Simulator de UCP
in arhitectura pipeline

Grumazescu Silviu

Cuprins

1. Introducere

1.1 Context

Scopul acestui proiect este simularea executiei instructiunilor intr-o unitate centrala de procesare de tip pipeline. Intr-o arhitectura pipeline, etapele de executie a unei instructiuni sunt impartite pe mai multe cicluri de ceas, motivul fiind dorinta de a micsora lungimea drumului critic iar prin urmare permitandu-ne sa avem o frecventa de ceas mai mare. Pentru o intelegere mai buna a functionarii unei astfel de arhitecturi, acest proiect isi propune sa simuleze executia unor instructiuni introduse de utilizator, ilustrand la fiecare ciclu de ceas starea registrilor in fiecare etapa de procesare.

1.2 Specificatii

Unitatea centrala de procesare model aleasa pentru acest proiect este implementarea pipeline a arhitecturii MIPS pe 32 de biti. Partea functionala a aplicatiei va fi realizata in limbajul Java iar interfata va fi realizata folosindu-ne de framework-ul Swing.

Instructiunile rulate vor fi introduse de utilizator in formatul specific instructiunilor MIPS. Memoria de date sau memoria de instructiuni nu va fi limitata.

Instructiunile suportate de simulator vor fi de 3 tipuri:

- Instructiuni tip R: operatiile sunt registrii si rezultatul se stocheaza intr-un registru.

Format instructiune	RTL Abstract	Descriere
add $rd, $rs, $rt	RF[rd] <- RF[rs] + RF[rt]	Adunare
sub $rd, $rs, $rt	RF[rd] <- RF[rs] – RF[rt]	Scadere
and $rd, $rs, $rt	RF[rd] <- RF[rs] & RF[rt]	AND logic
or $rd, $rs, $rt	RF[rd] <- RF[rs] | RF[rt]	OR logic
xor $rd, $rs, $rt	RF[rd] <- RF[rs] ^ RF[rt]	XOR logic
sll $rd, $rs, sa	RF[rd] <- RF[rt] << sa	Shiftare logica stanga
srl $rd, $rs, sa	RF[rd] <- RF[rt] >> sa	Shiftare logica dreapta
slt $rd, $rs, $rt	If(RF[rs] < RF[rt]) then RF[rd] <- 1 else RF[rd] <- 0	Set less than
jr $rs	PC <- RF[rs]	Jump register

- Instructiuni de tip I: un operator registru si un operator imediat. Rezultatul se stocheaza intr-un registru

Format instructiune	RTL Abstract	Descriere
lw $rt, imm($rs)	RF[rt] <- M[RF[rs] + S_Ext(imm)]	Load word
sw $rt, imm($rs)	M[RF[rs] + S_Ext(imm)] <- RF[rt]	Store word
beq $rs, $rt, imm	If(RF[rs] == RF[rt]) then PC <- PC + 4 + S_Ext(imm) <<2	Branch on equal
bgez $rs, imm	If(RF[rs] >= 0) then PC <- PC + 4 + S_Ext(imm) <<2	Branch on greater or equal
addi $rt, $rs, imm	RF[rt] <- RF[rs] + S_Ext(imm)	Adunare cu imediat

- Instructiuni de tip J: Instructiuni de salt neconditionat

Format instructiune	RTL Abstract	Descriere
j target_address	PC <- PC[31:28] || target_address || 00	Jump

1.3 Obiective

Utilizatorul va fi capabil sa introduca instructiuni in limbaj de asamblare care vor fi transformate automat in cod masina, iar la pornirea simularii va putea vizualiza in interfata grafica starea tuturor registrilor la fiecare ciclu de ceas. Aplicatia va permite modificarea de catre utilizator a memoriei de date inaintea simularii si vizualizarea modificarii datelor pe parcursul rularii ei.

2. Studiu bibliografic

Varianta Pipeline a procesorului MIPS a aparut datorita dorintei de a miscora lungimea drumului critic pentru a putea avea o frecventa de ceas mai mare. Astfel, instructiunile sunt impartite in mai multe etape, etajele care sectioneaza caile de date transferand informatia cu ajutorul unor registri intermediari.

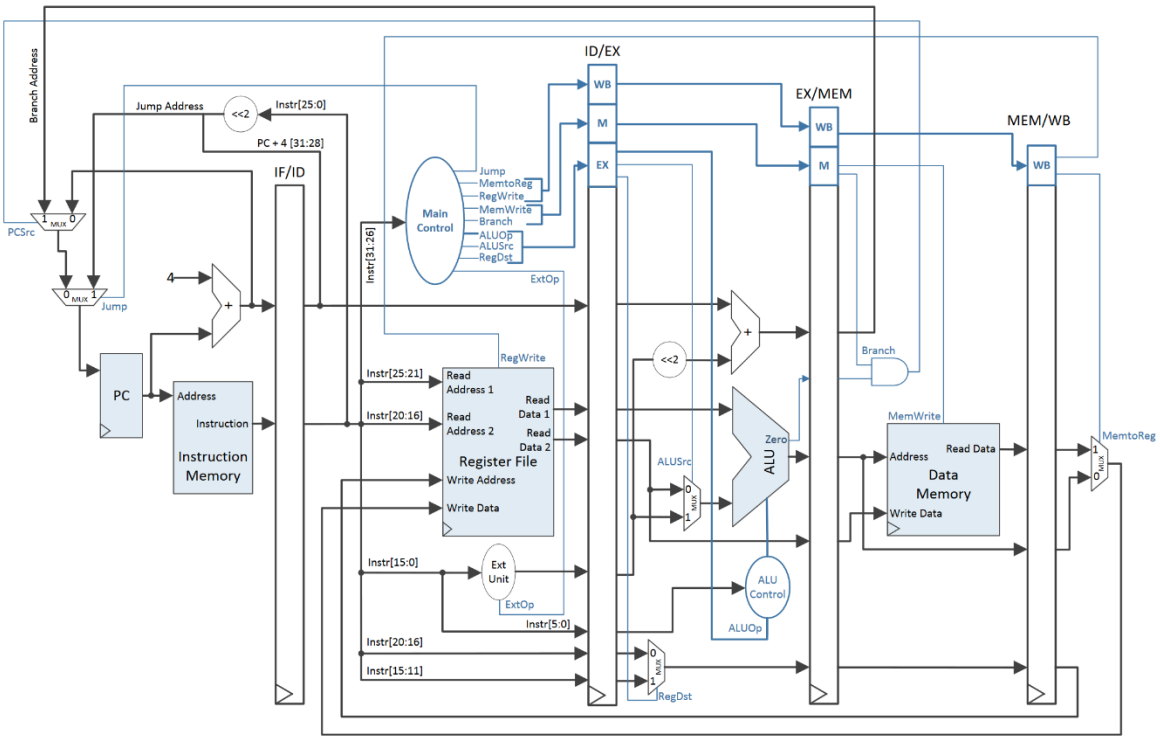

Figura 1. Diagrama procesorului MIPS Pipeline

Semnalele de control

- PCSrc : selecteaza adresa de salt
- Jump: determina incarcarea Jump Address in PC counter
- MemToReg: determina intrarea lui WriteData
- RegWrite: determina daca memoria TF scrie sau nu
- MemWrite: determina daca memoria de date scrie sau citeste
- Branch: folosit de instructiunile de salt conditionat
- ALUOp: selecteaza operatia facuta de ALU
- ALUSrc: selecteaza cel de al doilea operand ALU
- RegDst: selecteaza fie registrul $rt fie $rd pentru adresa de scriere in memoria RF
- Zero: returnat de ALU atunci cand operatia da ca rezultat 0
- ExtOp: decide daca imediatul este extins cu semn sau nu

Operatiile realizate de fiecare etaj:

1. Instruction Fetch
    - Instructiunea pe 26 de biti de la adresa data de program counter este transferata din memoria de instructiuni in registrul IF/ID.
    - Program counterul este fie incrementat cu 4 fie suprascris de o instructiune de salt.
    - Valoarea PC+4 este memorata in registrul IF/ID pentru a putea fi folosita ulterior
2. Instruction Decode & Register read
    - Instructiunea din IF/ID este decodificata, in functie de tipul instructiunii unitatea de control scriind in registrul ID/EX semnalele de control corespunzatoare
    - Registrii $rs, $rt de la adresele citite din registrul IF/ID sunt cititi apoi scrisi in registrul ID/EX
    - Imediatul este extins cu semn daca ExtOp este activ, iar apoi scris in registrul ID/EX
    - Daca semnalul RegWrite este activ atunci la adresa WriteRegister se va scrie WriteData
3. Execute operation
    - Unitatea ALU proceseaza un rezultat, operatia find determinata de semnalul de control ALUOp
    - Semnalul ALUSrc determina sursa celui de al doilea operand ALU
    - Se calculeaza (PC+4) << 2 pentru a putea fi utilizat ulterior
    - Se scriu in registrul EX/MEM:
        - Rezultatul ALU
        - Semnalele de control Branch, MemWrite, PCSrc, Zero si MemToReg
        - (PC + 4) << 2
        - $rt sau $rd in functie de RegDst
4. Memory
    - Memoria de date fie scrie fie citeste in functie de semnalul MemWrite
    - Semnalul PCSrc este activat daca semnalul Branch si Zero sunt activate
    - Se scriu in registrul MEM/WB:
        - Valoarea citita din memorie
        - Rezultatul ALU din EX/MEM
        - Semnalele de control MemToReg si PCSrc
        - WriteAddress din EX/MEM
5. WriteBack
    - Selecteaza in functie de MemToReg valoarea lui WriteData din memoria RF, fie rezultatul ALU fie valoarea extrasa din memoria de date.

3. Analiza

In acest capitol se vor analiza detaliile de implementarea a proiectului cat si cazurile de utilizare posibile care vor fi implementate.

3.1 Propunere de proiect:

Pentru a putea realiza o aplicatie cu adevarat utila pentru utilizatorii care doresc ca simuleze o secventa de instructiuni, principalul obiectiv al proiectului va fi reprezentarea cat mai realista a fluxului de date in fiecare etapa de executie. Acest fapt ne constrange sa avem un design complex care nu va urmari doar obtinerea unui rezultat ci si crearea unui sistem bine structurat de transmitere a datelor intre componente. Acestea fiind spuse proiectul va avea urmtoarele functionalitati:

- O interfata care va permite introducerea de catre utilizator a unor serii de instructiuni in limbaj de asamblare MIPS, si care va afisa starea atat a registrilor si a memoriei cat si a valorilor calculate asincron de catre fiecare etaj in parte.
- Un sistem ce poate transforma instructiunile din limbaj de asamblare in cod masina.
- O arhitectura a aplicatiei care va copia structura hardware a unui procesor MIPS.

3.2 Analiza functionalitatilor

Interfata cu utililizatorul:

Interfata va fi realizata folosindu-ne de framework-ul Swing din Java, ce beneficiaza de o serie de elemente de interfata necesare pentru proiectul nostru cum ar fi butoane, casete de text si tabele.

Utilizatorul va fi capabil sa adauge propriile instructiuni pentru a fi executate de simulator, astfel primii pasi pentru utilizarea aplicatiei vor fi introducerea intr-o caseta text a instructiunilor si validarea acestora printr-un buton care fie va incarca valorile, fie va returna un mesaj de eroare.

Al doilea pas va fi pornirea simularii, utilizatorul fiind capabil sa vada informatiile din registrii si memorie la fiecare impuls de ceas, pasul simularii fiind incrementat printr-un buton de step.

Cazurile de utilizare sunt reprezentate in Figura 2.

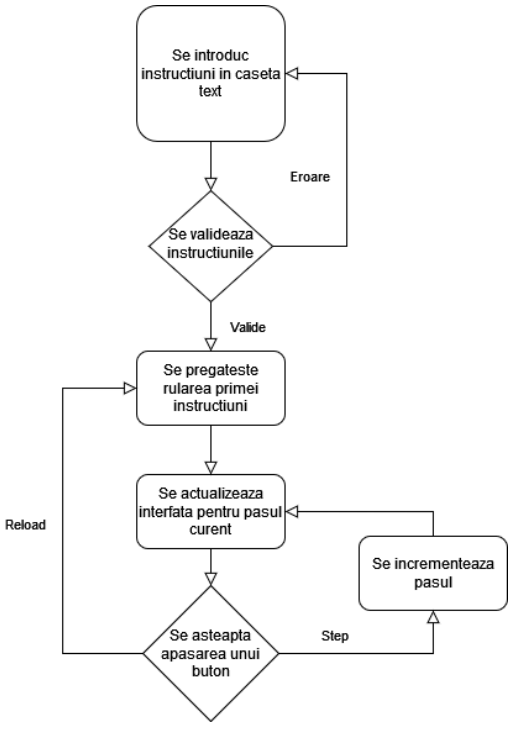

Figura 2. Diagrama cazurilor de utilizare

Parsarea datelor:

Pentru transformarea datelor din limbaj de asamblare in cod masina, ne vom folosi de expresii regex si de alte unelte oferite de java pentru manipularea string-urilor.

O expresie regex este un sablon cu ajutorul caruia engine-ul regex incearca sa potriveasca un text de intrare. Acesta poate fi folosit pentru a recunoaste o instructiune dar si pentru a extrage din aceasta operanzii. In java, expresiile regex sunt implementate in libraria java.util.regex.

3.3 Structura proiectului:

Pentru a putea reprezenta cat mai realist functionarea unui procesor MIPS, proiectul va fi structurat reprezentand prin clase majoritatea componentelor prezente in Figura 1.

Deoarece in arhitectura MIPS avem atat procese care se executa sincron cat si asincron, pentru a reprezenta cat mai realist fluxul de date am impartit operatiile executate in doua etape:

1. Etapa de pre-calculare

   In aceasta etapa se realizeaza toate operatiile asincrone care in realitate se intampla imediat dupa modificarea valorilor din registrii pipeline. Aceasta etapa se va realiza apeland functia compute() a fiecarui etaj pipeline, rezultatele stocandu-se in niste registrii intermediari. Valorile sunt puse in asteptare pentru a putea fi stocate ulterior sau folosite de alte componente asincrone din alte etaje.

2. Etape de clock
   In aceasta etapa se vor realiza operatiile sincrone, cum ar fi scrierea in Register File sau memorie, cat si scrierea registrilor pipeline care duc informatia la etajul urmator.

Asadar, la apasarea butonului de step de catre utilizator, simulatorul prima data va realiza operatiile sincrone pentru un ciclu de ceas (clock()) iar apoi va pre-calcula valorile produse de componentele asincrone in functie de datele stocate in registrii pipeline. Aceasta arhitectura a aplicatiei ne va permite sa vizualizam pe interfata grafica atat valorile setate in registrii la clock-ul anterior, cat si rezultatele care asteapta sa fie scrise la urmatorul ciclu de ceas.

3.4 Planificarea dezvoltarii

7 noiembrie – 21 noiembrie

Prototipizarea claselor specifice MIPS si a interfetei

21 noiembrie – 5 decembrie

Dezvoltarea expresiilor regex pentru parsarea codului de asamblare si testarea sa

5 noiembrie – 19 decembrie

Finalizarea implementarii claselor si a interfetei

19 decembrie – 2 ianuarie

Finalizarea aplicatiei in ansamblu si testarea acesteia

9 ianuarie

Predarea proiectului

4. Design

Register

Un element esential in construirea acestei arhitecturi va fi clasa Register reprezentand un registru pipeline. Intr-un registru pipeline valorile de la intrare vor fi scrise doar la semnalul de clock. Acest lucru va fi simulat prin stocarea informatiilor in doua Liste: waitingValues si Values. In etapa de **pre-calculare** etajele vor stoca valorile calculate in waitingValues, iar in etapa de **clock** elementele din *waitingValues* vor fi transferate in *values.*

Astfel, toate etajele vor procesa informatia cu valori din registre scrise la clock-ul anterior (values), evitand posibilitatea suprascrierii acestora inainte ca toate celelalte etaje sa fi procesat informatia din el.

Calea datelor este reprezentata in figura 4.

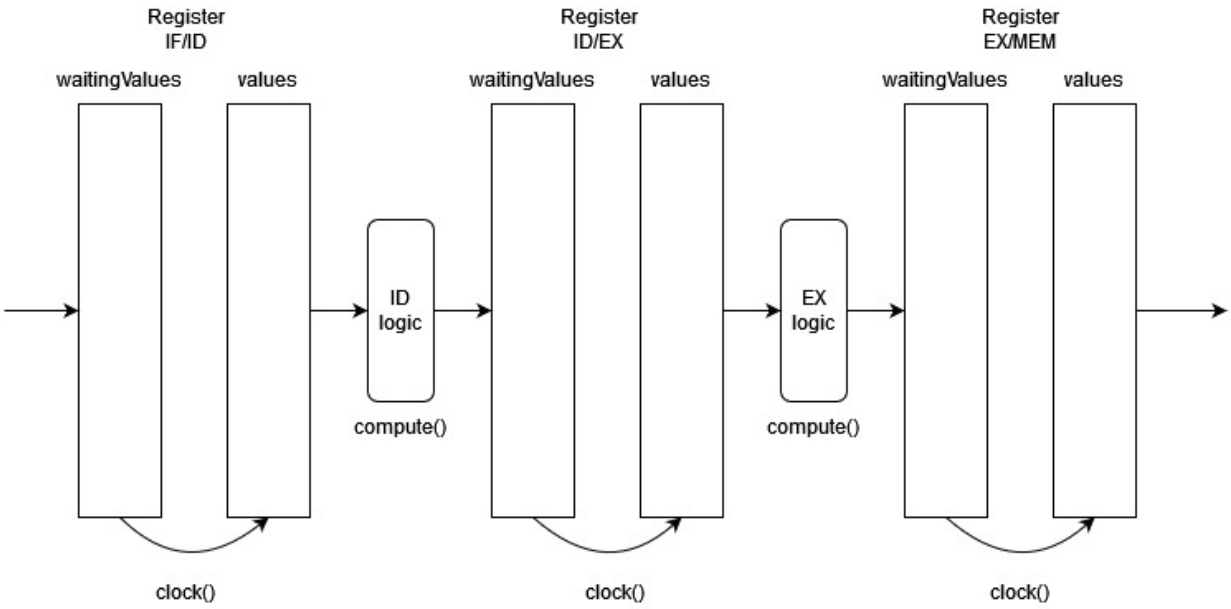

Figura 4. Calea datelor prin registrii

Design-ul componentelor hardware

Pentru reprezentarea componentelor hardware, clasele vor fi structurate ierarhic, clasa MIPS continand toate etajele, iar fiecare etaj continand componentele sale specifice. Diagram UML a claselor este reprezentata in figura 3.

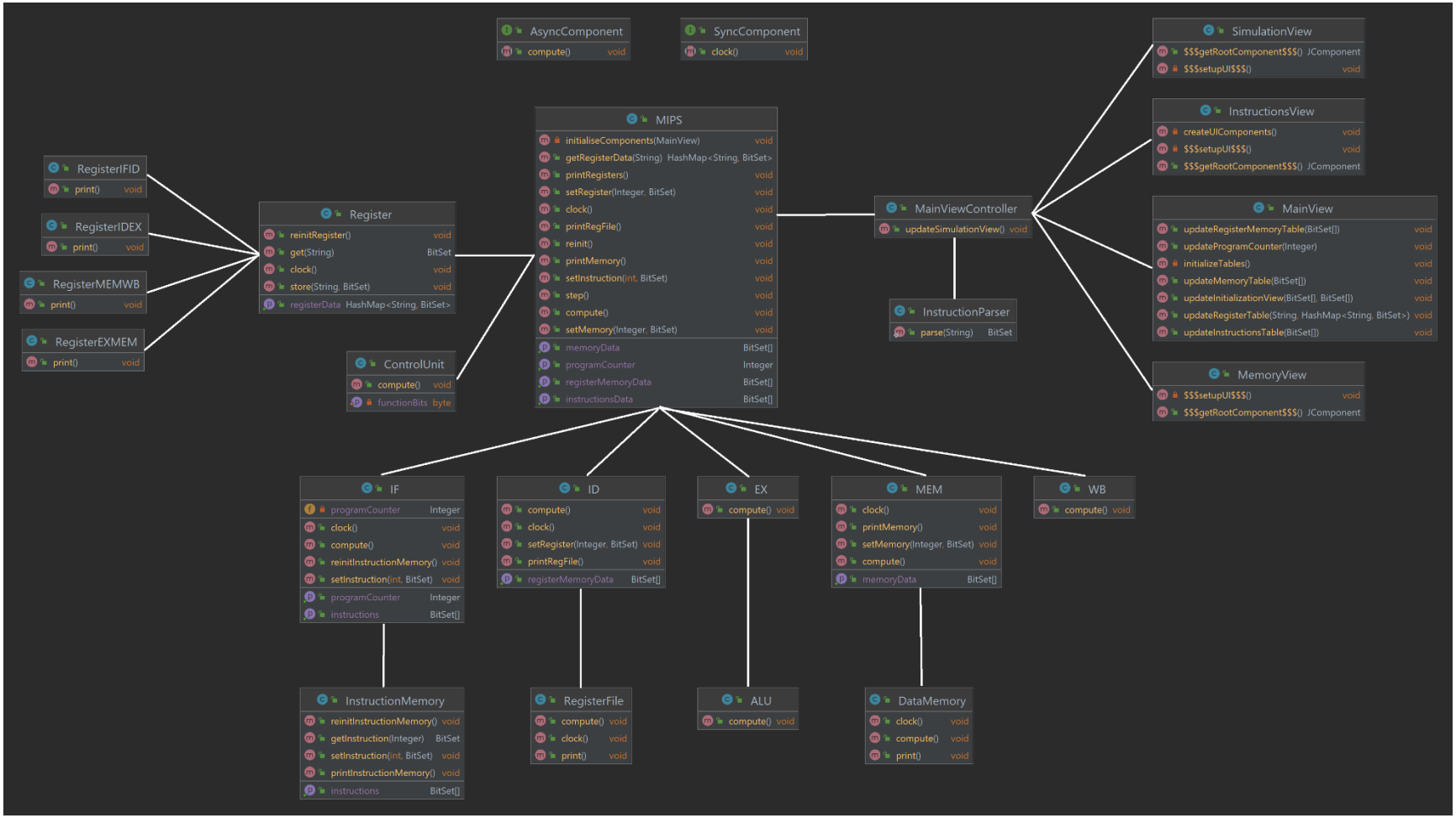

Figura 3. Diagrama UML a claselor

Pentru a marca tipul de componenta, se vor defini doua interfete:

- AsyncComponent – clasa implementeaza functia compute() pentru calculul operatiilor asincrone
- SyncComponent – clasa implementeaza functia clock() pentru calculul operatiilor ce tin de impulsul de ceas.

Stabilirea acestei ierarhii si marcarea claselor cu interfetele de mai sus ne va permite sa apelam din parinte in fiu functiile clock() si compute(), in etapa operatiilor sincrone clasa MIPS apeland functia clock() a tuturor etajelor, iar etajele apeland functia mai departe pentru toate componentele lor.

5. Implementare

Manipularea datelor pe biti

Pentru a reprezenta cat mai realist functionarea unui MIPS real, este important felul in care stocam si procesam informatia binara. Pentru a stoca un set de biti am folosit clasa BitSet din libraria „java.util”.

BitSet este o clasa care stocheaza informatia binara ca o lista de valori boolene, indexul reprezentant positia bitului iar valoarea reprezentand „1” sau „0”. Metoda get() a acestei clase ne permite sa extrage usor parti dintr-un numar binar, foarte utila in cadrul clasei ID in care decodificam instructiunea.

Deoarece clasa BitSet nu ne ofera toate metodele necesare, am implementat clasa BitUtility ce cuprinde mai multe metode utilitare pentru conversia informatiei:

- BinToInt(BitSet value) – Transforma un obiect BitSet intr-un Integer;
- BinToIntSigned(BitSet value) – Transforma un obiect BitSet intr-un Integer considerand numarul in complement fata de 2
- IntToBin(Integer value) – Transforma un Integer intr-un obiect BitSet
- StringToBin(String number) – Transforma un String intr-un obiect BitSet

Implementarea partii functionale a MIPS-ului

Clasele ce fac parte din implementarea MIPS-ului propriu-zis se pot observa in figura 3, acestea urmand design-pattern-ul „Composite”.

Clasa MIPS este compusa din alte 5 clase ce reprezinta etajele sale, comunicarea dintre ele fiind realizata prin intermediul registrilor. Mai specific, la initializarea unui etaj, transmitem prin intermediul constructorului referintele catre locatiile de memorie unde sunt stocate datele necesare functionarii acestuia.

```
private void initialiseComponents(MainView mainView) {
    registerIFID = new RegisterIFID(mainView);
    registerIDEX = new RegisterIDEX(mainView);
    registerEXMEM = new RegisterEXMEM(mainView);
    registerMEMWB = new RegisterMEMWB(mainView);

    controlUnit = new ControlUnit(registerIFID.get(RegisterIFID.INSTRUCTION));

    writeBack = new WB(registerMEMWB);
    memory = new MEM(registerEXMEM, registerMEMWB, mainView);
    executeOperation = new EX(registerIDEX, registerEXMEM);
    instructionDecode = new ID(registerIFID, registerMEMWB, registerIDEX, writeBack.writeData, controlUnit.ControlSignals);
    instructionFetch = new IF(registerIFID, registerEXMEM, controlUnit.ControlSignals, memory.PCSrc);
}
```

Figura 5. Initializare componente

Registrii

Valorile din registrii sunt salvate sub forma unui HashMap, cheia fiind un String cu denumirea valorii iar valoarea fiind un obiect de tip BitSet. Aceasta structurare ne avantajeaza deoarece in primul rand cheia din HashMap poate fi folosita de interfata pentru a adauga o eticheta la semnalul respectiv, iar in al doilea rand putem transfera printr-o simpla bucla for toate informatiile din „waitingValues” in „values”.

Interfata

Pentru dezvoltarea interfetei am folosit elementele de GUI din java.swing. Implementarea acesteia a urmat design pattern-ul Model-View-Controller, astfel clasa MainViewController facand legatura dintre clasa MIPS (modelul) si clasele ce inglobeaza componentele grafice.

Layout-ul grafic ales pentru interfata este CardLayout, acesta permitandu-ne sa avem mai multe panouri prin care putem tranzitiona folosindu-ne de butoane.

Cele 3 panouri pe care le poate vizualiza utilizatorul sunt urmatoarele:

- InstructionsView – Fereastra de initializare a simularii in care se pot seta instructiunile, se poate modifica memoria de date si registrele. (Figura 6)

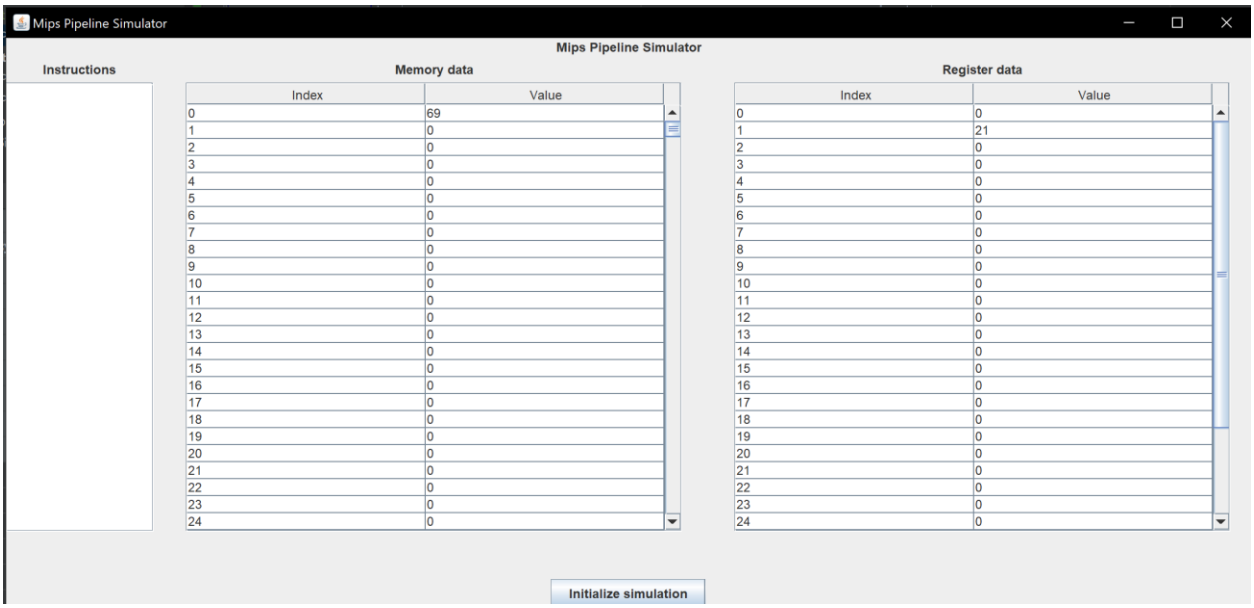

Figura 6. InstructionsView

- SimulationView – Fereastra afisata in momentul pornirii simularii. Afiseaza pe ecran instructiunile din InstructionMemory si valorile din cei 4 registrii pipeline.

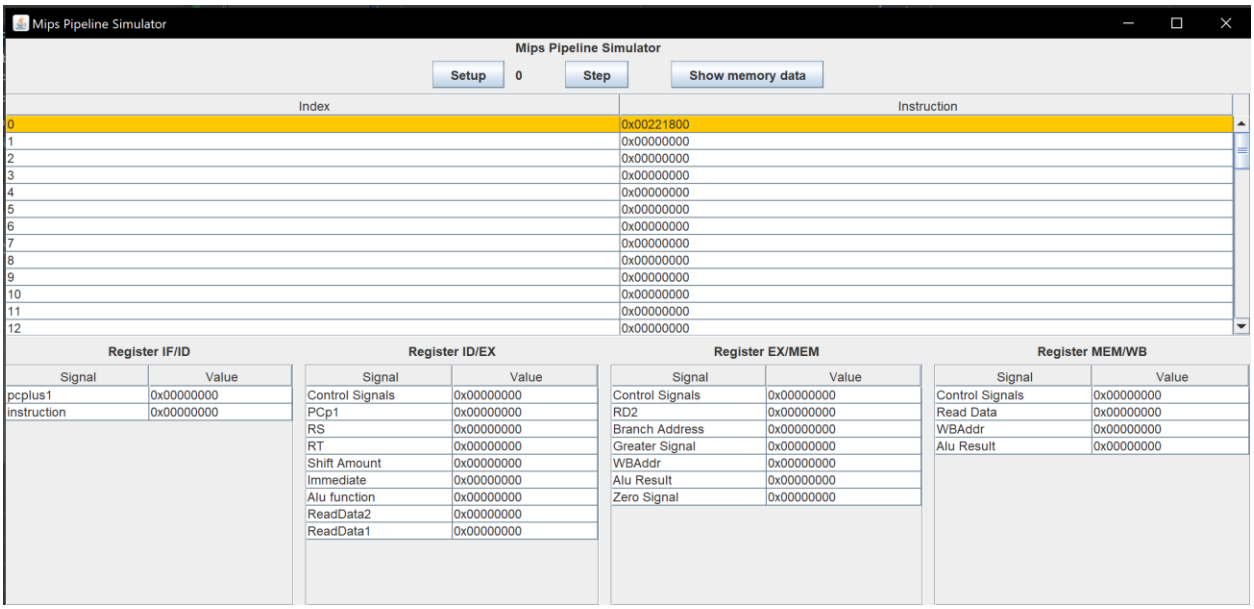

Figura 7. Simulation View

- MemoryView – Odata pornita simularea utilizatorul poate vizualiza in aceasta fereastra valorile din memoria de date si din Register File.

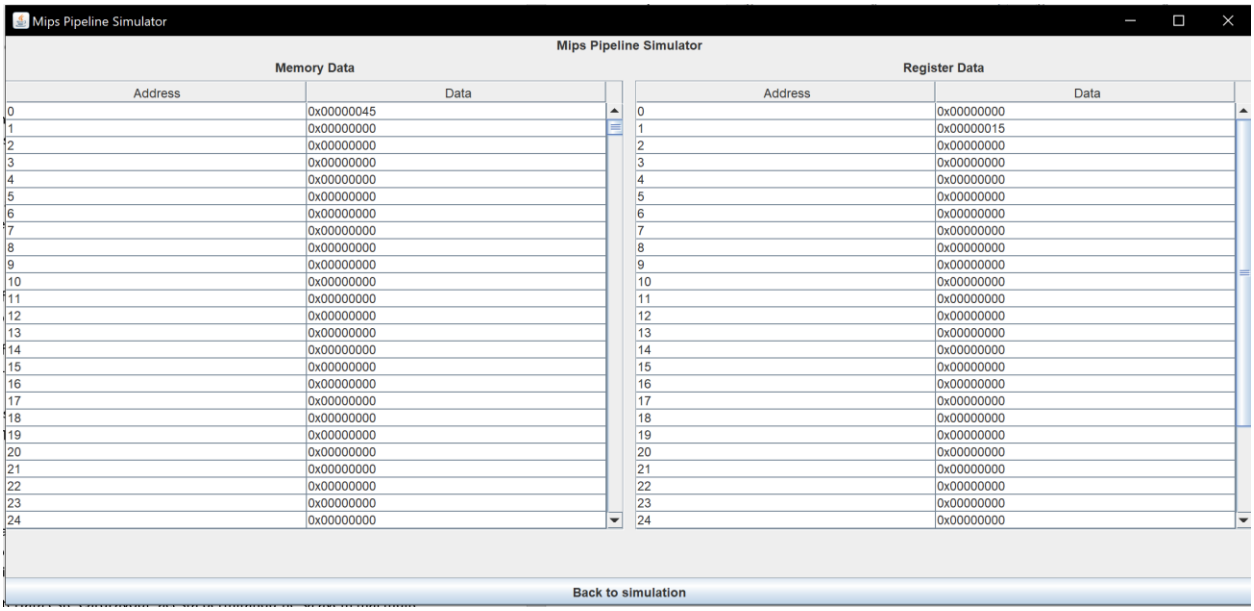

Figura 8. Memory View

Parsarea instructiunilor

Deoarece utilizatorul introduce in casuta de text instructiuni in limbaj de asamblare, a fost necesara implementarea unui parser care transforma textul introdus de acesta in obiecte de tip BitSet.

Clasa InstructionParser are o singura metoda statica parse(String instruction) capabila de a parsa instructiuni, implementarea luand in considerare atat numere pozitive cat si negative pentru immediat sau adresa de salt.

Ghid utilizare:

Parserul de instructiuni se asteapta ca instructiunile sa fie intr-un anumit format pentru a le putea interpreta corect.

1. Instructiunile se vor scrie cate una pe linii separate.
2. Formatul instructiunilor va fi urmatorul (acoladele nu trebuie scrise ci doar marcheaza campurile):

   ADD: add {rs} {rt} {rd}
   SUB: sub {rs} {rt} {rd}
   AND: and {rs} {rt} {rd}
   OR: or {rs} {rt} {rd}
   XOR: xor {rs} {rt} {rd}
   SLL: sll {rs} {rd} {shiftAmount}
   SRL: srl {rs} {rd} {shiftAmount}
   SLT: slt {rs} {rt} {rd}
   LW: lw {rs} {rt} {immediate}
   SW: sw {rs} {rt} {immediate}
   ADDI: addi {rs} {rt} {immediate}
   J: j {jumpAddress}
   BEQ: beq {rs} {rt} {branchOffset}

3. rs, rt, rd, shiftAmount, jumpAddress sunt numere pozitive intregi iar immediate si branchOffset sunt intregi cu semn (se va pune simbolul „-„ in fata numarului in cazul numerelor negative)

Exemplu de program:

0. lw 0 1 0
1. lw 0 2 1
2. add 0 0 0
3. add 0 0 0
4. add 0 0 0
5. beq 1 2 12
6. add 0 0 0
7. add 0 0 0
8. add 0 0 0
9. add 0 0 0

10. addi 4 4 5
11. add 0 0 0
12. add 0 0 0
13. addi 1 1 1
14. add 0 0 0
15. add 0 0 0
16. j 5
17. add 0 0 0
18. add 0 0 0

O bucla repetitiva care aduna repetat valoarea 5 in registrul 4 de cate ori se specifica la adresa de memorie 1.

6. Testare

Pentru testarea simulatorului vom incarca un program simplu ce va lua doua numere din memoria de date, le va aduna apoi le va stoca inapoi in memorie:

1. lw $1, 1($0)
2. lw $2, 2($0)
3. add $0, $0, $0
4. add $0, $0, $0
5. add $0, $0, $0
6. add $3, $1, $2
7. add $0, $0, $0
8. add $0, $0, $0
9. add $0, $0, $0
10. sw $3, 3($0)

Datorita hazardurior de date, suntem nevoiti sa inseram 3 instructiuni NoOp inaintea de instructiunea de add si sw, pentru a astepta ca datele sa fie stocate in registrii.

In memorie vom stoca la adresa 1 valoarea 20, iar la adresa 2 valoarea 5 (Figura 9):

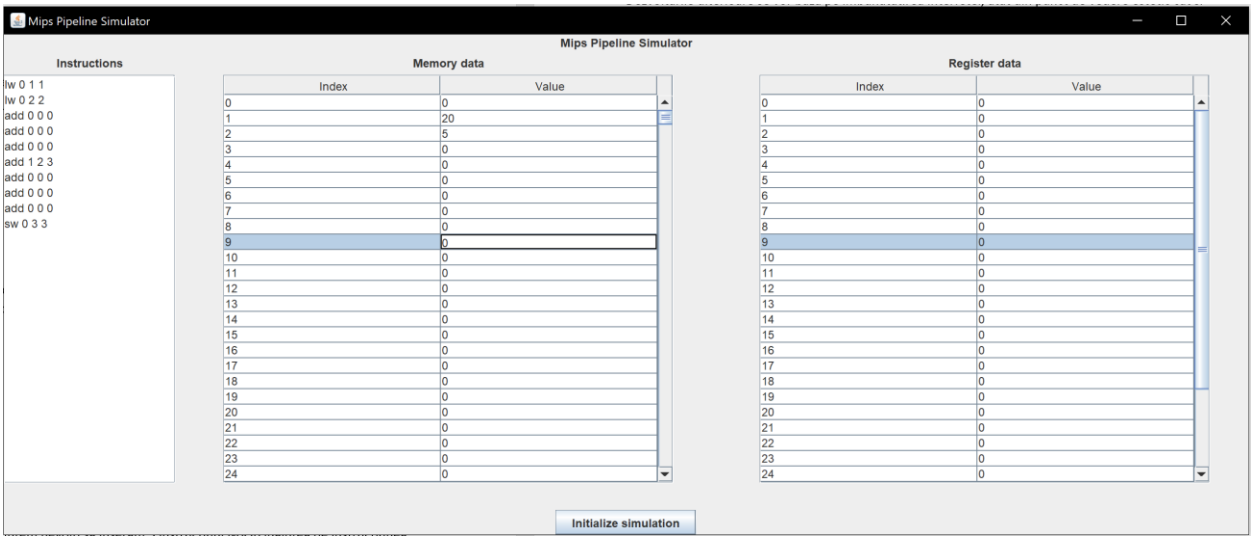

Figura 9. Testing 1

Dupa apasarea butonului „Initialize simulation”, vom apasa butonul de step, iar la pasul 13 putem observa ca la adresa 3 din memorie s-a stocat corect valoarea 25 (Figura 10, Figura 11).

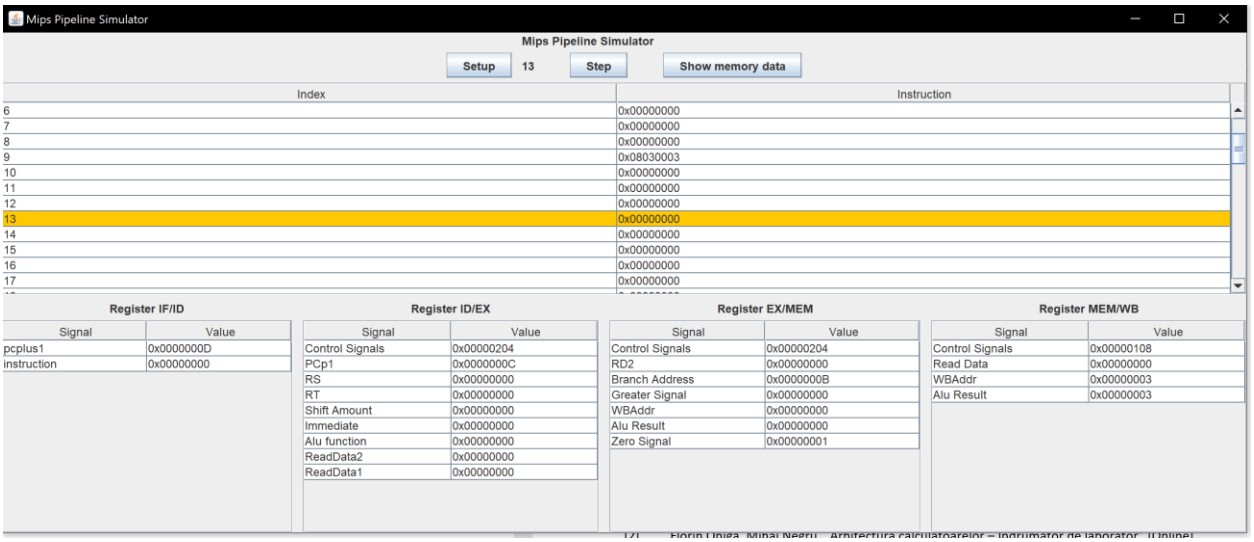

Figura 10. Testing 2

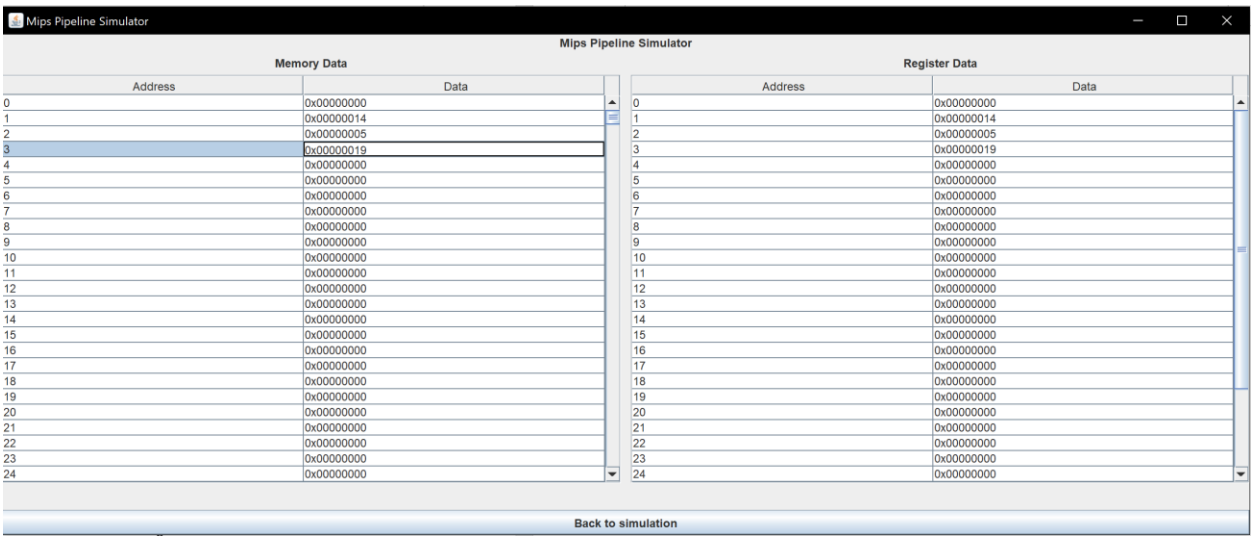

Figura 11. Testing 3

7. Concluzie

Dezvoltarile ulterioare se vor baza pe imbunatatirea interfetei, atat din punct de vedere estetic cat si functional, cum ar fi afisarea semnalelor de control. Deasemenea, parserul de instructiuni necesita imbunatatiri in privinta alertarii utilizatorului in cazul unor greseli de sintaxa.

Scopul acestui proiect a fost implementarea unui simulator de MIPS Pipeline, accentul punandu-se in special pe imitarea cat mai verosimila a unui procesor MIPS real, astfel oferind un material educational cat mai relevant pentru utilizator. Implementarea acestui proiect m-a facut sa am o intelegere mult mai buna atat asupra arhitecturii MIPS, cat si a limbajului Java.

8. Bibliografie

[1] Florin Oniga, „Arhitectura calculatoarelor” [Course]

[2] Florin Oniga, Mihai Negru, „Arhitectura calculatoarelor – Indrumator de laborator” [Online].

Disponibil: https://users.utcluj.ro/~onigaf/files/teaching/AC/AC_indrumator_laborator.pdf