# ASAMBLOR PENTRU UN PROCESOR RISC

Cătinean Andrei

SSC

# Cuprins

# 1.Introducere

## 1.1 Context

Asamblorul este o componentă esențială a dezvoltării software și a programării pentru procesorul RISC. Dezvoltarea unui asamblor dedicat pentru un anumit procesor RISC are multiple beneficii:

- Optimizare a performanței: Asamblorul permite programatorilor să aibă control precis asupra instrucțiunilor și acces la resursele hardware, ceea ce duce la optimizări ale performanței aplicațiilor.
- Limbaj de nivel inferior: Asamblorul oferă programatorilor acces la nivelul cel mai de jos al procesorului, permițând lucrul direct cu registre și memorie.
- Învățarea și înțelegerea mai profundă: Dezvoltarea unui asamblor este o oportunitate excelentă pentru a înțelege în profunzime funcționarea procesorului RISC și pentru a obține cunoștințe avansate despre arhitectura hardware.

## 1.2 Solutia propusa

Cerinte generale: Dezvoltarea unui Asamblor pentru un processor RISC. Dezvoltarea unei aplicatii Windows cu meniuri care generează un fișier VHDL pentru modulul memoriei de instrucțiuni conținând codul executabil al instrucțiunilor din fișierul sursă.

Design ales:

- Intreg proiectul va fi implementat in limbajul de programare Python

## 1.3 Plan de proiect

### 1.3.1 Studiu Bibliografic

Obiectiv:

Să se obțină o înțelegere solidă a cerințelor proiectului și a contextului general al dezvoltării unui asamblor pentru un procesor RISC.

Activități:

1. Investigarea Cerințelor Proiectului:
   - Revizuirea detaliată a cerințelor generale pentru dezvoltarea asamblorului.
   - Identificarea funcționalităților necesare pentru a satisface cerințele proiectului.
2. Cunoașterea Arhitecturii Procesorului RISC:
   - Studierea specificului arhitecturii RISC, concentrându-se pe aspecte precum setul de instrucțiuni, modurile de adresare și optimizările de performanță asociate.
3. Explorarea Resurselor Online:
   - Căutarea și analiza surselor online precum documentații oficiale, articole științifice și tutoriale despre dezvoltarea asamblorului și arhitectura procesorului RISC.
4. Compararea cu Proiecte Anterioare:
   - Identificarea și analiza altor proiecte similare de dezvoltare a asamblorului pentru procesorul RISC.
   - Evaluarea problemelor întâlnite și a soluțiilor implementate în proiecte anterioare.

### 1.3.2 Analiză

Obiectiv:

Definirea clară a cerințelor și specificațiilor pentru aplicația asamblorului și stabilirea fundației pentru faza de design.

Activități:

1. Stabilirea Specificațiilor Tehnice:
   - Detalierea specificațiilor tehnice ale asamblorului, inclusiv formatul instrucțiunilor, modurile de adresare acceptate și funcționalitățile cheie.
   - Stabilirea interfeței de comunicare între asamblor și modulul de memorie de instrucțiuni.
2. Analiza Modurilor de Adresare:
   - Evaluarea modurilor de adresare relevante pentru arhitectura procesorului RISC și integrarea acestora în cerințele asamblorului.

### 1.3.3 Design

Obiectiv:

Dezvoltarea arhitecturală a aplicației asamblorului, definirea componentelor și a interacțiunilor dintre acestea.

Activități:

1. Proiectarea Structurii de Bază:
   - Definirea arhitecturii de bază a asamblorului, inclusiv componente precum analizorul lexical, analizorul sintactic și generatorul de cod.
   - Stabilirea modului în care aceste componente interacționează pentru a traduce codul sursă în cod mașină.
2. Detalierea Fluxului de Lucru:
   - Definirea clară a fluxului de lucru al asamblorului, începând de la citirea codului sursă și până la generarea fișierului VHDL.
   - Identificarea punctelor critice și a deciziilor cheie în fluxul de lucru.
3. Stabilirea Convențiilor de Codificare:
   - Stabilirea unor convenții clare de codificare pentru asamblor, astfel încât să se asigure coerența și ușurința în dezvoltare.

### 1.3.4 Implementare

Obiectiv:

Scrierea codului efectiv pentru asamblor și pentru interfața grafică asociată.

Activități:

1. Implementarea Analizorului Lexical și Sintactic:
   - Dezvoltarea componentelor care realizează analiza lexicală și sintactică a codului sursă.
2. Scrierea Generatorului de Cod:
   - Implementarea modulului responsabil pentru generarea codului mașină pe baza rezultatelor analizei.
3. Dezvoltarea Interfeței Grafice:
   - Implementarea interfeței grafice pentru aplicația Windows, inclusiv meniurile și funcționalitățile de interacțiune cu utilizatorul.

1.3.5 Testare

Obiectiv:

Efectuarea unor teste exhaustive pentru a identifica și corecta erorile și pentru a evalua performanța asamblorului.

Activități:

1. Testarea Unitară:
   - Efectuarea testelor individuale pentru fiecare componentă a asamblorului.
   - Identificarea și remedierea erorilor la nivel de module.
2. Testarea Integrată:
   - Verificarea interacțiunilor dintre componente prin testarea integrată a întregului sistem.
3. Testarea Funcționalităților Cheie:
   - Evaluarea performanței și corectitudinii funcționalităților cheie, precum generarea corectă a codului mașină și interacțiunea cu modulul de memorie de instrucțiuni.

# 2.Studiu bibliografic

## 2.1 Overview

Un asamblor este un program sau o utilitate informatică care traduce codul sursă scris în limbaj de asamblare în cod mașină, care este un limbaj de bază pe care un procesor îl poate înțelege și executa.

## 2.2 RISC vs CISC

In cazul acesta, asamblorul va fi folosit pentru un procesor RISC. RISC și CISC sunt două abordări diferite în proiectarea arhitecturilor de calculatoare. Arhitectura RISC se concentrează pe simplificarea setului de instrucțiuni și pe eficiența execuției, în timp ce arhitectura CISC oferă o gamă largă de instrucțiuni complexe pentru a satisface nevoile programatorilor. Fiecare abordare are avantaje și dezavantaje, iar alegerea între ele depinde de cerințele specifice ale aplicațiilor și de obiectivele de performanță ale sistemului.

Exemple de instrucțiuni RISC:

- LOAD: Încarcă un registru cu o valoare din memorie.
- STORE: Stochează valoarea dintr-un registru în memorie.
- ADD: Realizează o operație de adunare între două registre.
- SUBTRACT: Realizează o operație de scădere între două registre.
- JUMP: Realizează un salt necondiționat la o adresă specificată.
- Exemple de instrucțiuni CISC:

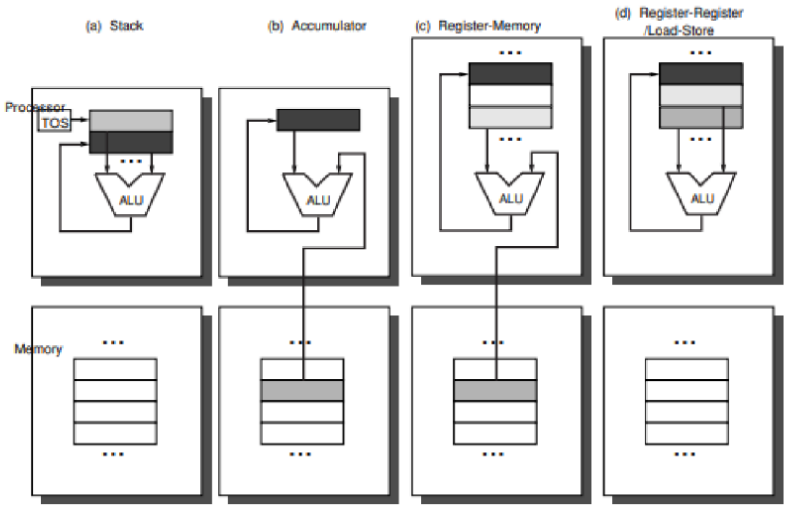
Arhitectura CISC are un set mai mare și mai variabil de instrucțiuni. Acestea pot include instrucțiuni complexe care efectuează mai multe operații într-o singură instrucțiune. Unele exemple de instrucțiuni CISC includ:

- MOV: Transferă date între registre sau memorie.
- ADD: Adună două valori și stochează rezultatul într-un registru.
- MUL: Realizează o operație de înmulțire între două valori.
- DIV: Realizează o operație de împărțire între două valori.
- LOAD: Încarcă date din memorie într-un registru, dar poate implica și alte operații, precum deplasarea sau scalarea datelor.

## 2.3 Moduri de adresare

În limbajul de asamblare, există mai multe moduri de a accesa și manipula datele stocate în memorie.  Iată câteva tipuri comune de moduri de adresare a memoriei în limbajul de asamblare:

- Adresare Directă: Aceasta este cea mai simplă formă de adresare. Adresa sau locația de memorie la care se află datele este specificată direct în instrucțiune.
- Adresare Indirectă: În acest mod, adresa efectivă a datelor nu este specificată direct în instrucțiune. În schimb, se utilizează un registru sau un alt loc de memorie care conține adresa.
- Adresare Indexată: Acest mod de adresare implică adăugarea sau scăderea unei valori la un registru pentru a calcula adresa efectivă.
- Adresare cu Offset Constant: Se adaugă sau se scade o constantă la o adresă de bază pentru a obține adresa efectivă.
- Adresare cu Index și Scară (Index Scaling): Acest mod de adresare implică înmulțirea valorii unui registru (index) cu o scară specificată și apoi adăugarea sau scăderea de la o adresă de bază. Acest lucru este folosit des în arhitecturile moderne pentru a accesa tabele și structuri de date complexe.
- Adresare cu Registrul Program Counter (PC): Adresa de memorie este calculată utilizând valoarea curentă a Program Counter (PC), care indică adresa instrucției următoare.



## 2.4 Optimizări ale Performanței și Eficienței în Dezvoltarea Asamblorului:

Dezvoltarea unui asamblor eficient implică implementarea unor optimizări pentru a îmbunătăți performanța și eficiența sa. Aceste optimizări pot include:

- Eliminarea instrucțiunilor redundante: Identificarea și eliminarea instrucțiunilor care nu contribuie la rezultatul final.
- Reordonarea instrucțiunilor: Rearanjarea instrucțiunilor pentru a profita de caracteristicile hardware ale procesorului și pentru a minimiza ciclurile goale de ceas.
- Utilizarea de registre eficient: Minimizarea acceselor la memorie prin stocarea temporară a datelor în registre.
- Îmbunătățirea strategiilor de salt: Optimizarea instrucțiunilor de salt pentru a minimiza ciclurile de execuție.

## 2.5 Parsarea codului

Parsarea codului în limbaj de asamblare implică procesul de analiză și interpretare a codului sursă în limbaj de asamblare pentru a identifica instrucțiunile, etichetele, modurile de adresare și operanzii. Există mai multe metode de parsare a codului în limbaj de asamblare, fiecare cu propriile sale caracteristici. Iată câteva dintre aceste metode:

- Analiza Linie cu Linie: Această metodă implică citirea codului sursă linie cu linie și identificarea instrucțiunilor și a etichetelor pe baza regulilor limbajului de asamblare. În timp ce se parcurge fiecare linie, se pot construi structuri de date care să rețină informațiile necesare pentru generarea codului mașină. Această metodă este potrivită pentru codurile sursă simple și ușor de citit.
- Analiza cu Ajutorul Expresiilor Regulate: Expresiile regulate pot fi folosite pentru a identifica și extrage șabloane specifice din codul sursă. Aceste șabloane pot reprezenta instrucțiuni, etichete sau alte elemente semnificative. Odată identificate, aceste elemente pot fi procesate în consecință.

- Folosirea Limbajelor de Programare de Înalt Nivel: În unele cazuri, dezvoltatorii pot folosi limbaje de programare de înalt nivel, precum Python sau C++, pentru a implementa parsarea codului în limbaj de asamblare. Aceasta poate facilita dezvoltarea și depanarea analizatorului.

# 3 Analiza

## 3.1 Schema bloc simplificata

Procesorul RISC se caracterizează prin execuția majorității instrucțiunilor într-un singur ciclu de ceas și utilizează tehnica de pipeline pentru a permite reducerea perioadei semnalului de ceas. Acest lucru se realizează împărțind calea de date în mai multe etaje cu registre intermediare pentru a păstra rezultatele parțiale și a permite transferul acestora între etaje. Astfel, fiecare etaj termină execuția într-un ciclu de ceas, permițând o creștere a frecvenței de ceas.

Caracteristicile cheie ale acestui procesor RISC includ:

- Memorii separate pentru instrucțiuni și date.
- Arhitectură de tip Load/Store, unde operațiile sunt efectuate doar între registre, iar memoria de date este accesată doar prin instrucțiuni de încărcare și memorare.
- Trei formate de instrucțiuni, toate cu aceeași lungime de 32 de biți.
- Moduri de adresare reduse și instrucțiuni simple care pot fi executate într-un singur ciclu de ceas.
- Unitatea de control conține contorul de program (PC), memoria de instrucțiuni, registrul de instrucțiuni (RI), decodificatorul de instrucțiuni și registrele dintre etajele pipeline.
- Contorul de program (PC) se actualizează în fiecare ciclu de ceas și conține adresa instrucțiunii care va fi extrasă din memoria de instrucțiuni.
- Decodificatorul de instrucțiuni generează semnalele de comandă necesare funcționării procesorului.

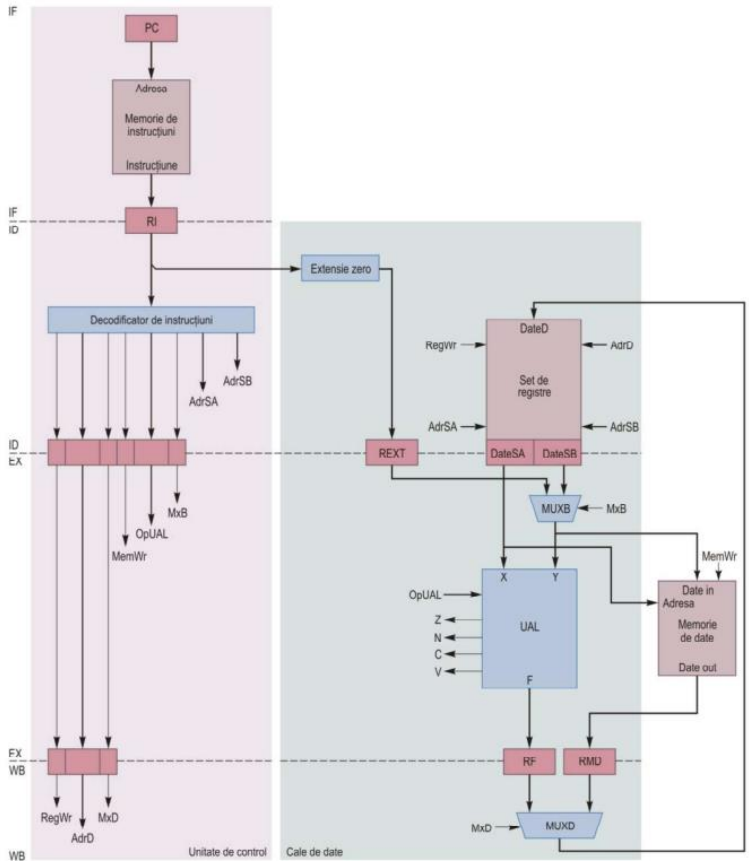
Această structură și utilizarea memorii separate de instrucțiuni și date permit extragerea și execuția unei instrucțiuni într-un singur ciclu de ceas.

Decodificatorul de instrucțiuni generează semnalele de comandă necesare pentru funcționarea procesorului RISC. Iată o scurtă descriere a acestor semnale:

- AdrSA: Acest semnal reprezintă adresa registrului sursă A, care conține primul operand al instrucțiunii.
- AdrSB: Acest semnal reprezintă adresa registrului sursă B, care conține al doilea operand al instrucțiunii.
- AdrD: Acest semnal reprezintă adresa registrului destinație, care va conține rezultatul operației executate de instrucțiune.
- MxB: Semnalul de selecție pentru multiplexorul MUXB, care decide ce intră în unitatea aritmetică și logică (UAL). Acesta poate fi conținutul registrului sursă B sau o valoare constantă dintr-un câmp al instrucțiunii.
- OpUAL: Acest semnal reprezintă codul funcției executate de unitatea aritmetică și logică (UAL). Acest cod specifică operația care trebuie efectuată, cum ar fi adunare, scădere, înmulțire, sau operații logice.
- MemWr: Semnalul care validează operația de scriere în memoria de date. Acest semnal indică dacă datele trebuie să fie scrise în memoria de date în urma unei instrucțiuni.
- MxD: Semnalul de selecție pentru multiplexorul MUXD, care decide ce intră în setul de registre la portul de intrare DateD. Acesta poate fi ieșirea unității aritmetice și logice sau un cuvânt citit din memoria de date.
- RegWr: Semnalul de validare a scrierii în setul de registre. Acest semnal indică dacă datele trebuie să fie scrise în setul de registre, inclusiv registrul destinație al instrucțiunii.

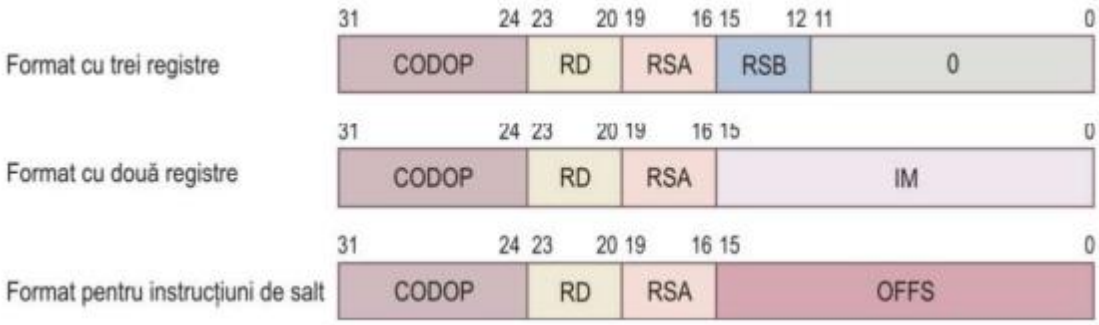
Calea de date include setul de registre, circuitul pentru extensia cu zero a constantelor de 16 biți la valori de 32 de biți, unitatea aritmetică și logică (UAL), multiplexoarele MUXB și MUXD, precum și registrele necesare pentru transferul informațiilor între etajele pipeline.

Setul de registre are 16 registre de 32 de biți și operează ca o memorie rapidă cu 16 cuvinte și trei porturi de acces. Aceste porturi permit citirea conținutului registrilor sursă A și B, precum și scrierea în registrul destinație al instrucțiunii. Toate cele trei operații pot avea loc în același ciclu de ceas și sunt validate prin semnalul RegWr.




## 3.2 Formatul instructiunilor

Procesorul pentru care este dezvoltat asamblorul suporta 3 tipuri de instructiuni.




Procesorul RISC are 16 registre de 32 de biți.Ele sunt notate R0, R1 etc. R0 poate fi utilizat ca oricare alt registru. Pentru a seta un registru la valoarea zero, se poate folosi o operație logică SAU EXCLUSIV cu același registru.

Instrucțiunile procesorului RISC folosesc un cuvânt de 32 de biți, cu 8 biți cei mai semnificativi reprezentând codul operației (CODOP).

**Primul** format implică trei registre: două pentru surse (RSA și RSB) și unul pentru destinație (RD).

**Al doilea** format al instrucțiunilor utilizează două registre: registrul sursă A (adresat de câmpul RSA) și registrul destinație (adresat de câmpul RD). Acest format este folosit în instrucțiunile aritmetice și logice care necesită o valoare imediată ca al doilea operand. Valoarea imediată este preluată din câmpul IM de 16 biți al instrucțiunii. De asemenea, câmpul IM este utilizat pentru a specifica numărul de poziții cu care se deplasează conținutul registrului sursă în instrucțiunile de deplasare logică la dreapta sau la stânga. Numărul de poziții cu care se realizează deplasarea este reprezentat în cei cinci biți mai puțin semnificativi ai câmpului IM.

**Al treilea** format al instrucțiunilor este similar cu al doilea format, dar include și câmpul OFFS, care conține deplasamentul adresei de destinație pentru instrucțiunile de salt și instrucțiunea de apel al unei proceduri. Adresa de destinație a acestor instrucțiuni este obținută prin adunarea deplasamentului din câmpul OFFS la conținutul contorului de program PC. Prin urmare, instrucțiunile de salt și instrucțiunea de apel folosesc adresarea relativă, unde registrul PC este actualizat prin adunarea valorii din câmpul OFFS la conținutul acestui registru, valoare considerată un număr cu semn în complement față de doi. În cazul instrucțiunii de salt necondiționat, se utilizează doar câmpul OFFS al instrucțiunii și nu se folosesc câmpurile RSA și RD. În schimb, pentru instrucțiunile de salt condiționat, se adaugă câmpul RSA, care specifică registrul sursă a cărui conținut este testat pentru a determina dacă saltul va fi executat sau nu. În instrucțiunea de apel al unei proceduri, câmpul RSA este înlocuit cu câmpul RD, care indică registrul în care se va stoca adresa de revenire.

3.3 Tipuri de instructiuni

Tabelul descrie operațiile efectuate de instrucțiunile procesorului RISC și furnizează exemple de cod asamblare pentru fiecare instrucțiune.

| Mnemonica | Operație | Exemplu |
|---|---|---|
| NOP | Nicio operație | NOP |
| MOVA | R(RD) ← R(RSA) | MOVA R1,R4 |
| ADD | R(RD) ← R(RSA) + R(RSB) | ADD R2,R1,R4 |
| SUB | R(RD) ← R(RSA) – R(RSB) | SUB R3,R7,R8 |
| AND | R(RD) ← R(RSA) **and** R(RSB) | AND R4,R3,R6 |
| OR | R(RD) ← R(RSA) **or** R(RSB) | OR R4,R5,R7 |
| XOR | R(RD) ← R(RSA) **xor** R(RSB) | XOR R8,R2,R4 |
| NOT | R(RD) ← **not** R(RSA) | NOT R9,R1 |
| ADDI | R(RD) ← R(RSA) + *exts* (IM) | ADDI R2,R0,4 |
| SUBI | R(RD) ← R(RSA) – *exts* (IM) | SUBI R5,R7,1 |
| ANDI | R(RD) ← R(RSA) **and** *extz* (IM) | ANDI R8,R2,0x8000 |
| ORI | R(RD) ← R(RSA) **or** *extz* (IM) | ORI R1,R3,0x4000 |
| XORI | R(RD) ← R(RSA) **xor** *extz* (IM) | XORI R6,R6,1 |
| ADDU | R(RD) ← R(RSA) + *extz* (IM) | ADDU R2,R3,128 |
| SUBU | R(RD) ← R(RSA) – *extz* (IM) | SUBU R5,R7,10 |
| MOVB | R(RD) ← R(RSB) | MOVB R4,R5 |
| SHR | R(RD) ← **shr** (R(RSA)) cu IM poziții | SHR R2,R2,4 |
| SHL | R(RD) ← **shl** (R(RSA)) cu IM poziții | SHL R6,R6,8 |
| LD | R(RD) ← M(R(RSA)) | LD R8,R3 |
| ST | M(R(RSA)) ← R(RSB) | ST R3,R5 |
| JMPR | PC ← R(RSA) | JMPR R3 |
| SGTE | **if** R(RSA) ≥ R(RSB) **then** R(RD) ← 1 | SGTE R4,R5,R7 |
| SLT | **if** R(RSA) < R(RSB) **then** R(RD) ← 1 | SLT R4,R5,R7 |
| BZ | **if** R(RSA) = 0 **then** PC ← PC + 1 + *exts* (OFFS) | BZ R3,8 |
| BNZ | **if** R(RSA) ≠ 0 **then** PC ← PC + 1 + *exts* (OFFS) | BNZ R3,-12 |
| JMP | PC ← PC + 1 + *exts* (OFFS) | JMP 10 |
| JMPL | R(RD) ← PC + 1, PC ← PC + 1 + *exts* (OFFS) | JMPL R8,16 |
| HALT | PC ← PC | HALT |

Instrucțiunile de încărcare (LD) și de memorare (ST) accesează locații de memorie specificate de M(R(RSA)). Operațiile sunt în general simple și constau în transferuri între registre. Instrucțiunile cu un operand imediat permit reducerea acceselor la memoria de date atunci când se lucrează cu constante. Valoarea imediată este de 16 biți, așa că trebuie să fie extinsă pentru a forma un operand de 32 de biți.

- Pentru operațiile logice, valoarea imediată este extinsă cu zerouri în cele 16 biți mai semnificativi, notată ca extz (IM).
- Pentru operațiile aritmetice, valoarea imediată este extinsă cu semn, notată ca exts (IM). Bitul de semn al valorii imediate (bitul 15) este copiat în cei 16 biți mai semnificativi ai operandului pentru a forma un operand de 32 de biți în complement față de doi.

De asemenea, se folosește o notare similară, exts (OFFS), pentru a indica extinderea cu semn a câmpului OFFS care conține deplasamentul adresei destinație pentru instrucțiunile de salt.

Instrucțiunile de salt condiționat permit controlul fluxului programului în funcție de condiții specifice:

- BZ (Branch If Zero): Această instrucțiune determină dacă registrul specificat conține zero și efectuează un salt (o schimbare a adresei de program) dacă acesta este zero.
- BNZ (Branch If Not Zero): Similar cu BZ, această instrucțiune verifică dacă registrul specificat conține o valoare diferită de zero și realizează un salt dacă este adevărat.
- SGTE (Set If Greater Than or Equal): Această instrucțiune înregistrează valoarea 1 în registrul destinație dacă registrul sursă A conține o valoare mai mare sau egală cu cea din registrul sursă B. În caz contrar, înregistrarea va conține valoarea 0. Aceasta permite efectuarea de comparații între două valori.
- SLT (Set If Less Than): Instrucțiunea SLT înregistrează valoarea 1 în registrul destinație dacă registrul sursă A conține o valoare mai mică decât cea din registrul sursă B. În caz contrar, înregistrarea va conține valoarea 0. Aceasta este folosită pentru comparații între două valori.

Instrucțiunea JMPL (Jump and Link) permite realizarea de apeluri de proceduri. Ea efectuează următoarele acțiuni:

- Conținutul registrului PC se incrementează și se salvează în registrul destinație specificat în instrucțiune.
- Suma dintre conținutul registrului PC și extensia cu semn a câmpului OFFS din instrucțiune se înregistrează în registrul PC, realizând saltul la adresa specificată.

Pentru revenirea din procedura apelată, se poate utiliza instrucțiunea JMPR (Jump Register) cu același registru sursă A care a fost utilizat ca registru destinație

în instrucțiunea JMPL. Pentru apelarea altor proceduri într-o procedură, se recomandă utilizarea altor registre pentru păstrarea adreselor de revenire sau o stivă software care să gestioneze adresele de revenire din și în registrul sursă A. Aceasta asigură o gestionare corectă a fluxului programului într-un mediu RISC.

### 3.4 Codificarea operatiilor

Pentru a simplifica decodificarea instrucțiunilor, codurile operațiilor sunt selectate astfel încât cei patru biți mai puțin semnificativi ai acestora să corespundă cu codul de selecție al operației respective efectuate de Unitatea Aritmetică și Logică (UAL) ori de câte ori este utilizat acest cod. În acest fel, semnalul OpUAL, care este aplicat la intrarea de selecție a operației executate de UAL, poate fi direct extras din cei patru biți mai puțin semnificativi ai câmpului codului operației CODOP.

| Mnemonica | CODOP |
|---|---|
| NOP | 0000 0000 |
| MOVA | 0100 0000 |
| ADD | 0000 0010 |
| SUB | 0000 0101 |
| AND | 0000 1000 |
| OR | 0000 1001 |
| XOR | 0000 1010 |
| NOT | 0000 1011 |
| ADDI | 0010 0010 |
| SUBI | 0010 0101 |
| ANDI | 0010 1000 |
| ORI | 0010 1001 |
| XORI | 0010 1010 |
| ADDU | 0100 0010 |
| SUBU | 0100 0101 |
| MOVB | 0000 1100 |
| SHR | 0000 1101 |
| SHL | 0000 1110 |
| LD | 0001 0000 |
| ST | 0010 0000 |
| JMPR | 0111 0000 |
| SGTE | 0111 0101 |
| SLT | 0110 0101 |
| BZ | 0110 0000 |
| BNZ | 0101 0000 |
| JMP | 0110 1000 |
| JMPL | 0011 0000 |
| HALT | 0110 1001 |

### 3.5 Moduri de adresare

Procesorul RISC operează cu patru moduri distincte de adresare: adresarea registrelor, adresarea indirectă prin registru, adresarea imediată și adresarea relativă. În acest context, modul de adresare asociat unei instrucțiuni este determinat exclusiv de câmpul codului operației din instrucțiune, și nu prin intermediul unui câmp separat dedicat modului de adresare. Prin urmare, modul de adresare pentru o instrucțiune dată este fix și nu poate fi modificat ulterior.

Instrucțiunile cu formatul ce implică trei registre utilizează modul de adresare numit "adresarea registrelor." Adresarea indirectă prin registru este specifică doar instrucțiunilor LD și ST, acestea fiind singurele instrucțiuni care interacționează direct cu memoria de date. Instrucțiunile cu formatul bazat pe două registre folosesc adresarea imediată, iar valoarea imediată este disponibilă în câmpul IM al instrucțiunii. Adresarea relativă este asociată instrucțiunilor de salt și instrucțiunii de apel al unei proceduri, astfel încât adresele generate prin acest mod de adresare se referă exclusiv la memoria de instrucțiuni.

În situația în care un program necesită utilizarea unui mod de adresare indisponibil, cum ar fi adresarea indexată, este necesară implementarea unei secvențe de instrucțiuni suplimentare pentru a simula funcționalitatea dorită.

# 4 Design

### 4.1 Structura proiectului

Proiectul va contine urmatoarele clase de baza:

- **Parser** : Clasa care va citi fiecare linie din fisierul text primit ca intrare si va genera codul binar al instructiunilor din fisier in cazul in care nu sunt prezente erori
- **MemoryGenerator** : Clasa care primeste codul binar al instructiunilor si genereaza fisierul VHDL cu modulul memoriei de instructiuni continand codul executabil
- **UserInterface**: Această clasă gestioneaza interacțiunea cu utilizatorul

### 4.2 Arhitectura proiectului

Pentru acest proiect am ales arhitectura MVC. Model-View-Controller (MVC) este un pattern arhitectural utilizat în dezvoltarea software pentru a organiza și structura codul într-un mod care să ofere o descompunere clară a responsabilităților și să faciliteze gestionarea complexității proiectelor.

**1. Model (M)** - Parser și MemoryGenerator

Clasa Parser comunica cu clasa MemoryGenerator pentru a genera conținutul memoriei.

**2. View (V)** - UserInterface

Clasa UserInterface gestionează interacțiunea cu utilizatorul, afișarea rezultatelor și gestionarea evenimentelor

**3. Controller (C)**

Clasa Controller coordonează interacțiunea dintre Model și View. Principalele sale responsabilități includ:

- Procesarea comenzilor utilizatorului și inițierea procesului de asamblare.
- Comunicarea cu Model pentru a realiza procesul de asamblare și pentru a primi rezultatele.
- Comunicarea cu View pentru a afișa rezultatele și pentru a gestiona evenimentele utilizatorului.

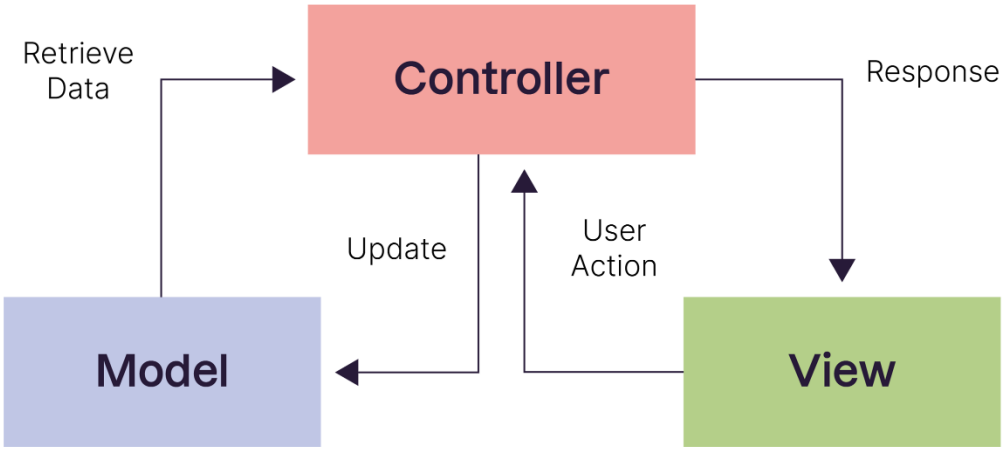
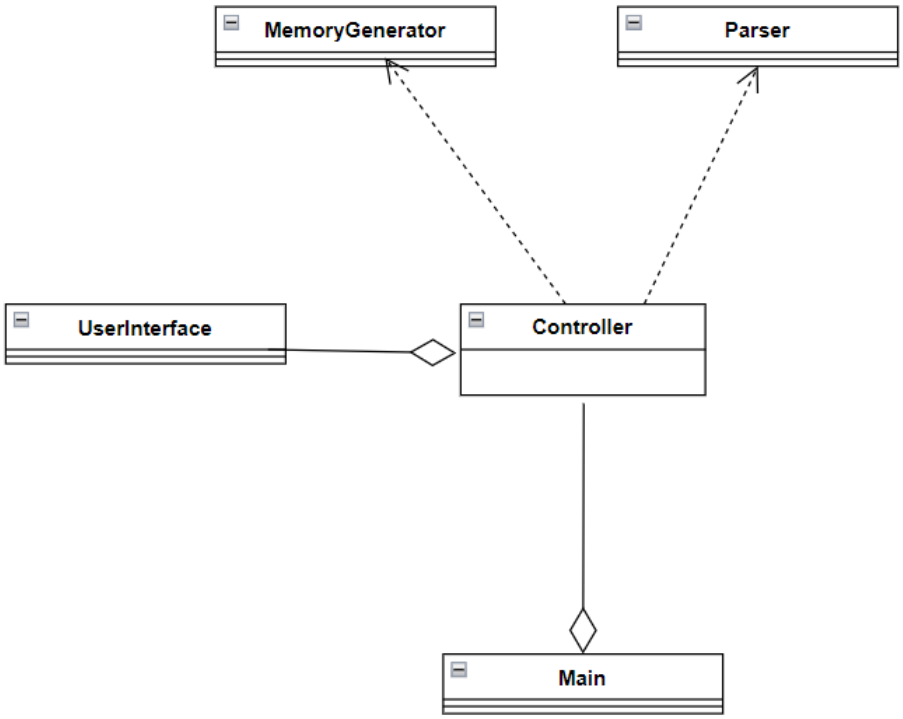


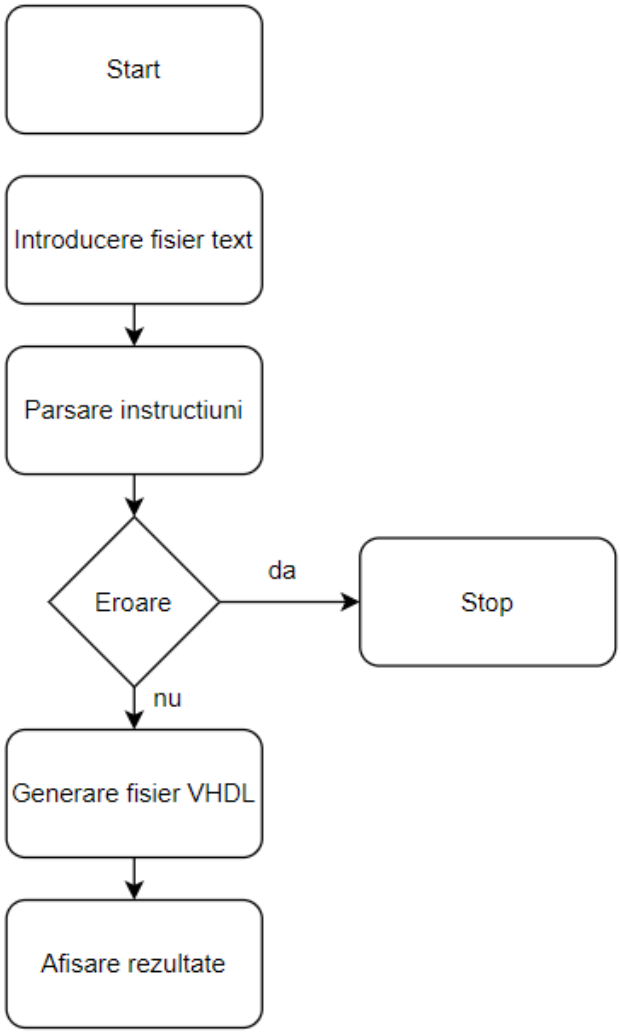
Diagrama de UML:




4.3 Flow-ul de executie

- Citirea Intrării: Utilizatorul introduce un fișier de asamblare.
- Asamblare: Clasa Assembler citește și parsează instrucțiunile.
- Generare Conținut Memorii: Clasa MemoryGenerator generează conținutul memoriei.
- Afișare Rezultate: Clasa UserInterface afișează rezultatele.

Flowchart:


Start
Introducere fisier text
Parsare instructiuni
Eroare
da
Stop
nu
Generare fisier VHDL
Afisare rezultate

# 5 Implementare

## 5.1 Clasa Parser

Metoda parse_file:

```
def parse_file(self, text_file):
    binary_codes = []
    line_nr = 1
    with open(text_file, 'r') as file:
        for line in file:
            binary_codes.append(self.parse_instruction(line.strip(), line_nr))
            line_nr = line_nr + 1
    print(binary_codes)
    return binary_codes
```

- Primește un fișier de text (text_file) ca parametru.
- Deschide fișierul și parcurge fiecare linie din el.
- Pentru fiecare linie, apelează parse_instruction pentru a genera codul binar corespunzător instrucțiunii și adaugă rezultatul la lista binary_codes.
- Returnează lista binary_codes care conține codurile binare pentru toate instrucțiunile din fișier.

Metoda parse_instruction:

- Primește o instrucțiune și numărul liniei corespunzătoare.
- Inițializează binary_code ca șir gol.
- Definește un dicționar opcodes care asociază codurile operaționale cu numele instrucțiunilor.

```
def parse_instruction(self, instruction, line):
    binary_code = ""

    opcodes = {
        'NOP': '00000000',
        'MOVA': '01000000',
        'ADD': '00000010',
        'SUB': '00000101',
        'AND': '00001000',
        'OR': '00001001',
        'XOR': '00001010',
        'NOT': '00001011',
        'ADDI': '00100010',
```

- Încearcă să valideze formatul instrucțiunii și să genereze codul binar corespunzător pe baza acesteia.
- În caz de eroare, aruncă o excepție de tip ValueError cu un mesaj care include detaliile erorii și numărul liniei.

```
instruction = instruction.replace(',', ' ')
parts = instruction.split()

opcode = opcodes.get(parts[0])
if opcode is None:
    raise ValueError(f"Invalid operation: {parts[0]} at line {str(line)}")

try:
    self.validate_instruction_format(parts[0], len(parts) - 1)

    if parts[0] in {'NOP', 'HALT'}:
        binary_code = opcode + '00000000000000000000000'
    elif parts[0] in {'MOVA', 'NOT', 'LD'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + self.register_to_binary(
            parts[2]) + '0000000000000000'
    elif parts[0] in {'MOVB'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + '0000' + self.register_to_binary(
            parts[2]) + '000000000000'
```

```
    elif parts[0] in {'ST'}:
        binary_code = opcode + '0000' + self.register_to_binary(parts[1]) + self.register_to_binary(
            parts[2]) + '000000000000'
    elif parts[0] in {'ADD', 'SUB', 'AND', 'OR', 'XOR', 'SGTE', 'SLT'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + self.register_to_binary(
            parts[2]) + self.register_to_binary(parts[3]) + '000000000000'
    elif parts[0] in {'ADDI', 'SUBI', 'ANDI', 'ORI', 'XORI'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + self.register_to_binary(
            parts[2]) + self.immediate_to_binary(parts[3])
    elif parts[0] in {'ADDU', 'SUBU', 'SHR', 'SHL'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + self.register_to_binary(
            parts[2]) + self.immediate_to_binary(parts[3], unsigned: True)
    elif parts[0] in {'JMPR'}:
        binary_code = opcode + '0000' + self.register_to_binary(parts[1]) + '0000000000000000'
    elif parts[0] in {'BZ', 'BNZ'}:
        binary_code = opcode + '0000' + self.register_to_binary(parts[1]) + self.immediate_to_binary(parts[2])
    elif parts[0] in {'JMP'}:
        binary_code = opcode + '00000000' + self.immediate_to_binary(parts[1])
    elif parts[0] in {'JMPL'}:
        binary_code = opcode + self.register_to_binary(parts[1]) + '0000' + self.immediate_to_binary(parts[2])

    print(str(len(binary_code)) + " " + binary_code)
    return binary_code
except Exception as e:
    raise ValueError(f"Error: {e}. Line {str(line)}")
```

Metoda register_to_binary:

```
def register_to_binary(self, register):
    if not register.startswith("R") or not register[1:].isdigit():
        raise ValueError("Invalid register format. Use format 'Rx' where x is a number between 0 and 15.")

    register_number = int(register[1:])

    if not (0 <= register_number <= 15):
        raise ValueError("Invalid register number. Use a number between 0 and 15.")

    return format(register_number, '04b')
```

- Primește un nume de registru (register) și verifică dacă acesta respectă formatul "Rx", unde x este un număr între 0 și 15.
- Converteste numărul de registru la o reprezentare binară pe 4 biți.

Metoda immediate_to_binary:

```
def immediate_to_binary(self, immediate, unsigned=False):
    if immediate.startswith("B'") and immediate.endswith("'"):
        binary_value = immediate[2:-1]
        if not all(bit in "01" for bit in binary_value) or len(binary_value) > 16:
            raise ValueError("Invalid binary value. Use a valid binary format (e.g., b'001').")
        immediate = int(binary_value, 2)
    elif immediate.startswith("X'") and immediate.endswith("'"):
        hex_value = immediate[2:-1]
        if not all(c in "0123456789ABCDEF" for c in hex_value) or len(hex_value) > 4:
            raise ValueError("Invalid hexadecimal value. Use a valid hexadecimal format (e.g., x'FFFF').")
        immediate = int(hex_value, 16)
    else:
        try:
            immediate = int(immediate)
        except ValueError:
            raise ValueError(
                "Invalid immediate value. Use an integer, binary (e.g., b'001'), or hexadecimal (e.g., x'FFFF').
        if unsigned and immediate < 0:
            raise ValueError("Unsigned immediate value must be non-negative")
    value = immediate & 0xFFFF
    return format(value, '016b')
```

- Primește o valoare imediată (immediate) care poate fi un număr întreg, un șir binar sau un șir hexazecimal.
- Încearcă să valideze și să convertească valoarea imediată într-o reprezentare binară pe 16 biți. Ridică o excepție dacă formatul nu este valid.

Metoda validate_instruction_format:

```
    'JMPR': 1,
    'SGTE': 3,
    'SLT': 3,
    'BZ': 2,
    'BNZ': 2,
    'JMP': 1,
    'JMPL': 2,
    'HALT': 0
}

expected_count = expected_operands.get(instruction_name)
if expected_count is None or num_operands != expected_count:
    raise ValueError(f"Invalid number or format of operands for {instruction_name} ")
```

- Primește numele unei instrucțiuni (instruction_name) și numărul de operanzi pentru acea instrucțiune (num_operands).
- Verifică dacă numărul de operanzi corespunde cu cel așteptat pentru instrucțiunea respectivă. Ridică o excepție ValueError în caz contrar.

5.2 Clasa MemoryGenerator

Metoda generate_vhdl_from_memory_content:

- Primește un conținut de memorie (memory_content) și un nume opțional pentru fișierul de ieșire (output_file_name).
- Construiește un șir de cod VHDL care definește o entitate (Instruction_Memory) și arhitectura acesteia.
- Defineste un tablou (memorie ROM) cu 256 de elemente, fiecare fiind un șir de 32 de biți.
- Inițializează un semnal rom cu valorile specificate în memory_content.
- Salvează codul VHDL într-un fișier cu numele specificat (output_file_name).

```
type rom_mem is array(0 to 255) of std_logic_vector(31 downto 0);
signal rom: rom_mem:=(
"""
        for index, value in enumerate(memory_content):
            vhdl_code += f"    {index} => B\"{value}\",\n"

        vhdl_code += """    others => x"00000000"
    );

begin

Instruction<=rom(conv_integer(PC));

end Behavioral;
"""

        with open(output_file_name, "w") as vhdl_file:
            vhdl_file.write(vhdl_code)
```

5.3 AssemblerController

Metoda start_assembling:

- Obține calea către fișierul de intrare (file_path) și numele fișierului de ieșire (output_file_name) din interfața grafică.
- Elimină liniile goale din fișierul de intrare.
- Creează instanțe ale claselor Parser și MemoryGenerator.
- Parsează fișierul de intrare pentru a obține codurile binare utilizând parse_file din clasa Parser.
- Generează un fișier VHDL utilizând codurile binare și generate_vhdl_from_memory_content din clasa MemoryGenerator.
- Afisează un mesaj de succes în interfața grafică.

```
def start_assembling(self):
    file_path = self.gui.file_path.get()
    output_file_name = self.gui.output_file_name.get()
    try:
        self.remove_blank_lines(file_path)
        if file_path:
            parser = Parser()
            memory_generator = MemoryGenerator()
            binary_codes = parser.parse_file(file_path)
            memory_generator.generate_vhdl_from_memory_content(binary_codes, output_file_name)

            self.gui.show_message("Assembling completed.")
    except Exception as e:
        self.gui.show_message(f"Error during assembling: {str(e)}")
```

Metoda remove_blank_lines:

- Elimină liniile goale din fișierul specificat (file_path) și salvează rezultatul înapoi în același fișier.

```
def remove_blank_lines(self, file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    non_blank_lines = [line.strip().upper() for line in lines if line.strip()]

    with open(file_path, 'w') as file:
        file.write('\n'.join(non_blank_lines) + '\n')
```

### 5.4 AssemblerGUI

Metoda __init__:

- Configurează aspectul ferestrei și poziția pe ecran.
- Creează etichete, câmpuri de intrare și butoane pentru selectarea fișierului de intrare și specificarea numelui fișierului de ieșire.
- Setează stilul pentru elementele GUI.

Metoda browse_file:

- Deschide o fereastră de dialog pentru a permite utilizatorului să selecteze un fișier.

Metoda show_message:

- Afișează o fereastră de mesaj cu un mesaj dat.

# 6 Testare

### 6.1 Overview

Pentru a testa proiectul am ales sa testez componentele separate (Unit testing)

### 6.2 Testare Parser

```
class TestParser(unittest.TestCase):

    def test_parse_instruction(self):
        parser = Parser()

        test_cases = [
            {"instruction": "ADD R2,R1,R4", "expected_binary": "00000010001000010100000000000000"},
            {"instruction": "BZ R3,8", "expected_binary": "01100000000000110000000000001000"},
        ]
        for case in test_cases:
            binary_code = parser.parse_instruction(case["instruction"], line: 1)

            self.assertEqual(binary_code, case["expected_binary"])
```

```
Ran 1 test in 0.002s

OK
```

## 6.3 Testare MemoryGenerator

```
        memory_content = [
            "00000000000000000000000000000001"
        ]



        expected_vhdl_code = """library IEEE;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_1164.ALL;

entity Instruction_Memory is
    Port (
        PC : in STD_LOGIC_VECTOR (7 downto 0);
        Instruction : out STD_LOGIC_VECTOR (31 downto 0));
end Instruction_Memory;

architecture Behavioral of Instruction_Memory is

type rom_mem is array(0 to 255) of std_logic_vector(31 downto 0);
signal rom: rom_mem:=(
    0 => B"00000000000000000000000000000001",
    others => x"00000000"
    );

begin

Instruction<=rom(conv_integer(PC));

end Behavioral;
```

```
  generated_vhdl_code = memory_generator.generate_vhdl_from_memory_content(memory_content,

  self.assertEqual(generated_vhdl_code, expected_vhdl_code)
```

```
Ran 1 test in 0.003s

OK
```

# 7 Bibliografie


- Lucrare laborator **CAPITOLUL 8 IMPLEMENTAREA UNUI PROCESOR RISC**
- https://www.spiceworks.com/tech/tech-general/articles/risc-vs-cisc/
- https://docs.python.org/3/library/unittest.html