI
        reg_e <= {{32{instruction_e[31]}},instruction_e[31:12], 12'b0};
    end
    7'b0010111: begin // AUIPC
        reg_e <= pc - 12 + {{32{instruction_e[31]}},instruction_e[31:12],
12'b0}; // Valoare PC de la fetch
    end

```

```

    end
  7'b1101111: begin // JAL
    Jump <= 1;
    reg_e<= pc - 8;
    //In cadrul acestei instructiuni, se vor adauga instructiuni nop si se va continua cu etapa fetch a instructiunii
    jumpAddress_e<=(pc-12)+{{43{instruction_e[31]}},
{instruction_e[31],instruction_e[19:12],instruction_e[20],instruction_e[30:21]},1'b0};
    pc_jump<=(pc-12)+{{43{instruction_e[31]}},
{instruction_e[31],instruction_e[19:12],instruction_e[20],instruction_e[30:21]},1'b0};
    //Initializare cu valori pentru a nu executa in mod eronat instructiuni.
    instruction_f<=32'hAAAAAAFF;
    instruction_d<=32'hFFFFFFFF;
    opcode_d<=0;
    rd_d<=0;
    funct3_d<=0;
    rs1_d<=0;
    rs2_d<=0;
    funct7_d<=0;
    shamt_d<=0;
    imm12_d<=0;
    instruction_e<=0;
    opcode_e<=0;
    rd_e<=3;
    funct3_e<=0;
    rs1_e<=1;
    rs2_e<=2;
    funct7_e<=0;
    shamt_e<=0;
    imm12_e<=0;
  end
  7'b1100111: begin // JALR
    //Aceasta instructiune functioneaza similar cu precedenta, dar modul de calcul al adresei de salt este diferit.
    reg_e<= pc - 8;
    Jump <= 1;
    jumpAddress_e<=registers[rs1_e]+{{52{imm12_e[11]}},imm12_e[11:1],1'b0};
    pc_jump<=registers[rs1_e]+{{52{imm12_e[11]}},imm12_e[11:1],1'b0};
    instruction_f<=32'hBBBBBBFF;
    instruction_d<=32'hFFFFFFFF;
    opcode_d<=0;
    rd_d<=0;
    funct3_d<=0;
    rs1_d<=0;
    rs2_d<=0;
    funct7_d<=0;
    shamt_d<=0;
    imm12_d<=0;
    instruction_e<=0;

```

```

        opcode_e<=0;
        rd_e<=31;
        funct3_e<=0;
        rs1_e<=1;
        rs2_e<=2;
        funct7_e<=0;
        shamt_e<=0;
        imm12_e<=0;
    end
    7'b1100011: begin
        // Instructiuni branch
        case (funct3_e)
            //Pentru aceste instructiuni, daca saltul se face, se va proceda similar
            // ca in cazul salturilor neconditionate.
            3'b000: begin // BEQ
                if(registers[rs1_e] == registers[rs2_e]) begin
                    Jump<=1;
                    reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
                    pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
                    instruction_f<=32'hAAAAAAFF;
                    instruction_d<=32'hFFFFFFF;
                    opcode_d<=0;
                    rd_d<=0;
                    funct3_d<=0;
                    rs1_d<=0;
                    rs2_d<=0;
                    funct7_d<=0;
                    shamt_d<=0;
                    imm12_d<=0;
                    instruction_e<=0;
                    opcode_e<=0;
                    rd_e<=3;
                    funct3_e<=0;
                    rs1_e<=1;
                    rs2_e<=2;
                    funct7_e<=0;
                    shamt_e<=0;
                    imm12_e<=0;
                end
            end
            else
                reg_e<=0;
        end
        3'b001: begin // BNE
            if(registers[rs1_e] != registers[rs2_e]) begin
                Jump<=1;
                reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst

```

```

ruction_e[11:8],1'b0);
            pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
            instruction_f<=32'hAAAAAAFF;
instruction_d<=32'hFFFFFFF;
    opcode_d<=0;
    rd_d<=0;
    funct3_d<=0;
    rs1_d<=0;
    rs2_d<=0;
    funct7_d<=0;
    shamt_d<=0;
    imm12_d<=0;
    instruction_e<=0;
    opcode_e<=0;
    rd_e<=3;
    funct3_e<=0;
    rs1_e<=1;
    rs2_e<=2;
    funct7_e<=0;
    shamt_e<=0;
    imm12_e<=0;
    end
    else
        reg_e<=0;
    end
3'b100: begin // BLT
    if((registers[rs1_e] - registers[rs2_e])>>63) begin
        Jump<=1;
        reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
            pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
            instruction_f<=32'hAAAAAAFF;
instruction_d<=32'hFFFFFFF;
    opcode_d<=0;
    rd_d<=0;
    funct3_d<=0;
    rs1_d<=0;
    rs2_d<=0;
    funct7_d<=0;
    shamt_d<=0;
    imm12_d<=0;
    instruction_e<=0;
    opcode_e<=0;
    rd_e<=3;
    funct3_e<=0;
    rs1_e<=1;

```

```

rs2_e<=2;
funct7_e<=0;
shamt_e<=0;
imm12_e<=0;
    end
    else
        reg_e<=0;
    end
3'b101: begin // BGE
    if(!((registers[rs1_e] - registers[rs2_e])>>63)) begin
        Jump<=1;
        reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
        pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
            instruction_f<=32'hAAAAAAFF;
        instruction_d<=32'hFFFFFFF;
        opcode_d<=0;
        rd_d<=0;
        funct3_d<=0;
        rs1_d<=0;
        rs2_d<=0;
        funct7_d<=0;
        shamt_d<=0;
        imm12_d<=0;
        instruction_e<=0;
        opcode_e<=0;
        rd_e<=3;
        funct3_e<=0;
        rs1_e<=1;
        rs2_e<=2;
        funct7_e<=0;
        shamt_e<=0;
        imm12_e<=0;
    end
    else
        reg_e<=0;
end
3'b110: begin // BLTU
    if(registers[rs1_e] < registers[rs2_e]) begin
        Jump<=1;
        reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
        pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],inst
ruction_e[11:8],1'b0};
            instruction_f<=32'hAAAAAAFF;
        instruction_d<=32'hFFFFFFF;

```

```

opcode_d<=0;
rd_d<=0;
funct3_d<=0;
rs1_d<=0;
rs2_d<=0;
funct7_d<=0;
shamt_d<=0;
imm12_d<=0;
instruction_e<=0;
opcode_e<=0;
rd_e<=3;
funct3_e<=0;
rs1_e<=1;
rs2_e<=2;
funct7_e<=0;
shamt_e<=0;
imm12_e<=0;
    end
    else
        reg_e<=0;
    end
3'b111: begin // BGEU
    if(registers[rs1_e] >= registers[rs2_e]) begin
        Jump<=1;
        reg_e<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],instruction_e[11:8],1'b0};
        pc_jump<=pc-12+
{{52{instruction_e[31]}},instruction_e[31],instruction_e[7],instruction_e[30:25],instruction_e[11:8],1'b0};
        instruction_f<=32'hAAAAAAFF;
        instruction_d<=32'hFFFFFFF;
        opcode_d<=0;
        rd_d<=0;
        funct3_d<=0;
        rs1_d<=0;
        rs2_d<=0;
        funct7_d<=0;
        shamt_d<=0;
        imm12_d<=0;
        instruction_e<=0;
        opcode_e<=0;
        rd_e<=3;
        funct3_e<=0;
        rs1_e<=1;
        rs2_e<=2;
        funct7_e<=0;
        shamt_e<=0;
        imm12_e<=0;
    end
    else

```

```

        reg_e<=0;
    end
endcase
end
endcase
end
//Etapa memory
jumpAddress_m<=jumpAddress_e;
if(bula!=1) begin
    opcode_m <= opcode_e;
funct3_m <= funct3_e;
funct7_m <= funct7_e;
imm12_m <= imm12_e;
shamt_m <= shamt_e;
rs1_m <= rs1_e;
rs2_m <= rs2_e;
rd_m <= rd_e;
instruction_m<=instruction_e;
//Pentru aceasta etapa, se executa operatii specifice pentru instructiunile
de load si store in memorie. Restul instructiunilor doar vor primi in reg_m valoarea
anterioara din reg_e.
case (opcode_m)
 7'b0000011: begin
    // Instructiuni load
    case (funct3_m)
      3'b000: begin // LB
        reg_m <= {{56{dataMemory[registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m]}[7]}},dataMemory[registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0];
      end
      3'b001: begin // LH
        reg_m <= {{48{dataMemory[1+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7]}},dataMemory[1+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0]};
      end
      3'b010: begin // LW
        reg_m <= {{32{dataMemory[3+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7]}},dataMemory[3+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[2+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[1+registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[registers[rs1_m]+
{{52{imm12_m[11]}}},imm12_m][7:0]};
      end
      3'b011: begin // LD
        reg_m<= {dataMemory[7+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[6+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],
dataMemory[5+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[4+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],
dataMemory[3+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[2+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],
dataMemory[1+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0],dataMemory[0+registers[rs1_m]+{{52{imm12_m[11]}}},imm12_m][7:0]};
      end
    endcase
  end
end

```

```

[7:0],dataMemory[registers[rs1_m]+{{52{imm12_m[11]}},imm12_m}][7:0];
    end
    3'b100: begin // LBU
        reg_m <= {56'b0,dataMemory[registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0];
    end
    3'b101: begin // LHU
        reg_m <= {48'b0,dataMemory[1+registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0],dataMemory[registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0];
    end
    3'b110: begin // LWU
        reg_m <= {32'b0,dataMemory[3+registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0],dataMemory[2+registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0],dataMemory[1+registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0],dataMemory[registers[rs1_m]+
{{52{imm12_m[11]}},imm12_m}][7:0];
    end
    endcase
end
7'b0100011: begin
    // Instrucțiuni store
    case (funct3_m)
        3'b000: begin // SB
            dataMemory[registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]} <= registers[rs2_m][7:0];
            reg_m<=registers[rs2_m][7:0];
        end
        3'b001: begin // SH
            {dataMemory[1+registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]}[7:0],dataMemory[registers[rs1_m]+
{{52{instruction_m[31]}},{instruction_m[31:25],instruction_m[11:7]}]}[7:0]} <=
registers[rs2_m][15:0];
            reg_m<=registers[rs2_m][15:0];
        end
        3'b010: begin // SW
            {dataMemory[3+registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]}[7:0],dataMemory[2+registers[rs1_m]+
{{52{instruction_m[31]}},{instruction_m[31:25],instruction_m[11:7]}]}
[7:0],dataMemory[1+registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]}[7:0],dataMemory[registers[rs1_m]+
{{52{instruction_m[31]}},{instruction_m[31:25],instruction_m[11:7]}]}[7:0]} <=
registers[rs2_m][31:0];
            reg_m<= registers[rs2_m][31:0];
        end
        3'b011: begin // SD
            {dataMemory[7+registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]}[7:0],dataMemory[6+registers[rs1_m]+
{{52{instruction_m[31]}},{instruction_m[31:25],instruction_m[11:7]}]}
[7:0],dataMemory[5+registers[rs1_m]+{{52{instruction_m[31]}},
{instruction_m[31:25],instruction_m[11:7]}]}[7:0],dataMemory[4+registers[rs1_m]+
{{52{instruction_m[31]}},{instruction_m[31:25],instruction_m[11:7]}]}]

```

```

[7:0],dataMemory[3+registers[rs1_m]+{{52{instruction_m[31]}}},  

{instruction_m[31:25],instruction_m[11:7]}][7:0],dataMemory[2+registers[rs1_m]+  

{{52{instruction_m[31]}}},{instruction_m[31:25],instruction_m[11:7]}]  

[7:0],dataMemory[1+registers[rs1_m]+{{52{instruction_m[31]}}},  

{instruction_m[31:25],instruction_m[11:7]}][7:0],dataMemory[registers[rs1_m]+  

{{52{instruction_m[31]}}},{instruction_m[31:25],instruction_m[11:7]}][7:0}] =  

registers[rs2_m];  

    reg_m<=registers[rs2_m];  

end  

default: reg_m<=0;  

endcase  

end  

default: reg_m<=reg_e;  

endcase  

end  

//Etapa write back  

opcode_wb<=opcode_m;  

funct3_wb <= funct3_m;  

funct7_wb <= funct7_m;  

rd_wb <= rd_m;  

instruction_wb<=instruction_m;  

//Este etapa in care se scrie rezultatul operatiei in result si in registrul  

destinatie.  

case (opcode_wb)  

7'b0110011: begin  

// Instructiuni de tip R  

case (funct3_wb)  

3'b000: begin // ADD, SUB  

if (funct7_wb == 7'b0000000)  

begin  

if (rd_wb!=0) begin  

registers[rd_wb]<= reg_m;  

result<=reg_m;  

end  

else  

result<=0;  

end  

else if (funct7_wb == 7'b0100000)  

begin  

if (rd_wb!=0) begin  

registers[rd_wb]<= reg_m;  

result<=reg_m;  

end  

else  

result<=0;  

end  

end  

3'b001: begin // SLL pentru RV64I  

if (rd_wb!=0) begin  

registers[rd_wb]<= reg_m;  

result<=reg_m;

```

```

    end
  else
    result<=0;
  end
3'b010: begin // SLT
  if (rd_wb!=0) begin
    registers[rd_wb] <= reg_m;
  result<= reg_m;
  end
  else
    result<=0;
  end
3'b011: begin // SLTU
  if (rd_wb!=0) begin
    registers[rd_wb] <= reg_m;
  result<=reg_m;
  end
  else
    result<=0;
  end
3'b100: begin // XOR
  if (rd_wb!=0) begin
    registers[rd_wb]<= reg_m;
    result<=reg_m;
  end
  else
    result<=0;
  end
3'b101: begin
  if (funct7_wb == 7'b0000000) begin // SRL pentru RV64I
    if (rd_wb!=0) begin
      registers[rd_wb]<= reg_m;
      result<=reg_m;
    end
    else
      result<=0;
  end else if (funct7_wb == 7'b0100000) begin // SRA pentru RV64I
    if (rd_wb!=0) begin
      registers[rd_wb]<= reg_m;
      result<=reg_m;
    end
    else
      result<=0;
  end
end
3'b110: begin // OR
  if (rd_wb!=0) begin
    registers[rd_wb]<= reg_m;
    result<=reg_m;
  end
  else

```

```

                result<=0;
            end
            3'b111: begin // AND
                if (rd_wb!=0) begin
                    registers[rd_wb]<= reg_m;
                    result<=reg_m;
                end
                else
                    result<=0;
            end
            default: result <= 0;
        endcase
    end
    7'b0010011: begin
        // Instructiuni de tip I
        case (funct3_wb)
            3'b000: begin // ADDI
                if (rd_wb!=0) begin
                    registers[rd_wb] <= reg_m;
                    result<=reg_m;
                end
                else
                    result<=0;
            end
            3'b010: begin // SLTI
                if (rd_wb!=0) begin
                    registers[rd_wb]<= reg_m;
                    result<=reg_m;
                end
                else
                    result<=0;
            end
            3'b011: begin // SLTIU
                if (rd_wb!=0) begin
                    registers[rd_wb]<= reg_m;
                    result<=reg_m;
                end
                else
                    result<=0;
            end
            3'b100: begin // XORI
                if (rd_wb!=0) begin
                    registers[rd_wb]<= reg_m;
                    result<=reg_m;
                end
                else
                    result<=0;
            end
            3'b110: begin // ORI
                if (rd_wb!=0) begin
                    registers[rd_wb]<= reg_m;

```

```

        result<=reg_m;
    end
    else
        result<=0;
    end
3'b111: begin // ANDI
    if (rd_wb!=0) begin
        registers[rd_wb]<= reg_m;
        result<=reg_m;
    end
    else
        result<=0;
    end
3'b001: begin // SLLI
    if (rd_wb!=0) begin
        registers[rd_wb]<= reg_m;
        result<=reg_m;
    end
    else
        result<=0;
    end
3'b101: begin
    if (funct7_wb == 7'b0000000)
        begin // SRLI
            if (rd_wb!=0) begin
                registers[rd_wb]<= reg_m;
                result<=reg_m;
            end
            else
                result<=0;
        end
    else if (funct7_wb == 7'b0100000)
        begin // SRAI
            if (rd_wb!=0) begin
                registers[rd_wb]<= reg_m;
                result<=reg_m;
            end
            else
                result<=0;
        end
    end
    default: result <= 0;
endcase
end
7'b0000011: begin//Load-uri
    case (funct3_m)
        3'b000: begin // LB
            registers[rd_m] <= reg_m;
            result <= reg_m;
        end
        3'b001: begin // LH

```

```

        registers[rd_m] <= reg_m;
        result <= reg_m;
    end
3'b010: begin // LW
    registers[rd_m] <= reg_m;
    result <= reg_m;
end
3'b011: begin // LD
    registers[rd_m] <= reg_m;
    result <= reg_m;
end
3'b100: begin // LBU
    registers[rd_m] <= reg_m;
    result <= reg_m;
end
3'b101: begin // LHU
    registers[rd_m] <= reg_m;
    result <= reg_m;
end
3'b110: begin // LWU
    registers[rd_m] <= reg_m;
    result <= reg_m;
end
endcase
end
7'b0100011: begin//Store-uri
    result<=reg_m;
end
7'b0110111: begin // LUI
    if (rd_wb!=0) begin
        registers[rd_wb]<= reg_m;
        result<=reg_m;
    end
    else
        result<=0;
end
7'b0010111: begin // AUIPC
    if (rd_wb!=0) begin
        registers[rd_wb]<= reg_m;
        result<=reg_m;
    end
    else
        result<=0;
end
7'b1101111: begin // JAL
    registers[rd_wb]<= reg_m;
    result<=jumpAddress_m;
end
7'b1100111: begin // JALR
    registers[rd_wb]<= reg_m;
    result<=jumpAddress_m;

```

```

    end
  7'b1100011: begin
    // Instructiuni branch
    case (funct3_wb)
      3'b000: begin // BEQ
        result<=reg_m;
      end
      3'b001: begin // BNE
        result<=reg_m;
      end
      3'b100: begin // BLT
        result<=reg_m;
      end
      3'b101: begin // BGE
        result<=reg_m;
      end
      3'b110: begin // BLTU
        result<=reg_m;
      end
      3'b111: begin // BGEU
        result<=reg_m;
      end
      default: result <= 0;
    endcase
  end
  default: result<= 0;
endcase
end
else
//Daca reset este activ, reinitializez valorile componentelor(mai putin ROM).
begin
  pc <= 64'h0;
  result<=0;
  Jump <= 0;
  jumpAddress_e<=0;
  jumpAddress_m<=0;
  bula<=0;
  for (i = 0; i <= 31; i = i + 1)
    registers[i] <= (i==2)?128:0;
  $readmemh("data.mem", dataMemory);
  instruction_f=32'hAAAAAAFF;
  instruction_d=32'hFFFFFFF;
  instruction_e<=0;
  instruction_m<=0;
  instruction_wb<=0;
  opcode_d<=0;
  funct3_d<=0;
  funct7_d<=0;
  imm12_d<=0;
  shamt_d<=0;
  rs1_d<=0;

```

```

rs2_d<=0;
rd_d<=0;
opcode_e<=0;
funct3_e<=0;
funct7_e<=0;
imm12_e<=0;
shamt_e<=0;
rs1_e<=0;
rs2_e<=0;
rd_e<=0;
opcode_m<=0;
funct3_m<=0;
funct7_m<=0;
imm12_m<=0;
shamt_m<=0;
rs1_m<=0;
rs2_m<=0;
rd_m<=0;
opcode_wb<=0;
funct3_wb<=0;
funct7_wb<=0;
rd_wb<=0;
end

end
endmodule

```

Revenire la secțiunea 3.1.2. Implementarea modulului RISC-VProcessor.

Anexa 4. Codul sursă complet al clasei CodificatorRISCV

Revenire la secțiunea 3.2. Implementarea codificatorului în varianta C#.

```

using System;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Linq;
using System.Windows.Forms;

namespace Codificator_RISC_V
{
    public partial class CodificatorRISCV : Form
    {
        private List<string> _loads, _stores, _jumps, _branches, _others;
        private Dictionary<string, string> _registerAlternateName;
        /// <summary>
        /// Functia apelata la apasarea butonului de salvare in fisier
        instructiune.
        /// </summary>
        /// <param name="sender"></param>
    }
}

```

```

    /// <param name="e"></param>
    private void buttonInstr_Click(object sender, EventArgs e)
    {
        saveFileDialog.Filter = "Memory File (*.mem)|*.mem|All Files (*.*)|*.*";
        saveFileDialog.FilterIndex = 1;
        saveFileDialog.FileName = "instruction_.mem";
        string continutInput = textBoxInput.Text;
        if (continutInput.Length == 0)
        {
            MessageBox.Show("Nu pot salva un fișier dacă nu au fost procesate
instrucțiuni!");
            return;
        }
        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            string caleFisier = saveFileDialog.FileName;

            try
            {
                string rezultat = "";
                //Se extrag octetii din instructiune in ordine inversa si se
afiseaza cate un octet pe linie,
                //pentru a putea fi cititi de functia readmemh din Verilog.
                List<string> instructions = continutInput.Split('\r').Select(x => x.Replace("\n", "")).Where(s => !
string.IsNullOrEmpty(s)).ToList();
                foreach (string instruction in instructions)
                {
                    string[] componente = instruction.Split(' ');
                    rezultat += string.Join(Environment.NewLine,
componente[0].Substring(6, 2), componente[0].Substring(4, 2),
componente[0].Substring(2, 2), componente[0].Substring(0, 2), "");
                }
                rezultat = rezultat.Substring(0, rezultat.Length - 2);
                File.WriteAllText(caleFisier, rezultat);

                MessageBox.Show("Instrucțiunile au fost salvate în format
mem!");
            }
            catch (Exception ex)
            {
                MessageBox.Show("A apărut o eroare la salvarea fișierului: "
+ ex.Message);
            }
        }
    }
    /// <summary>
    /// Functia apelata la apasarea butonului de incarcare a unor
instructiuni din fisier.
    /// </summary>

```

```

/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonIncFis_Click(object sender, EventArgs e)
{
    //Se deschide fisierul si se apeleaza functia de codificare pentru
    fiecare instructiune
    openFileDialog.Filter = "Assembly Files (*.s)|*.s";

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        string caleFisier = openFileDialog.FileName;

        try
        {
            using (StreamReader reader = new StreamReader(caleFisier))
            {
                while (!reader.EndOfStream)
                {
                    string linie = reader.ReadLine();
                    textBoxInstr.Text = linie.Trim(' ').Replace("\r\n",
                    ""));
                    this.buttonCod_Click(sender, e);
                }
            }

            MessageBox.Show("Fișier încărcat și interpretat cu succes!");
        }
        catch (Exception ex)
        {
            MessageBox.Show("A apărut o eroare la citirea și
interpretarea fișierului: " + ex.Message);
        }
    }
}

/// <summary>
/// Functia apelata la apasarea butonului de eliminare a instructiunilor
din dreapta.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonDel_Click(object sender, EventArgs e)
{
    //Se elimina continutul textBox-ului aferent
    textBoxInput.Text = "";
}

/// <summary>
/// Functia apelata la apasarea butonului de salvare in fisier input,
folosit la
/// explicarea codificarii instructiunii.
/// </summary>

```

```

    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void buttonSal_Click(object sender, EventArgs e)
    {
        saveFileDialog.Filter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";
        saveFileDialog.FilterIndex = 1;
        saveFileDialog.FileName = "input_.txt";
        if (textBoxInput.Text.Length == 0)
        {
            MessageBox.Show("Nu pot salva un fișier dacă nu au fost procesate
instrucțiuni!");
            return;
        }
        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            string caleFisier = saveFileDialog.FileName;

            try
            {
                //Instructiunile afisate in textBox corespund cu formatul
                //in care vreau sa afisez instructiunile in fisier.
                File.WriteAllText(caleFisier, textBoxInput.Text.Substring(0,
textBoxInput.Text.Length - 2));

                MessageBox.Show("Instrucțiunile au fost salvate în format
input!");
            }
            catch (Exception ex)
            {
                MessageBox.Show("A apărut o eroare la salvarea fișierului: "
+ ex.Message);
            }
        }
    }

    /// <summary>
    /// Functia de initializare a codificatorului, se initializeaza liste
de instructiuni si dictionarul de nume alternative pentru registri
    /// </summary>
    public CodificatorRISCV()
    {
        InitializeComponent();
        //Imi initializez liste de instructiuni si numele alternative
pentru registri.
        _loads = new List<string> { "lb", "lh", "lw", "ld", "lbu", "lhu",
"lwu" };
        _stores = new List<string> { "sb", "sh", "sw", "sd" };
        _jumps = new List<string> { "jal", "jalr" };
        _branches = new List<string> { "beq", "bne", "blt", "bge", "bltu",
"bgt" };
    }
}

```

```

"bgeu" };
    _others = new List<string> { "lui", "auipc" };
    _registerAlternateName = new Dictionary<string, string>();
    _registerAlternateName.Add("zero", "x0");
    _registerAlternateName.Add("ra", "x1");
    _registerAlternateName.Add("sp", "x2");
    _registerAlternateName.Add("gp", "x3");
    _registerAlternateName.Add("tp", "x4");
    _registerAlternateName.Add("t0", "x5");
    _registerAlternateName.Add("t1", "x6");
    _registerAlternateName.Add("t2", "x7");
    _registerAlternateName.Add("s0", "x8");
    _registerAlternateName.Add("fp", "x8");
    _registerAlternateName.Add("s1", "x9");
    _registerAlternateName.Add("a0", "x10");
    _registerAlternateName.Add("a1", "x11");
    _registerAlternateName.Add("a2", "x12");
    _registerAlternateName.Add("a3", "x13");
    _registerAlternateName.Add("a4", "x14");
    _registerAlternateName.Add("a5", "x15");
    _registerAlternateName.Add("a6", "x16");
    _registerAlternateName.Add("a7", "x17");
    _registerAlternateName.Add("s2", "x18");
    _registerAlternateName.Add("s3", "x19");
    _registerAlternateName.Add("s4", "x20");
    _registerAlternateName.Add("s5", "x21");
    _registerAlternateName.Add("s6", "x22");
    _registerAlternateName.Add("s7", "x23");
    _registerAlternateName.Add("s8", "x24");
    _registerAlternateName.Add("s9", "x25");
    _registerAlternateName.Add("s10", "x26");
    _registerAlternateName.Add("s11", "x27");
    _registerAlternateName.Add("t3", "x28");
    _registerAlternateName.Add("t4", "x29");
    _registerAlternateName.Add("t5", "x30");
    _registerAlternateName.Add("t6", "x31");
}
/// <summary>
/// Functie apelata la apasarea butonului de codificare a instructiunii.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void buttonCod_Click(object sender, EventArgs e)
{
    try
    {
        //Citirea instructiunii
        string instructiune = textBoxInstr.Text;
        //Impartirea instructiunii pe componente
        string[] componente = instructiune.Split(' ');
        //In functie de numarul de componente si de valoarea acestora, se

```

```

va calcula codificarea hexazecimala a instructiunii.
        //Daca apar erori in parsarea instructiunii, utilizatorul va fi
notificat printr-un MessageBox.
        if (componente.Length == 4)
        {
            string rd = componente[1].Replace(", ", "");
            string rs1 = componente[2].Replace(", ", "");
            if (!rd.StartsWith("x"))
            {
                _registerAlternateName.TryGetValue(rd, out rd);
                if (rd == null)
                {
                    MessageBox.Show($"Nu am găsit echivalent pentru rd!
{instructiune}");
                    return;
                }
            }
            if (!rs1.StartsWith("x"))
            {
                _registerAlternateName.TryGetValue(rs1, out rs1);
                if (rs1 == null)
                {
                    MessageBox.Show($"Nu am găsit echivalent pentru rs1!
{instructiune}");
                    return;
                }
            }
            int rdInt = int.Parse(rd.Substring(1));
            int rs1Int = int.Parse(rs1.Substring(1));
            if (rdInt > 32 || rs1Int > 32 || rs1Int < 0 || rdInt < 0)
            {
                MessageBox.Show($"Registrii în afara ariei de lucru!
{instructiune}");
                return;
            }
            if (componente[0].EndsWith("i"))
            {
                string imm = componente[3];
                int immInt = int.Parse(imm);
                if (immInt > 2047 || immInt < -2048)
                {
                    MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
                    return;
                }
                if ((immInt > 63 || immInt < 0) & (componente[0] ==
"slli" || componente[0] == "srli" || componente[0] == "srai"))
                {
                    MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
                    return;
                }
            }
        }
    }
}

```

```

        }
        int funct3, funct7;
        int opcode;
        switch (componente[0])
        {
            case "addi": opcode = 0b0010011; funct7 = 0; funct3 =
0; break;
            case "slli": opcode = 0b0010011; funct7 = 0; funct3 =
0b001; break;
            case "slti": opcode = 0b0010011; funct7 = 0; funct3 =
0b010; break;
            case "xori": opcode = 0b0010011; funct7 = 0; funct3 =
0b100; break;
            case "ori": opcode = 0b0010011; funct7 = 0; funct3 =
0b110; break;
            case "andi": opcode = 0b0010011; funct7 = 0; funct3 =
0b111; break;
            case "srli": opcode = 0b0010011; funct7 = 0; funct3 =
0b101; break;
            case "srai": opcode = 0b0010011; funct3 = 0b101;
            funct7 = 0b0100000; break;
            default: MessageBox.Show($"Operație nesuportată!
{instructiune}"); return;
        }
        int instrCod = opcode | (rdInt << 7) | (funct3 << 12) |
(rs1Int << 15) | (immInt << 20) | (funct7 << 25);
        textBoxInput.AppendText($"{instrCod.ToString("X8")}

{instructiune}" + Environment.NewLine);
    }
    else if (componente[0] == "sltiu")
    {
        string imm = componente[3];
        int immInt = int.Parse(imm);
        if (immInt > 2047 || immInt < -2048)
        {
            MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
            return;
        }
        int funct3 = 0b011, opcode = 0b0010011;
        int instrCod = opcode | (rdInt << 7) | (funct3 << 12) |
(rs1Int << 15) | (immInt << 20);
        textBoxInput.AppendText($"{instrCod.ToString("X8")}

{instructiune}" + Environment.NewLine);
    }
    else if (_branches.Contains(componente[0]))
    {
        string imm = componente[3];
        int immInt = int.Parse(imm);
        if (immInt > 2047 || immInt < -2048)
        {

```

```

        MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
        return;
    }
    int opcode = 0b1100011, funct3;
    switch (componente[0])
    {
        case "beq": funct3 = 0b000; break;
        case "bne": funct3 = 0b001; break;
        case "blt": funct3 = 0b100; break;
        case "bge": funct3 = 0b101; break;
        case "bltu": funct3 = 0b110; break;
        case "bgeu": funct3 = 0b111; break;
        default: MessageBox.Show($"Operație nesuportată!
{instructiune}"); return;
    }

    int instrCod = opcode | (funct3 << 12) | (rdInt << 15) |
(rs1Int << 20) | (((immInt >> 12) & 1) << 31) | (((immInt >> 5) & 0b11111) <<
25) | (((immInt >> 1) & 0b111) << 8) | (((immInt >> 11) & 1) << 7);
    textBoxInput.AppendText($"{instrCod.ToString("X8")}

{instructiune}" + Environment.NewLine);
}
else
{
    int funct3, funct7;
    string rs2 = componente[3].Replace(", ", "");
    if (!rs2.StartsWith("x"))
    {
        _registerAlternateName.TryGetValue(rs2, out rs2);
        if (rs2 == null)
        {
            MessageBox.Show($"Nu am găsit echivalent pentru
rs2! {instructiune}");
            return;
        }
    }
    int rs2Int = int.Parse(rs2.Substring(1));
    if (rs2Int < 0 || rs2Int > 31)
    {
        MessageBox.Show($"Registru rs2 înafara ariei de
lucru! {instructiune}");
        return;
    }
    int opcode = 0b0110011;
    switch (componente[0])
    {
        case "add": funct7 = 0; funct3 = 0; break;
        case "sub": funct7 = 0b0100000; funct3 = 0; break;
        case "sll": funct7 = 0; funct3 = 0b001; break;
        case "slt": funct7 = 0; funct3 = 0b010; break;
    }
}

```

```

        case "sltu": funct7 = 0; funct3 = 0b011; break;
        case "xor": funct7 = 0; funct3 = 0b100; break;
        case "or": funct7 = 0; funct3 = 0b110; break;
        case "and": funct7 = 0; funct3 = 0b111; break;
        case "srl": funct7 = 0; funct3 = 0b101; break;
        case "sra": funct3 = 0b101; funct7 = 0b0100000;
    break;
    default: MessageBox.Show($"Operatie nesuportata!
{instructiune}"); return;
}
int instrCod = opcode | (rdInt << 7) | (funct3 << 12) |
(rs1Int << 15) | (rs2Int << 20) | (funct7 << 25);
textBoxInput.AppendText($"{instrCod.ToString("X8")}
{instructiune}" + Environment.NewLine);
}
else if (componente.Length == 3)
{
    if (_others.Contains(componente[0]))
    {
        string rd = componente[1].Replace(",", "");
        if (!rd.StartsWith("x"))
        {
            _registerAlternateName.TryGetValue(rd, out rd);
            if (rd == null)
            {
                MessageBox.Show($"Nu am găsit echivalent pentru
rd! {instructiune}");
                return;
            }
        }
        int rdInt = int.Parse(rd.Substring(1));
        if (rdInt < 0 || rdInt > 31)
        {
            MessageBox.Show($"Registru rd înafara ariei de lucru!
{instructiune}");
            return;
        }
        string imm = componente[2];
        int immInt = int.Parse(imm);
        if (immInt > 1048575 || immInt < -1048576)
        {
            MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
            return;
        }
        int opcode;
        switch (componente[0])
        {
            case "lui": opcode = 0b0110111; break;
            case "auipc": opcode = 0b0010111; break;

```

```

                default: MessageBox.Show($"Operatie nesuportata!
{instructiune}"); return;
            }
            int instrCod = opcode | (rdInt << 7) | (immInt << 12);
            textBoxInput.AppendText($"{instrCod.ToString("X8")}

{instructiune}" + Environment.NewLine);
        }
        else if (_jumps.Contains(componente[0]))
        {//Jump-uri
            if (componente[0] == "jal")
            {
                int opcode = 0b1101111;
                string rd = componente[1].Replace(",", "");
                if (!rd.StartsWith("x"))
                {
                    _registerAlternateName.TryGetValue(rd, out rd);
                    if (rd == null)
                    {
                        MessageBox.Show($"Nu am găsit echivalent
pentru rd! {instructiune}");
                        return;
                    }
                }
                int rdInt = int.Parse(rd.Substring(1));
                if (rdInt < 0 || rdInt > 31)
                {
                    MessageBox.Show($"Registru rd înafara ariei de
lucru! {instructiune}");
                    return;
                }
                string imm = componente[2];
                int immInt = int.Parse(imm);
                if (immInt > 1048575 || immInt < -1048576)
                {
                    MessageBox.Show($"Valoare prea mare sau mică
pentru imm! {instructiune}");
                    return;
                }
                int instrCod = opcode | (rdInt << 7) | (((immInt >>
20) & 1) << 31) | (((immInt >> 1) & 0b111111111) << 21) | (((immInt >> 11) & 1)
<< 20) | (((immInt >> 12) & 0b11111111) << 12);
                textBoxInput.AppendText($"{instrCod.ToString("X8")}

{instructiune}" + Environment.NewLine);
            }
            else if (componente[0] == "jalr")
            {
                int opcode = 0b1100111;
                string rd = componente[1].Replace(",", "");
                if (!rd.StartsWith("x"))
                {
                    _registerAlternateName.TryGetValue(rd, out rd);

```

```

        if (rd == null)
        {
            MessageBox.Show($"Nu am găsit echivalent
pentru rd! {instructiune}");
            return;
        }
        int rdInt = int.Parse(rd.Substring(1));
        if (rdInt < 0 || rdInt > 31)
        {
            MessageBox.Show($"Registru rd înafara ariei de
lucru! {instructiune}");
            return;
        }
        string[] offsetRs1 = componente[2].Split('(');
        string rs1 = offsetRs1[1].Replace(")", "");
        if (!rs1.StartsWith("x"))
        {
            _registerAlternateName.TryGetValue(rs1, out rs1);
            if (rs1 == null)
            {
                MessageBox.Show($"Nu am găsit echivalent
pentru rs1! {instructiune}");
                return;
            }
            int rs1Int = int.Parse(rs1.Substring(1));
            if (rs1Int < 0 || rs1Int > 31)
            {
                MessageBox.Show($"Registru rs1 înafara ariei de
lucru! {instructiune}");
                return;
            }
            string imm = offsetRs1[0];
            int immInt = int.Parse(imm);
            if (immInt > 2047 || immInt < -2048)
            {
                MessageBox.Show($"Valoare prea mare sau mică
pentru imm! {instructiune}");
                return;
            }
            int instrCod = opcode | (rdInt << 7) | (rs1Int << 15)
| (immInt << 20);
            textBoxInput.AppendText($"{instrCod.ToString("X8")}
{instructiune}" + Environment.NewLine);
        }
    }
    else if (_loads.Contains(componente[0]))
    {//Load-uri
        int opcode = 0b0000011, funct3;
        string rd = componente[1].Replace(", ", "");
    }
}

```

```

        if (!rd.StartsWith("x"))
        {
            _registerAlternateName.TryGetValue(rd, out rd);
            if (rd == null)
            {
                MessageBox.Show($"Nu am găsit echivalent pentru
rd! {instructiune}");
                return;
            }
            int rdInt = int.Parse(rd.Substring(1));
            if (rdInt < 0 || rdInt > 31)
            {
                MessageBox.Show($"Registru rd înafara ariei de lucru!
{instructiune}");
                return;
            }
            string[] offsetRs1 = componente[2].Split('(');
            string rs1 = offsetRs1[1].Replace(")", "");
            if (!rs1.StartsWith("x"))
            {
                _registerAlternateName.TryGetValue(rs1, out rs1);
                if (rs1 == null)
                {
                    MessageBox.Show($"Nu am găsit echivalent pentru
rs1! {instructiune}");
                    return;
                }
                int rs1Int = int.Parse(rs1.Substring(1));
                if (rs1Int < 0 || rs1Int > 31)
                {
                    MessageBox.Show($"Registru rs1 înafara ariei de
lucru! {instructiune}");
                    return;
                }
                string imm = offsetRs1[0];
                int immInt = int.Parse(imm);
                if (immInt > 2047 || immInt < -2048)
                {
                    MessageBox.Show($"Valoare prea mare sau mică pentru
imm! {instructiune}");
                    return;
                }
                switch (componente[0])
                {
                    case "lb": funct3 = 0; break;
                    case "lh": funct3 = 0b001; break;
                    case "lw": funct3 = 0b010; break;
                    case "ld": funct3 = 0b011; break;
                    case "lbu": funct3 = 0b100; break;
                }
            }
        }
    }
}

```

```

        case "lhu": funct3 = 0b101; break;
        case "lwu": funct3 = 0b110; break;
        default: MessageBox.Show($"Operatie nesuportata!
{instructiune}"); return;
    }
    int instrCod = opcode | (rdInt << 7) | (rs1Int << 15) |
(immInt << 20) | (funct3 << 12);
    textBoxInput.AppendText($"{instrCod.ToString("X8")}")
{instructiune}" + Environment.NewLine);
}
else if (_stores.Contains(componente[0]))
{//Store-uri
    int opcode = 0b0100011, funct3;
    string rs2 = componente[1].Replace(", ", "");
    if (!rs2.StartsWith("x"))
    {
        _registerAlternateName.TryGetValue(rs2, out rs2);
        if (rs2 == null)
        {
            MessageBox.Show($"Nu am găsit echivalent pentru
rd! {instructiune}");
            return;
        }
    }
    int rs2Int = int.Parse(rs2.Substring(1));
    if (rs2Int < 0 || rs2Int > 31)
    {
        MessageBox.Show($"Registru rs2 înafara ariei de
lucru! {instructiune}");
        return;
    }
    string[] offsetRs1 = componente[2].Split('(');
    string rs1 = offsetRs1[1].Replace(")", "");
    if (!rs1.StartsWith("x"))
    {
        _registerAlternateName.TryGetValue(rs1, out rs1);
        if (rs1 == null)
        {
            MessageBox.Show($"Nu am găsit echivalent pentru
rs1! {instructiune}");
            return;
        }
    }
    int rs1Int = int.Parse(rs1.Substring(1));
    if (rs1Int < 0 || rs1Int > 31)
    {
        MessageBox.Show($"Registru rs1 înafara ariei de
lucru! {instructiune}");
        return;
    }
    string imm = offsetRs1[0];
}

```

[Revenire la secțiunea 3.2. Implementarea codificatorului în varianta C#.](#)

Anexa 5. Continutul fisierului main.py al codificatorului Python

```
import sys
import os
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QMessageBox, QFileDialog
from main_window_ui import Ui_Dialog
#Initializarea listelor de instructiuni si a numelor alternative pentru registri
```

```

loads = ["lb", "lh", "lw", "ld", "lbu", "lhu", "lwu"]
stores = ["sb", "sh", "sw", "sd"]
jumps = ["jal", "jalr"]
branches = ["beq", "bne", "blt", "bge", "bltu", "bgeu"]
others = ["lui", "auipc"]
registerAlternateName = {
    "zero": "x0",
    "ra": "x1",
    "sp": "x2",
    "gp": "x3",
    "tp": "x4",
    "t0": "x5",
    "t1": "x6",
    "t2": "x7",
    "s0": "x8",
    "fp": "x8",
    "s1": "x9",
    "a0": "x10",
    "a1": "x11",
    "a2": "x12",
    "a3": "x13",
    "a4": "x14",
    "a5": "x15",
    "a6": "x16",
    "a7": "x17",
    "s2": "x18",
    "s3": "x19",
    "s4": "x20",
    "s5": "x21",
    "s6": "x22",
    "s7": "x23",
    "s8": "x24",
    "s9": "x25",
    "s10": "x26",
    "s11": "x27",
    "t3": "x28",
    "t4": "x29",
    "t5": "x30",
    "t6": "x31",
}
#Functie care trunchiaza un numar la 32 de biti
def limit_32_bits(number):
    return number & 0xFFFFFFFF

#Clasa corespunzatoare ferestrei
class MainWindow(QtWidgets.QDialog):
    #Initializarea ferestrei cu functiile callback
    def __init__(self):
        super(MainWindow, self).__init__()
        self.ui = Ui_Dialog()

```

```

        self.ui.setupUi(self)
        self.ui.buttonDel.clicked.connect(self.buttonDel_clicked)
        self.ui.buttonSal.clicked.connect(self.buttonSal_clicked)
        self.ui.buttonIncFis.clicked.connect(self.buttonIncFis_clicked)
        self.ui.buttonCod.clicked.connect(self.buttonCod_clicked)
        self.ui.buttonInstr.clicked.connect(self.buttonInstr_clicked)

    #Functie care face validarea unui regisztr; va verifica daca regisztrul dat ca
parametru este valid.
    def valideare_regisztr(self, rd, nume_reg):
        if rd[0] != "x":
            try:
                rd = registerAlternateName[rd]
            except KeyError:
                QMessageBox.warning(self,"Avertisment",f"Nu am găsit echivalent
pentru {nume_reg}! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)
                return 100
            rd = int(rd[1:])
            if rd > 32 or rd < 0:
                QMessageBox.warning(self,"Avertisment",f"Regisztrul {nume_reg} in afara
ariei de lucru! {self.ui.textInstr.toPlainText()}",QMessageBox.Ok,)
                return 100
            return rd
        #Functia de eliminare a continutului textBox-ului cu instructiuni din dreapta
        def buttonDel_clicked(self):
            self.ui.listInput.clear()
        #Functia de incarcare a unui fisier cu instructiuni in asamblare
        def buttonIncFis_clicked(self):
            file_path, _ = QFileDialog.getOpenFileName(self, "Citire fișier
asamblare", "", "Assembly Files (*.s)")
            if file_path:
                with open(file_path, "r") as file:
                    for line in file:
                        self.ui.textInstr.setText(line.strip())
                        self.buttonCod_clicked()
                QMessageBox.information(self, "Parcursare completă", "Fișierul a fost
parcurs!", QMessageBox.Ok)
        #Functia de salvare a instructiunilor din textBox in format explicativ
        def buttonSal_clicked(self):
            file_path, _ = QFileDialog.getSaveFileName(self, "Salvare în fișier
input", "input_.txt", "Text Files (*.txt)")
            if file_path:
                with open(file_path, "w") as file:
                    for index in range(self.ui.listInput.count()):
                        item = self.ui.listInput.item(index)
                        file.write(f"{item.text()}\n")
                QMessageBox.information(self, "Salvare completă", "Instructiunile
au fost salvate în format input!", QMessageBox.Ok,)
        #Functia de salvare a instructiunilor in format executabil pe procesor
        def buttonInstr_clicked(self):
            file_path, _ = QFileDialog.getSaveFileName(self, "Salvare în fișier
instruction","instruction_.mem","Memory Files (*.mem"),)

```

```

if file_path:
    with open(file_path, "w") as file:
        for index in range(self.ui.listInput.count()):
            instruction = self.ui.listInput.item(index).text()
            componente = instruction.split(" ")
            rezultat = (componente[0][6:8] + os.linesep + componente[0]
[4:6] + os.linesep + componente[0][2:4] + os.linesep + componente[0][0:2] +
os.linesep)
                file.write(f"{rezultat}")
                QMessageBox.information(self, "Salvare completă", "Instrucțiunile
au fost salvate în format mem!", QMessageBox.Ok, )
#Functia de codificare a unei instructiuni
def buttonCod_clicked(self):
    try:
        #Impartirea instructiunii pe componente
        componente = self.ui.textInstr.toPlainText().split(" ")
        #In functie de numarul de componente si valoarea acestora, se
        formeaza instructiunea codificata in format hexazecimal
        #In cazul in care apar probleme pe parcursul parsarii instructiunii,
        utilizatorul va fi notificat printr-un MessageBox
        if len(componente) == 4:
            rd = componente[1].strip(",")
            rs1 = componente[2].strip(",")
            rd = self.validare_registru(rd, "rd")
            if rd == 100:
                return
            rs1 = self.validare_registru(rs1, "rs1")
            if rs1 == 100:
                return
            if componente[0][-1] == "i" or componente[0] == "sliu":
                imm = int(componente[3])
                if imm > 2047 or imm < -2048:
                    QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică sau mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok, )
                    return
                if (imm > 63 or imm < 0) and (componente[0] == "slli" or
componente[0] == "srli" or componente[0] == "srai"):
                    QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică sau mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok, )
                    return
                if componente[0] == "addi":
                    opcode = 0b0010011
                    funct7 = 0
                    funct3 = 0
                elif componente[0] == "slli":
                    opcode = 0b0010011
                    funct7 = 0
                    funct3 = 0b001
                elif componente[0] == "slii":
                    opcode = 0b0010011
                    funct7 = 0
                    funct3 = 0

```

```

        funct3 = 0b010
    elif componente[0] == "xori":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b100
    elif componente[0] == "ori":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b110
    elif componente[0] == "andi":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b111
    elif componente[0] == "srli":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b101
    elif componente[0] == "srai":
        opcode = 0b0010011
        funct7 = 0b0100000
        funct3 = 0b101
    elif componente[0] == "sltiu":
        opcode = 0b0010011
        funct7 = 0
        funct3 = 0b011
    else:
        QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
        return
    instrCod = limit_32_bits(opcode | (rd << 7) | (funct3 << 12)
| (rs1 << 15) | (imm << 20) | (funct7 << 25))
    self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
    elif componente[0] in branches:
        opcode = 0b1100011
        imm = int(componente[3])
        if imm > 2047 or imm < -2048:
            QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică sau mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        if componente[0] == "beq":
            funct3 = 0b000
        elif componente[0] == "bne":
            funct3 = 0b001
        elif componente[0] == "blt":
            funct3 = 0b100
        elif componente[0] == "bge":
            funct3 = 0b101
        elif componente[0] == "bltu":
            funct3 = 0b110
        elif componente[0] == "bgeu":

```

```

        funct3 = 0b111
    else:
        QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
    instrCod = limit_32_bits(opcode | (funct3 << 12) | (rd << 15)
| (rs1 << 20) | (((imm >> 12) & 1) << 31) | (((imm >> 5) & 0b111111) << 25) |
(((imm >> 1) & 0b1111) << 8) | (((imm >> 11) & 1) << 7))
            self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
    else:
        rs2 = componente[3].strip(",")
        rs2 = self.validare_registru(rs2, "rs2")
        if rs2 == 100:
            return
        opcode = 0b0110011
        if componente[0] == "add":
            funct3 = 0
            funct7 = 0
        elif componente[0] == "sub":
            funct7 = 0b0100000
            funct3 = 0
        elif componente[0] == "sll":
            funct7 = 0
            funct3 = 0b001
        elif componente[0] == "slt":
            funct7 = 0
            funct3 = 0b010
        elif componente[0] == "sltu":
            funct7 = 0
            funct3 = 0b011
        elif componente[0] == "xor":
            funct7 = 0
            funct3 = 0b100
        elif componente[0] == "or":
            funct7 = 0
            funct3 = 0b110
        elif componente[0] == "and":
            funct7 = 0
            funct3 = 0b111
        elif componente[0] == "srl":
            funct7 = 0
            funct3 = 0b101
        elif componente[0] == "sra":
            funct7 = 0b0100000
            funct3 = 0b101
        else:
            QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
    instrCod = limit_32_bits(opcode | (rd << 7) | (funct3 << 12)

```

```

| (rs1 << 15) | (rs2 << 20) | (funct7 << 25))
    self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
    elif len(componente) == 3:
        if componente[0] in others:
            rd = componente[1].strip(",")
            rd = self.validare_registru(rd, "rd")
            if rd == 100:
                return
            imm = int(componente[2])
            if imm > 1048575 or imm < -1048576:
                QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
                return

        if componente[0] == "lui":
            opcode = 0b0110111
        elif componente[0] == "auipc":
            opcode = 0b0010111
        else:
            QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        instrCod = limit_32_bits(opcode | (rd << 7) | (imm << 12))
        self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
        elif componente[0] in jumps:
            if componente[0] == "jal":
                opcode = 0b1101111
                rd = componente[1].strip(",")
                rd = self.validare_registru(rd, "rd")
                if rd == 100:
                    return
                imm = int(componente[2])
                if imm > 1048575 or imm < -1048576:
                    QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
                    return
                instrCod = limit_32_bits(opcode | (rd << 7) | (((imm >>
20) & 1) << 31) | (((imm >> 1) & 0b1111111111) << 21) | (((imm >> 11) & 1) << 20)
| ((imm >> 12) & 0b11111111) << 12))
                self.ui.listInput.addItem("{:08X}".format(instrCod) + "
" + self.ui.textInstr.toPlainText())
            elif componente[0] == "jalr":
                opcode = 0b1100111
                rd = componente[1].strip(",")
                rd = self.validare_registru(rd, "rd")
                if rd == 100:
                    return
                offsetRs1 = componente[2].split("(")
                rs1 = offsetRs1[1].strip(")")

```

```

        rs1 = self.validare_registru(rs1, "rs1")
        if rs1 == 100:
            return
        imm = int(offsetRs1[0])
        if imm > 2047 or imm < -2048:
            QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        instrCod = limit_32_bits(opcode | (rd << 7) | (rs1 << 15)
| (imm << 20))
        self.ui.listInput.addItem("{:08X}".format(instrCod) + "
" + self.ui.textInstr.toPlainText())
        elif componente[0] in loads:
            opcode = 0b00000011
            rd = componente[1].strip(",")
            rd = self.validare_registru(rd, "rd")
            if rd == 100:
                return
            offsetRs1 = componente[2].split("(")
            rs1 = offsetRs1[1].strip(")")
            rs1 = self.validare_registru(rs1, "rs1")
            if rs1 == 100:
                return
            imm = int(offsetRs1[0])
            if imm > 2047 or imm < -2048:
                QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
                return
            if componente[0] == "lb":
                funct3 = 0
            elif componente[0] == "lh":
                funct3 = 0b001
            elif componente[0] == "lw":
                funct3 = 0b010
            elif componente[0] == "ld":
                funct3 = 0b011
            elif componente[0] == "lbu":
                funct3 = 0b100
            elif componente[0] == "lhu":
                funct3 = 0b101
            elif componente[0] == "lwu":
                funct3 = 0b110
            else:
                QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
                return
            instrCod = limit_32_bits(opcode | (rd << 7) | (rs1 << 15) |
(immm << 20) | (funct3 << 12))
            self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
        elif componente[0] in stores:

```

```

        opcode = 0b0100011
        rs2 = componente[1].strip(",")
        rs2 = self.validare_registru(rs2, "rs2")
        if rs2 == 100:
            return
        offsetRs1 = componente[2].split("(")
        rs1 = offsetRs1[1].strip(")")
        rs1 = self.validare_registru(rs1, "rs1")
        if rs1 == 100:
            return
        imm = int(offsetRs1[0])
        if imm > 2047 or imm < -2048:
            QMessageBox.warning(self, "Avertisment", f"Valoare prea
mică/mare pentru imm! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        if componente[0] == "sb":
            funct3 = 0
        elif componente[0] == "sh":
            funct3 = 0b001
        elif componente[0] == "sw":
            funct3 = 0b010
        elif componente[0] == "sd":
            funct3 = 0b011
        else:
            QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        instrCod = limit_32_bits(opcode | (rs2 << 20) | (rs1 << 15) |
(funct3 << 12) | ((imm & 0b11111) << 7) | (((imm >> 5) & 0b1111111) << 25))
        self.ui.listInput.addItem("{:08X}".format(instrCod) + " " +
self.ui.textInstr.toPlainText())
        else:
            QMessageBox.warning(self, "Avertisment", f"Operatie
nesuportata! {self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
        else:
            QMessageBox.warning(self, "Avertisment", f"Operatie nesuportata!
{self.ui.textInstr.toPlainText()}", QMessageBox.Ok,)
            return
    except Exception as e:
        QMessageBox.warning(self, "Avertisment", f"A apărut o eroare la
tratarea instrucțiunii: {self.ui.textInstr.toPlainText()}
{str(e)}", QMessageBox.Ok,)
        return

#Functia principală, în care se instantiază clasa MainWindow.
def main():
    app = QtWidgets.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())

```

```
if __name__ == "__main__":
    main()
```

Revenire la secțiunea 3.3. Implementarea codificatorului în varianta Python.

Anexa 6. Codul sursă al emulatorului pentru Raspberry Pi Pico

```
import time
import machine
#Functie de extindere a semnului unui numar la 64 de biti
def sign_extend(number, bits):
    mask = (1 << bits) - 1

    sign_bit = number & (1 << (bits - 1))

    if sign_bit:
        return number | (~mask & 0xFFFFFFFFFFFFFF)
    else:
        return number

#Functie de trunchiere a unui numar la 64 de biti
def limit64bits(number):
    return number&0xFFFFFFFFFFFFFF

#Functie de trunchiere a unui numar la 16 de biti
def limit16bits(number):
    return number&0xFFFF

#Functia de control a LED-urilor, in functie de rezultatul unei instructiuni
def set_pins_from_result(result):
    gpio_pins = [machine.Pin(pin, machine.Pin.OUT) for pin in range(16)]

    for i in range(16):
        if result & (1 << i):
            gpio_pins[i].high()
        else:
            gpio_pins[i].low()

#Clasa principală
class RISCVProcessor:
    #Constructorul clasei, se initializeaza toate componentele procesorului si
    #memoriile
    def __init__(self):
        self.pc = 0
        self.regdif = 0
        self.Jump = False
        self.jumpAddress = 0
        self.registers = [0] * 32
        self.dataMemory = [0] * 100
        self.memory = [0] * 176
        self.instruction = 0
        self.opcode = 0
        self.funct3 = 0
        self.funct7 = 0
```

```

        self.imm12 = 0
        self.shamt = 0
        self.rs1 = 0
        self.rs2 = 0
        self.rd = 0
        self.result = 0
    #Initializare memorie ROM
    with open("instructions.txt", "r") as f:
        hex_instructions = f.read().split()
        self.memory= hex_instructions
    #with open("data.mem", "r") as f:
    #    hex_data = f.read().split()
    #    self.dataMemory[:len(hex_data)] = [int(x, 16) for x in hex_data]

    #Functia de preluare din memorie si de executare a unei instructiuni
    def execute(self):
        #Pentru a face o distinctie intre reinceperea programului, se afiseaza la
        #consola un spatiu, informatie folosita la debug
        if(self.pc==0):
            print()
        #Preluarea instructiunii din ROM, in functie de valoarea registrului PC
        instruction = (int(self.memory[self.pc+3],16)<<24)+

(int(self.memory[self.pc+2],16)<<16)+(int(self.memory[self.pc+1],16)<<8)+

(int(self.memory[self.pc+0],16))
        #Decodificarea instructiunii in componente
        self.opcode = instruction & 0b1111111
        self.funct3 = (instruction >> 12) & 0b111
        self.funct7 = (instruction >> 25) & 0b1111111
        self.imm12 = (instruction >> 20) & 0b111111111111
        self.shamt = (instruction >> 20) & 0b11111
        self.rs1 = (instruction >> 15) & 0b11111
        self.rs2 = (instruction >> 20) & 0b11111
        self.rd = (instruction >> 7) & 0b11111
        #Executarea propriu-zisa a instructiunii, in functie de valorile
        #decodificate
        if self.opcode == 0b0110011: # Instructiuni de tip R
            if self.funct3 == 0b000: # ADD, SUB
                if self.funct7 == 0b0000000: #ADD
                    self.registers[self.rd] =
limit64bits(self.registers[self.rs1] + self.registers[self.rs2])
                    self.result=limit16bits(self.registers[self.rd])
                    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
                elif self.funct7 == 0b0100000: #SUB
                    self.registers[self.rd] =
limit64bits(self.registers[self.rs1] - self.registers[self.rs2])
                    self.result=limit16bits(self.registers[self.rd])
                    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
                elif self.funct3 == 0b001: # SLL
                    self.registers[self.rd] = limit64bits(self.registers[self.rs1] <<
(self.registers[self.rs2]&0b11111))
                    self.result=limit16bits(self.registers[self.rd])

```

```

        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b010: # SLT
        self.regdif = limit64bits(self.registers[self.rs1] -
self.registers[self.rs2])
        self.registers[self.rd] = int(self.regdif >> 63)
        self.result=limit16bits(self.registers[self.rd])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b011: # SLTU
        self.registers[self.rd] = 1 if self.registers[self.rs1] <
self.registers[self.rs2] else 0
        self.result=limit16bits(self.registers[self.rd])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b100: # XOR
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] ^
self.registers[self.rs2])
        self.result=limit16bits(self.registers[self.rd])
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b101:
        if self.funct7 == 0b0000000: # SRL
            self.registers[self.rd] =
limit64bits(self.registers[self.rs1] >> (self.registers[self.rs2]&0b11111))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct7 == 0b0100000: # SRA
            self.registers[self.rd] =
limit64bits(self.registers[self.rs1] >> (self.registers[self.rs2]&0b11111))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b110: # OR
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] |
self.registers[self.rs2])
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b111: # AND
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] &
self.registers[self.rs2])
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        else:
            self.result=0
    elif self.opcode == 0b0010011: # Instructiuni de tip I
        self.funct3 = self.funct3
        if self.funct3 == 0b000: # ADDI
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] +
sign_extend(self.imm12,12))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b001: # SLLI
            self.registers[self.rd] = limit64bits(self.registers[self.rs1] <<
self.shamt)
            self.result=limit16bits(self.registers[self.rd])

```

```

        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b010: # SLTI
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] -
sign_extend(self.imm12,12))>>63
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b011: # SLTIU
        self.registers[self.rd] = self.registers[self.rs1] <
sign_extend(self.imm12,12)
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b100: # XORI
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] ^
sign_extend(self.imm12,12))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b110: # ORI
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] |
sign_extend(self.imm12,12))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b111: # ANDI
        self.registers[self.rd] = limit64bits(self.registers[self.rs1] &
sign_extend(self.imm12,12))
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b101:
        if self.funct7 == 0b0000000: #SRLI
            self.registers[self.rd] =
limit64bits(self.registers[self.rs1] >> self.shamt)
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct7 == 0b0100000: #SRAI
            self.registers[self.rd] =
limit64bits(self.registers[self.rs1] >> self.shamt)
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        else:
            self.result=0
    elif self.opcode == 0b0000011: # Instrucţiuni Load
        if self.funct3 == 0b000: # LB
            self.registers[self.rd]=sign_extend(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xff,8)
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b001: # LH
            self.registers[self.rd]=sign_extend(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xFFFF,16)
                self.result=limit16bits(self.registers[self.rd])

```

```

        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b010: #LW

        self.registers[self.rd]=sign_extend(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xFFFFFFFF,32)
            self.result=limit16bits(self.registers[self.rd])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b011: #LD
            self.registers[self.rd]=sign_extend(limit64bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))])),64)
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b100: #LBU

            self.registers[self.rd]=self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xFF
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b101: #LHU

            self.registers[self.rd]=self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xFFFF
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b110: #LWU

            self.registers[self.rd]=self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]&0xFFFFFFFF
                self.result=limit16bits(self.registers[self.rd])
                print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        else:
            self.result=0
    elif self.opcode == 0b0100011: # Instructiuni Store; Pentru aceste
instructiuni, imm12 se calculeaza intr-un mod diferit
        if self.funct3== 0b000: # SB
            self.imm12=((instruction>>25)<<7)|((instruction>>7)&0b11111)
self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]=self.registers[self.rs2]&0xff

        self.result=limit16bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
        elif self.funct3 == 0b001: #SH
            self.imm12=((instruction>>25)<<7)|((instruction>>7)&0b11111)
self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]=self.registers[self.rs2]&0xFFFF

        self.result=limit16bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))])
            print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b010: #SW

```

```

        self.imm12=((instruction>>25)<<7)|((instruction>>7)&0b11111)
self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]
= self.registers[self.rs2]&0xFFFFFFFF

self.result=limit16bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign
_extend(self.imm12,12))])
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
elif self.funct3 == 0b011: #SD
    self.imm12=((instruction>>25)<<7)|((instruction>>7)&0b11111)
self.dataMemory[limit64bits(self.registers[self.rs1]+sign_extend(self.imm12,12))]
= self.registers[self.rs2]

self.result=limit16bits(self.dataMemory[limit64bits(self.registers[self.rs1]+sign
_extend(self.imm12,12))])
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
else:
    self.result=0
elif self.opcode == 0b0110111: # LUI
    self.imm12=instruction>>12
    self.registers[self.rd] = limit64bits(sign_extend(self.imm12,20) <<
12)
    self.result=limit16bits(self.registers[self.rd])
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
elif self.opcode == 0b0010111: # AUIPC
    self.imm12=instruction>>12
    self.registers[self.rd] = limit64bits(self.pc+
(sign_extend(self.imm12,20)<< 12))
    self.result=limit16bits(self.registers[self.rd])
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
elif self.opcode == 0b1101111: # JAL
    self.registers[self.rd] = limit64bits(self.pc+4)
    self.Jump=True
    self.imm12=(((instruction>>31)<<19)|(((instruction>>12)&0xFF)<<11)|(
instruction>>21)&0x3FF)<<1
    self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,20))
    self.result=limit16bits(self.jumpAddress)
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
elif self.opcode == 0b1100111: # JALR
    self.registers[self.rd] = self.pc + 4
    self.Jump = True
    self.jumpAddress = limit64bits(self.registers[self.rs1] +
sign_extend(self.imm12//2*2,12))
    self.result = self.jumpAddress
    self.result=limit16bits(self.jumpAddress)
    print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
elif self.opcode == 0b1100011:#Instructiuni branch; pentru aceste tipuri
de instructiuni, imm12 se calculeaza intr-un mod diferit
    self.imm12=((instruction>>31)<<12)|(((instruction>>7)&1)<<11)|(
((instruction>>25)&0b11111)<<5)|((instruction>>8)&0xf)<<1
    if self.funct3 == 0b000: #BEQ
        self.Jump=(self.registers[self.rs1]==self.registers[self.rs2])

```

```

        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b001: #BNE
        self.Jump=(self.registers[self.rs1]!=self.registers[self.rs2])
        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b100: #BLT
        self.Jump=(self.registers[self.rs1]-self.registers[self.rs2])>>63
        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b101: #BGE
        self.Jump=1-((self.registers[self.rs1]-
self.registers[self.rs2]))>>63
        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b110: #BLTU
        self.Jump=(self.registers[self.rs1]<self.registers[self.rs2])
        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    elif self.funct3 == 0b111: #BGEU
        self.Jump=(self.registers[self.rs1]>=self.registers[self.rs2])
        self.jumpAddress=limit64bits(self.pc+sign_extend(self.imm12,12))
        self.result=limit16bits(self.jumpAddress if self.Jump else 0)
        print(f"PC: {hex(self.pc)}, result: {hex(self.result)}")
    else:
        self.result=0
else:
    self.result=0
#Verificarea flag-ului de salt
if self.Jump:
    self.pc = self.jumpAddress
    self.Jump = False
else:
    self.pc += 4
#Afisarea pe LED-uri a rezultatului
set_pins_from_result(self.result)
#Functia de executare a programului din memoria ROM
def run(self):
    while self.pc<len(self.memory):
        self.execute()
        time.sleep(2)
#Instantierea clasei si executarea functiei run
processor = RISCVProcessor()
processor.run()

```

Revenire la secțiunea 3.4. Implementarea emulatorului pe Raspberry Pi Pico.