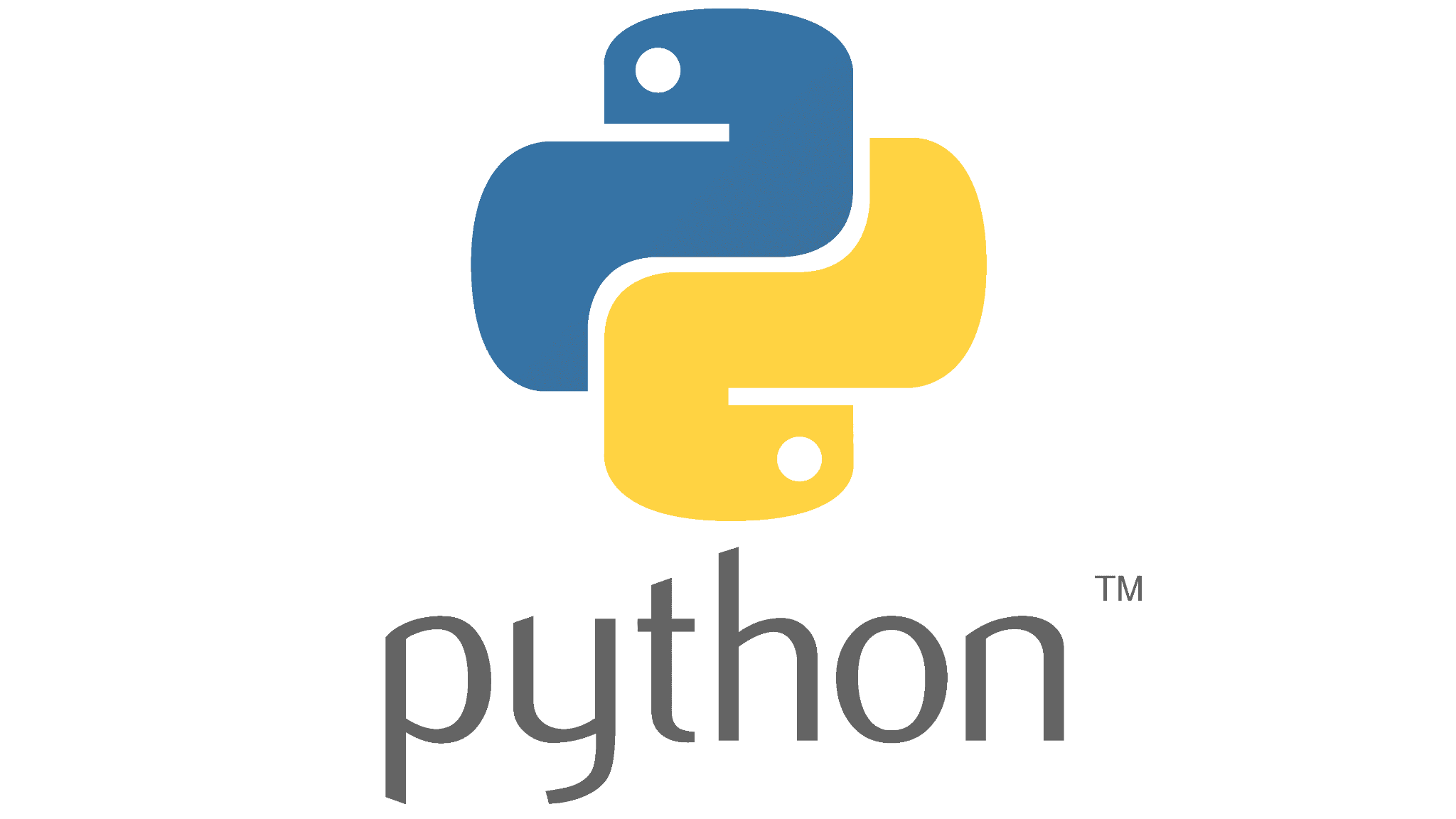
**Simulator RISC-V**

IN



Realizat de:

Dancău Rareș-Andrei, 1410A

Cojocaru Rareș, 1410A

# Cuprins

[Cuprins 2](#__RefHeading___Toc424_4041092833)

[Enunțarea temei 3](#__RefHeading___Toc438_4041092833)

[Arhitectura generală 3](#__RefHeading___Toc436_4041092833)

[Funcționalitatea 4](#__RefHeading___Toc434_4041092833)

[Rolul fiecărui membru 5](#__RefHeading___Toc432_4041092833)

[Complexitate și corectitudine 5](#__RefHeading___Toc430_4041092833)

[Demonstrația corectitudinii pentru o secvență 5](#__RefHeading___Toc440_4041092833)

[Complexitate 6](#__RefHeading___Toc444_4041092833)

[Testarea programului 6](#__RefHeading___Toc428_4041092833)

[Complexitatea cazurilor de test 6](#__RefHeading___Toc442_4041092833)

[Anexa 1. Explicații ale programelor de test 8](#__RefHeading___Toc426_4041092833)

# Enunțarea temei

În cadrul acestui proiect ne-am propus implementarea unui simulator în Python a unui procesor RISC-V ce poate executa un număr considerabil de instrucțiuni specifice arhitecturii. Folosind acest simulator, utilizatorii vor putea experimenta într-un mod virtual lucrul cu regiștrii și memoriile RAM și ROM ale unui procesor RISC-V. În continuare va fi descrisă arhitectura proiectului, funcționalitatea, testarea, complexitatea și corectitudinea.

# Arhitectura generală

Proiectul este compus din mai multe fișiere, printre care un script Python principal și fișiere aferente programelor de test. Mai jos poate fi identificată o descriere a fiecărei componente ale programului.

* RISC-V.py
  + Este script-ul principal, și se ocupă de simularea efectivă a procesorului. Procesorul este descris printr-o clasă ce conține următoarele elemente:
    - pc, ce are ca rol localizarea în cadrul memoriei ROM.
    - regdif este un registru folosit în cadrul unor instrucțiuni pentru a stoca diferențe dintre doi regiștri.
    - Jump este un flag care simbolizează dacă instrucțiunea curentă este una ce produce un salt(dacă modifică valoarea lui pc).
    - jumpAddress va stoca adresa la care se va efectua saltul.
    - registers reprezintă bancul de regiștrii și conține 32 de regiștri.
    - dataMemory este memoria RAM și conține 100 de celule de memorie.
    - memory reprezintă memoria ROM, în care vor fi stocate instrucțiunile ce vor fi executate de către procesor.
    - instruction ce reprezintă un șir de 32 biți ce va fi citit din memoria ROM. Pentru fiecare instrucțiune, se vor calcula opcode, funct3, funct7, imm12, shamt, rs1, rs2 și rd ce vor fi folosite în funcție de tipul instrucțiunii.
    - result este un registru pe 16 biți ce va conține rezultatul executării instrucțiunii curente. Are această dimensiune pentru a putea fi reprezentat pe 16 LED-uri.
    - Funcția initializare\_instructiuni se ocupă de inițializarea memoriei ROM cu octeții preluați din fișierul de instrucțiuni trimis ca parametru.
    - Funcția reset simulează o apăsare a butonului de reset a procesorului. Reinițializează toate variabilele.
    - Funcția execute se ocupă de preluarea din memoria ROM a instrucțiunilor, decodarea acestora și execuția efectivă.
    - Funcția run primește o listă de rezultate așteptate pentru a testa execuția procesorului.
  + Funcția main are rolul de a inițializa procesorul și de a executa câteva programe de test.
* instructions\_add\_load.mem
  + Este primul caz de test, și conține un program format din: 3 instrucțiuni addi, o instrucțiune add, o instrucțiune sw de stocare în memorie și o instrucțiune lw de extragere din memorie.
* instructions\_counter.mem
  + Conține un program format din patru instrucțiuni: două addi, un sub și un bne. Simulează un counter de la 1 la 10.
* instructions\_mare.mem
  + Este un program ce conține un număr mare instrucțiuni. Scopul lui este să parcurgă un număr cât mai mare de instrucțiuni pentru a putea testa ulterior execuția acestora.
* instructions\_paritate.mem
  + Este un program care în funcție de paritatea valorii din registrul a0, plasează în registrul t0 valoarea 0x3e7 pentru impar și 0x378 pentru număr par.
* instructions\_sll\_xor.mem
  + Este un program care execută diverse operații de shiftare și lucru pe biți.
* instructions\_stocari\_extrageri.mem
  + Se testează buna funcționare a operațiilor de stocare și extragere din memoria RAM cu numere negative.
* instructions\_subrutina.mem
  + Este un program care ilustrează modul de lucru cu funcții.
* instructions\_suma.mem
  + Este un program ce efectuează suma numerelor naturale de la 1 la 4 inclusiv, valoarea acestei sume fiind stocată în registrul t1.

# Funcționalitatea

Simulatorul reproduce modul în care un procesor ar interpreta și executa instrucțiuni RISC-V, demonstrând funcționarea acestuia într-un mediu controlat și simulat. În timpul execuției programului se produc următoarele operații :

* **Interpretarea instrucțiunilor:** procesorul identifică tipul fiecărei instrucțiuni, precum și operația specifică ce trebuie efectuată (adunare, scădere, etc.).
* **Executarea operațiilor simple:** după interpretare, procesorul efectuează operațiile implicate în instrucțiunea respectivă. Acestea sunt operații fundamentale, cum ar fi adunarea, scăderea, operațiile logice (ȘI, SAU, XOR), shiftarea sau manipularea datelor din memorie.
* **Actualizarea stării procesorului:** odată ce operația este efectuată, starea internă a procesorului se actualizează. Aceasta include modificarea valorilor din registrele procesorului sau din memoria acestuia, reflectând schimbările rezultate în urma execuției instrucțiunii.
* **Verificarea rezultatelor:** după ce procesorul execută un set de instrucțiuni, acesta verifică rezultatele obținute cu un set de rezultate așteptate.

# Rolul fiecărui membru

Cojocaru Rareș- Demonstrația corectitudinii, descrierea complexității, descrierea funcționalității.  
Dancău Rareș-Andrei- Implementarea simulatorului, realizarea programelor de test și scrierea celorlalte capitole din documentație.

# Complexitate și corectitudine

## Demonstrația corectitudinii pentru o secvență



P0

C1: if self.funct3 == 0b000: # ADD, SUB

C2: if self.funct7 == 0b0000000:

A1: self.registers[self.rd] = limit64bits(self.registers[self.rs1] + self.registers[self.rs2])

A2: self.result = hex(limit16bits(self.registers[self.rd]))

elif self.funct7 == 0b0100000:

A3: self.registers[self.rd] = limit64bits(self.registers[self.rs1] - self.registers[self.rs2])

A4: self.result = hex(limit16bits(self.registers[self.rd]))

C3: elif self.funct3 == 0b001: # SLL

A5: self.registers[self.rd] = limit64bits(self.registers[self.rs1] << (self.registers[self.rs2]&0b11111))

A6: self.result = hex(limit16bits(self.registers[self.rd]))

P: Instructiunea este de tip R ( Register )

Funct contine bitii specifici unei operatii de tipul ADD/SUB/SLL

Q: Registrul rd va contine rezultatul operatiei ADD/SUB/SLL intre registrele rs1 si rs2.

{C1,P0} = { if self.funct3 == 0b000: }

{C2, P0} = { if self.funct7 == 0b0000000: }

P1 = { self.registers[self.rd] = limit64bits(self.registers[self.rs1] + self.registers[self.rs2]) }

P2 = { self.result = hex(limit16bits(self.registers[self.rd])) }

{~C2, P0} = { elif self.funct7 == 0b0100000: }

P3 = { self.registers[self.rd] = limit64bits(self.registers[self.rs1] - self.registers[self.rs2]) }

P4 = { self.result = hex(limit16bits(self.registers[self.rd])) }

{~C1, P0} = { elif self.funct3 == 0b001: }

P5 = { self.registers[self.rd] = limit64bits(self.registers[self.rs1] << (self.registers[self.rs2]&0b11111)) }

P6 = { self.result = hex(limit16bits(self.registers[self.rd])) }

Demonstratie:

C1 & (C2 & P0) -> A1 & A2 => rd = rs1 + rs2 deci P -> Q

C1 & (~C2 & P0) -> A3 & A4 => rd = rs1 - rs2 deci P -> Q

~C1 & (C3 & P0) -> A5 & A6 => rd = rs1 << rs2 deci P -> Q

## Complexitate

În absența structurilor de tip branch și jump, performanța și complexitatea de execuție a proiectului sunt determinate de lungimea programului și de tipurile de instrucțiuni prezente. Cu fiecare instrucțiune executată secvențial, complexitatea temporală este direct proporțională cu numărul total de instrucțiuni din program. Astfel, timpul de execuție este influențat în principal de lungimea și tipurile de instrucțiuni, fără a avea ramificații sau salturi care să afecteze secvența liniară de execuție.

# Testarea programului

Simulatorul de procesor este testat prin executarea mai multor programe descrise anterior. Se folosește funcția run din clasa RISCVProcessor pentru a verifica rezultatul execuției procesorului pentru un set de instrucțiuni cu un set calculat anterior. Dacă cele două șiruri de valori sunt egale, atunci execuția procesorului a fost cu succes și se afișează un mesaj corespunzător, altfel se afișează cele două liste de valori ale execuției actuale și cea determinată anterior și se poate vedea la ce instrucțiuni rezultatul nu este cel așteptat.

## Complexitatea cazurilor de test

Pentru programele ce conțin instrucțiuni de tip branch a căror condiție este adevărată sau instrucțiuni de tip salt, complexitatea nu va depinde numai de numărul de instrucțiuni ci va depinde de câte instrucțiuni vor fi executate. Astfel, dacă un program are o buclă infinită, simulatorul nu se va opri niciodată sau dacă un program are în fișier un număr mare de instrucțiuni, dar conține un salt la ultima instrucțiune, atunci simulatorul va executa doar o parte din instrucțiuni, indiferent de cât de mare era fișierul inițial. Așadar, complexitatea va depinde de la caz la caz, iar programele executate trebuie să fie algoritmi. Pentru cazurile de test, complexitățile au fost calculate mai jos:

Notație: n este numărul de instrucțiuni din fișierul aferent programului

* instructions\_add\_load.mem
  + În cadrul acestui program, se execută toate instrucțiunile, deci complexitatea nu diferă în funcție de cazurile favorabile sau nefavorabile, deci avem O(n).
* instructions\_counter.mem
  + Acest program execută o buclă independentă de numărul de instrucțiuni ale programului, deci complexitatea lui poate fi considerată O(m), unde m este valoarea cu care este inițializat registrul t1.
* instructions\_mare.mem
  + În cadrul acestui program se execută aproape toate instrucțiunile, deci complexitatea lui este mereu O(n).
* instructions\_paritate.mem
  + Acest program are complexitatea O(n), executându-se aproape toate instrucțiunile din fișierul de intrare.
* instructions\_sll\_xor.mem
  + Acest program are complexitatea O(n), executându-se aproape toate instrucțiunile din fișierul de intrare.
* instructions\_stocari\_extrageri.mem
  + Acest program are complexitatea O(n), executându-se aproape toate instrucțiunile din fișierul de intrare.
* instructions\_subrutina.mem
  + Acest program are complexitatea O(n), executându-se toate instrucțiunile din fișierul de intrare.
* instructions\_suma.mem
  + Acest program, precum cel din instructions\_counter.mem conține o buclă independentă de numărul de instrucțiuni ale fișierului de intrare. Complexitatea lui va fi O(m).

În concluzie, complexitatea unui caz de test este dată de numărul de instrucțiuni executate de procesor, ce poate fi diferit de numărul de instrucțiuni din fișierul de intrare.

# Anexa 1. Explicații ale programelor de test

Programele de test vor fi privite ca șiruri de octeți de către simulator. Mai jos este o explicare a însemnătății acestor octeți, în funcție de fișierul folosit în cadrul programului. Așadar, din input\_mare.txt a rezultat instructions\_mare.mem și în același mod au rezultat celelalte fișiere de intrare ale simulatorului.

* input\_add\_lw.txt

00500513 //addi a0,zero,5 result=0x5

00700593 //addi a1,zero,7 result=0x7

00b50633 //add a2,a0,a1 result=0xc

00c02223 //sw a2,4(zero) result=0xc

ff460613 //addi a2,a2,-12 result=0x0

00402683 //lw a3,4(zero) result=0xc

* input\_counter.txt

00a00313 //addi t1, zero, 10 result=0xa

00100393 //addi t2,zero,1 result=0x1

40730333 //sub t1,t1,t2 intercalat cu result de mai jos cu exceptia 0x0 result=0x9 0x8 0x7 0x6 0x5 0x4 0x3 0x2 0x1 0x0

fe031ee3 //bne t1, zero,-2 result=0x8 pentru toate buclele

* input\_mare.txt

00A00093 //addi ra, zero, 10 result=0xa

01408113 //addi sp, ra, 20 result=0x1e

002081B3 //add gp, ra, sp result=0x28

40218233 //sub tp, gp, sp result=0xa

000012b7 //lui t0, 1 result=0x1000

03200313 //addi t1, zero, 50 result=0x32

0062F3B3 //and t2, t0, t1 result=0x0

0053EE33 //or t3, t2, t0 result=0x1000

007E4EB3 //xor t4, t3, t2 result=0x1000

003EFF13 //andi t5, t4, 3 result=0x0

006F6F93 //ori t6, t5, 6 result=0x6

009FC413 //xori s0, t6, 9 result=0xf

00245493 //srli s1, s0, 2 result=0x3

00949933 //sll s2, s1, s1 result=0x18

009959B3 //srl s3, s2, s1 result=0x3

00241493 //slli s1, s0, 2 result=0x3c

00302023 //sw gp, 0(zero) result=0x28

00002503 //lw a0, 0(zero) result=0x28

001122B3 //slt t0, sp, ra result=0x0

0091B333 //sltu t1, gp, s1 result=0x1

406453B3 //sra t2,s0,t1 result=0x7

02D4A693 //slti a3,s1,45 result=0x0

0049B913 //sltiu s2, s3, 4 result=0x1

40255793 //srai a5, a0, 2 result=0xa

01300223 //sb s3, 4(zero) result=0x3

00801423 //sh s0, 8(zero) result=0xf

01C03623 //sd t3, 12(zero) result=0x1000

00400C83 //lb s9, 4(zero) result=0x3

00801B83 //lh s7, 8(zero) result=0xf

00404F03 //lbu t5, 4(zero) result=0x3

00805B83 //lhu s7, 8(zero) result=0xf

00006503 //lwu a0, 0(zero) result=0x28

00C03E03 //ld t3, 12(zero) result=0x1000

00800bef //jal s7, 4 result=0x8c

08c78d67 //jalr s10, a5, 140 aceasta instructiune nu se executa

00014497 //auipc s1, 20 result=0x408c

06310263 //beq sp,gp,100 result=0x0

00a49263 //bne s1,a0,2 result=0x98

00a4c463 //blt s1,a0, 8 result=0x0

00f1d263 //bge gp,a5,2 result=0xa0

00A4E563 //bltu s1,a0, 10 result=0x0

00f1f263 //bgeu gp,a5,2 result=0xa8

* input\_paritate.txt

00900513 //addi a0,zero,9 result=0x9

00157293 //andi t0,a0,1 result=0x1

00028663 //beq t0,zero,6 result=0x0

3e700293 //addi t0,zero,999 result=0x3e7

0080036f //jal t1, 4 result=0x18

37800293 //addi t0,zero,888 aceasta instructiune nu se executa

* input\_sll\_xor.txt

00a00193 //addi gp, zero, 10 result=0xa

00200113 //addi sp, zero, 2 result=0x2

002192b3 //sll t0, gp, sp result=0x28

0021c1b3 //xor gp, gp, sp result=0x8

0032f333 //and t1, t0, gp result=0x8

002063b3 //or t2, zero, sp result=0x2

00404513 //xori a0, zero, 4 result=0x4

00506593 //ori a1, zero, 5 result=0x5

00111113 //slli sp,sp,1 result=0x4

00115113 //srli sp,sp,1 result=0x2

40115113 //srai sp,sp,1 result=0x1

00015463 //bge sp,zero,4 result=0x34

00000033 //add zero,zero,zero aceasta instructiune nu se executa

00003137 //lui sp,3 result=0x3000

fff12513 //slti a0, sp, -1 result=0x0

fff13593 //sltiu a1, sp, -1 result=0x1

ffe00213 //addi tp, zero, -2 result=0xfffe

0042a6b3 //slt a3,t0,tp result=0x0

0042b733 //sltu a4,t0,tp result=0x1

00600593 //addi a1,zero,6 result=0x6

00e5d7b3 //srl a5,a1,a4 result=0x3

4075d5b3 //sra a1,a1,t2 result=0x1

00214133 //xor sp, sp, sp result=0x0

0031c1b3 //xor gp, gp, gp result=0x0

00424233 //xor tp, tp, tp result=0x0

0052c2b3 //xor t0, t0, t0 result=0x0

00634333 //xor t1,t1,t1 result=0x0

0073c3b3 //xor t2,t2,t2 result=0x0

00b5c5b3 //xor a1,a1,a1 result=0x0

00e74733 //xor a4,a4,a4 result=0x0

00f7c7b3 //xor a5,a5,a5 result=0x0

* input\_stocari\_extrageri.txt

fff00093 //addi ra,zero,-1 result=0xffff

00100193 //addi gp,zero,1 result=0x1

00100023 //sb ra,0(zero) result=0xff

403080b3 //sub ra,ra,gp result=0xfffe

001010a3 //sh ra,1(zero) result=0xfffe

403080b3 //sub ra,ra,gp result=0xfffd

00103123 //sd ra,2(zero) result=0xfffd

0011e463 //bltu gp,ra,4 result=0x24

00000033 //add zero,zero,zero aceasta instructiune nu se executa

0030f463 //bgeu ra,gp,4 result=0x2c

00000033 //add zero,zero,zero aceasta instructiune nu se executa

00000503 //lb a0,0(zero) result=0xffff

00203583 //ld a1,2(zero) result=0xfffd

00101603 //lh a2,1(zero) result=0xfffe

00204683 //lbu a3,2(zero) result=0xfd

00005703 //lhu a4,0(zero) result=0xff

00106783 //lwu a5,1(zero) result=0xfffe

* input\_subrutina.txt

00000097 //auipc ra, 0 result=0x0

00c000ef //jal ra,6 (intrare în subrutină, se pune partea low. Împreună cu auipc este echivalentă call) result=0x10

00000013 //addi zero,zero,0(echivalent nop,aici se revine din subrutina) result=0x0 după ret

028000ef //jal ra,20(sar peste subrutina) result=0x34

00410113 //addi sp,sp,4(stiva crește de la adrese mici la mari) result=0x4(după jal ra,6)

00112023 //sw ra,0(sp) result=0x8 (valoarea lui ra)

00500513 //addi a0,zero,5 result=0x5

00300593 //addi a1,zero,3 result=0x3

00250513 //addi a0,a0,2 result=0x7

00b50533 //add a0, a0, a1 result=0xa

00012083 //lw ra, 0(sp) result=0x8 (valoarea lui ra)

ffc10113 //addi sp,sp,-4 result=0x0

000082e7 //jalr t0, ra, 0(echivalent ret) result=0x8

00000013 //addi zero,zero,0(aici am ajuns după ce am sărit din jal ra,20) result=0x0

* input\_suma.txt

00500293 //addi t0,zero,5 result=0x5

00100393 //addi t2,zero,1 result=0x1

00730333 //add t1,t1,t2 result=0x1 la prima buclă 0x3 0x6 0xa

00138393 //addi t2,t2,1 result=0x1 0x2 0x3 0x4 0x5 intercalat cu blt și add-ul precedent

fe53cce3 //blt t2,t0,-4 result=0x8 la toate buclele, apoi 0x0

00000313 //addi t1 zero 0 result=0x0