UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Structura sistemelor de calcul

Proiect

Simulator de UCP în arhitectură pipeline

Realizat de:

Ducai David-Alex, grupa 30233

Cuprins

1. Propunere de proiect

Obiectivul proiectului este realizarea unui simulator funcțional de unitate centrala de procesare (UCP) în arhitectură pipeline. În simulator vor fi implementate 15 instrucțiuni care pot fi realizate de către UCP.

Simulatorul va avea o interfață grafică, care îi va permite utilizatorului să creeze un program folosind tipurile de instrucțiuni implementate și să vadă execuția acestuia. La fiecare moment de timp, simulatorul va afișa calea de date și conținutul fiecărei componente implementate în UCP. Astfel, simulatorul va facilita urmărirea execuției de către procesor a unui program.

Simulatorul va fi implementat utilizând limbajul Java, deoarece acest limbaj oferă suport și facilitează implementarea interfețelor grafice.

2. Studiu bibliografic

Tipul de UCP aleasă pentru simulare este MIPS în arhitectură pipeline, pe 32 de biți. Aici vor fi descrise părțile relevante care vor fi implementate și reprezentate în simulator.

Microprocesorul MIPS este alcătuit din 5 etaje: IF (instruction fetch), ID (instruction decode), EX (execute address), MEM (memory access) și WB (write back).

În etajul IF se calculează adresa instrucțiunii care trebuie executată și se citește din memorie.

În etajul ID se decodifică instrucțiunea adusă din etajul anterior și se citesc operanzii pentru realizarea operației. Etajul ID conține un bloc de regiștri (register file / RF), care conține 32 de regiștri, numerotați de la 0 la 31. Operanzii pot fi citiți din regiștri sau chiar din instrucțiunea adusă din etajul IF în cazul operațiilor cu un operand de tip valoare imediată.

Etajul EX execută operația asupra operanzilor aduși din etajul ID în cazul instrucțiunilor artimetico-logice sau adresa la care se accesează memoria de date din etajul MEM în cazul operațiilor de load și store.

Etajul MEM conține memoria de date pentru instrucțiunile de load și store.

Etajul WB realizează scrierea rezultatului operației aritmetico-logice sau data citită din memorie (pentru operațiile load) în blocul de regiștri.

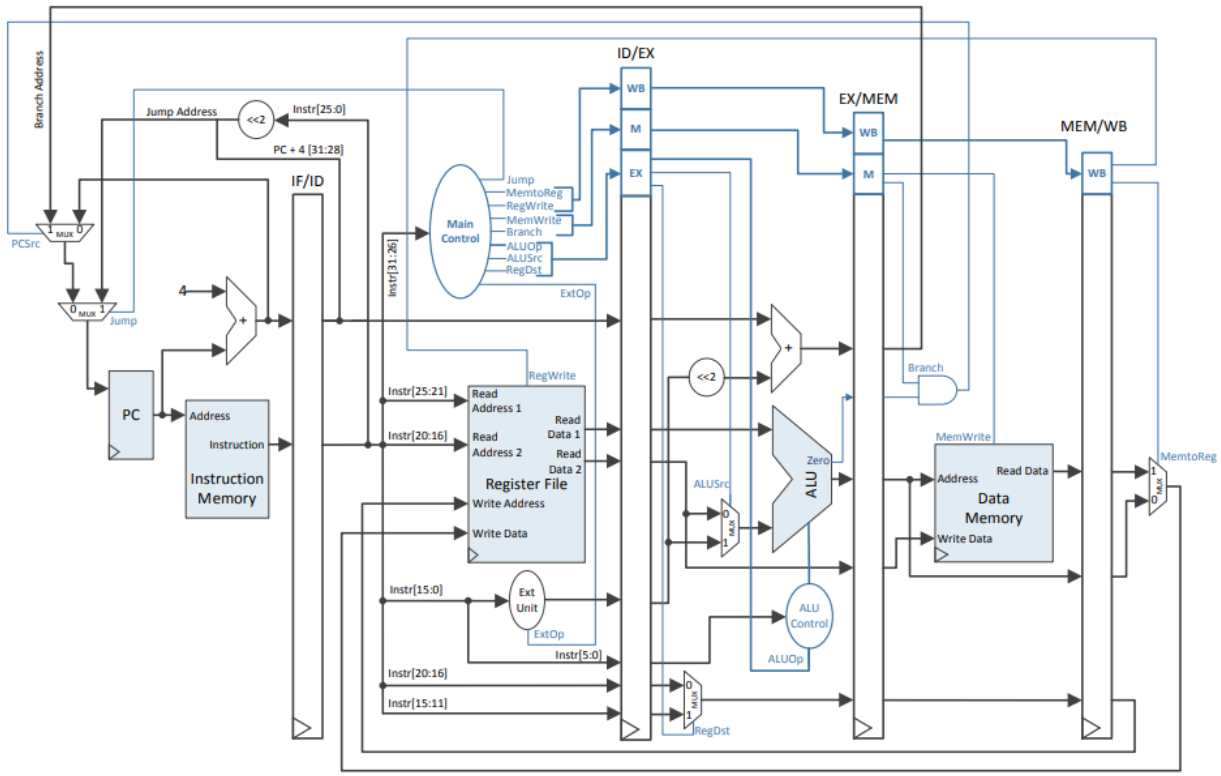


Fig. 2.1. Schema MIPS pipeline

Sursă: F. Oniga, M. Negru, „ARHITECTURA CALCULATOARELOR Îndrumător de laborator”

Microprocesorul MIPS utilizează setul de instrucțiuni RISC. Instrucțiunile executate de UCP sunt formate din 32 de biți și sunt grupate în 3 tipuri principale:

- Instrucțiuni de tip R (registru) – operanzii sunt regiștri din blocul de regiștri al etajului ID
- Instrucțiuni de tip I (imediat) – doi operanzi sunt regiștri din etajul ID, unul este o valoare imediată specificată în instrucțiune
- Instrucțiuni de tip J (jump/salt necondiționat) – un singur operand utilizat pentru calculul adresei de salt necondiționat

Instrucțiunile de tip R sunt în principal operații aritmetico-logice (adunare, scădere, shiftare, etc.). Biții intrucțiunii de tip R sunt grupați astfel (de la bitul cel mai semnificativ):

- opcode (6 biți) – reprezintă codul instrucțiunii; pentru tipul R, acest câmp este implicit 000000.
- rs (5 biți) –registrul primului operand sursă
- rt (5 biți) - registrul celui de-al doilea operand sursă
- rd (5 biți) – registrul operandului destinație
- sa / shift amount (5 biți) – câmp utilizat pentru operații de shiftare. Specifică numărul de poziții cu care se va realiza operația.
- func (6 biți) – specifică operația aritmetico-logică realizată de instrucțiune; unic pentru fiecare instrucțiune

Instrucțiunile de tip I cuprind operații artimetice, instrucțiuni de acces la memorie și instrucțiuni de salt condiționat. Biții intrucțiunii de tip I sunt grupați astfel (de la bitul cel mai semnificativ):

- opcode (6 biți) – codul instrucțiunii, unic pentru fiecare instrucțiune
- rs (5 biți) – registrul operandului sursă

- rt (5 biți) – registrul celui de-al doilea operand sau registrul destinație
- imm (16 biți) – valoare imediată; poate reprezinta: un operand (instrucțiuni aritmetice), adresă de memorie (load/store), adresă de salt (salt necondiționat)

Instrucțiunile de tip J cuprind instrucțiuni de salt necondiționat. La întâlnirea acestui tip de instrucțiune, programul va face imediat saltul la adresa indicată de instrucțiune. Biții intrucțiunii de tip J sunt grupați astfel (de la bitul cel mai semnificativ):

- opcode (6 biți) - codul instrucțiunii, unic pentru fiecare instrucțiune
- address (26 de biți) – se adaugă la PC+4 pentru a forma adresa de salt necondiționat

În varianta de MIPS cu arhitectură pipeline se mai regăsesc 4 regiștri, numiți sugestiv, între cele 5 etaje principale menționate anterior: IF/ID, ID/EX, EX/MEM și MEM/WB. Acești regiștri rețin date pentru instrucțiunea care se excută pe un anumit etaj la un moment dat.

În simulator va fi reprezentată grafic schema generală a unui microprocesor MIPS în arhitectură pipeline, descrisă mai sus și vor fi implementate, în total, 15 instrucțiuni din toate tipurile menționate: R, I și J.

3. Analiză

3.1. Componentele hardware ale MIPS pipeline necesare implementării

În acest subcapitol vor fi numite componentele hardware cele mai importante ale celor 5 etaje ale microprocesorului MIPS, care vor fi implementate în simulator. Structurile de date folosite pentru implementarea componentelor și a instrucțiunilor sunt descrise în subcapitolul 3.3.

Etajul IF:

- PC (program counter) – calculează adresa următoarei instrucțiuni de executat
- IM (instruction memory) – memorie ROM care conține instrucțiunile miniprogramului

Etajul ID:

- RF (register file) – blocul de regiștri
- MC (main control) – codifică semnalele de control ale microprocesorului în funcție de instrucțiune

Etajul EX:

- ALU (arithmetic-logic unit) – realizează operații artmetico-logice
- ALUCtrl (arithmetic-logic unit control) – codifică semnalul operației pe care o va efectua ALU

Etajul MEM:

- MEM (data memory)

De asemenea, vor fi implementați și cei patru regiștri dintre cele 5 etaje ale microprocesorului: IF/ID, ID/EX, EX/MEM, MEM/WB.

3.2. Alegerea instrucțiunilor

Pentru simulator vor fi implementate 15 instrucțiuni de tip R, I și J, pe 32 de biți. În acest subcapitol vor fi numite și descrise în detaliu.

3.2.1. Instrucțiuni de tip R

Prezentare teoretică			
Instrucțiune	**Sintaxă**	**Descriere**	**Operație**
add	add $d, $s, $t	Adună conținutul a două registre și memorează rezultatul în al treilea registru	$d ← $s + $t PC ← PC + 4
sub	sub $d, $s, $t	Scade conținutul a două registre și memorează rezultatul în al treilea registru	$d ← $s - $t PC ← PC + 4
sll	sll $d, $t, h	Se realizează deplasarea logică la stânga a conținutului unui registru, iar rezultatul este memorat în alt registru	$d ← $t << h PC ← PC + 4
srl	srl $d, $t, h	Se realizează deplasarea logică la dreapta a conținutului unui registru, iar rezultatul este memorat în alt registru	$d ← $t >> h PC ← PC + 4
and	and $d, $s, $t	Realizează operația ȘI logic între două registre și memorează rezultatul în al treilea registru	$d ← $s & $t PC ← PC + 4
or	or $d, $s, $t	Realizează operația SAU logic între două registre și memorează rezultatul în al treilea registru	$d ← $s | $t PC ← PC + 4
sllv	sllv $d, $t, $s	Se deplasează logic la stânga valoarea dintr-un registru, cu un număr de poziții indicat de alt registru, iar rezultatul este memorat într-un al treilea registru	$d ← $t << $s PC ← PC + 4
xor	xor $d, $s, $t	Realizează operația SAU exclusiv logic între două registre, memorează rezultatul în al treilea registru	$d ← $s ^ $t PC ← PC + 4

Codificarea instrucțiunilor			
Instrucțiune	**opcode**	**func**	**Format instrucțiune**
add	000000	000000	000000 sssss ttttt ddddd 00000 000000
sub		000001	000000 sssss ttttt ddddd 00000 000001
sll		000010	000000 00000 ttttt ddddd hhhhh 000010
srl		000011	000000 00000 ttttt ddddd hhhhh 000011
and		000100	000000 ssssst tttt ddddd 00000 000100
or		000101	000000 sssss ttttt ddddd 00000 000101
sllv		000110	000000 sssss ttttt ddddd 00000 000110
xor		000111	000000 sssss ttttt ddddd 00000 000111

3.2.2. Instrucțiuni de tip I

Prezentare teoretică			
Instrucțiune	**Sintaxă**	**Descriere**	**Operație**
addi	addi $t, $s, imm	Se adună conținutul unui registru cu o valoare imediată și memorează rezultatul în alt registru	$t ← $s + imm PC ← PC + 4
lw	lw $t, offset($s)	O valoare din memorie este încărcată într-un registru	$t ← MEM[$s + offset] PC ← PC + 4;
sw	sw $t, offset($s)	Valoarea dintr-un registru este stocată în memorie la o anumită adresă	MEM[$s + offset] ← $t PC ← PC + 4

beq	beq $s, $t, offset	Salt condiționat dacă valorile din cele două registre menționate sunt egale	if $s == $t then PC ← PC + 4 + (offset << 2) else PC ← PC + 4;
andi	andi $t, $s, imm	Se realizează operația ȘI logic între un registru și o valoare imediată și se memorează rezultatul în alt registru	$t ← $s & imm PC ← PC + 4
ori	ori $t, $s, imm	Se realizează operația SAU logic între un registru și o valoare imediată și se memorează rezultatul în alt registru	$t ← $s | imm PC ← PC + 4

Codificarea instrucțiunilor		
Instrucțiune	**opcode**	**Format instrucțiune**
addi	000001	000001 sssss ttttt iiiiiiiiiiiiiiii
lw	000010	000010 sssss ttttt iiiiiiiiiiiiiiii
sw	000011	000011 sssss ttttt iiiiiiiiiiiiiiii
beq	000100	000100 sssss ttttt iiiiiiiiiiiiiiii
andi	000101	000101 sssss ttttt iiiiiiiiiiiiiiii
ori	000110	000110 sssss ttttt iiiiiiiiiiiiiiii

3.2.3. Instrucțiuni de tip J

Prezentare teoretică			
Instrucțiune	**Sintaxă**	**Descriere**	**Operație**
j	j target	Salt necondiționat la adresă absolută	PC ← ((PC +4) & 0xf0000000) | (target << 2)

Codificarea instrucțiunilor		
Instrucțiune	**opcode**	**Format instrucțiune**
j	000111	000111 iiiiiiiiiiiiiiiiiiiiiiiiii

4. Design

4.1. Structuri de date utilizate pentru implementare

- Pentru implementarea componentelor menționate anterior (memoria de date, memoria de instrucțiuni, blocul de regiștri, regiștrii intermediari), vor fi create clase simbolice
- Pentru memoria de date, memoria de instrucțiuni și blocul de regiștri se vor folosi liste de tipul ArrayList.
- Se va crea o clasă pentru instrucțiuni, care va avea ca atribut un vector de tip boolean, care va conține instrucțiunea propriu-zisă, și metode de extragere a grupărilor de biți (rs, rt, rd, imm, etc.) pentru realizarea instrucțiunilor
- Se va crea o clasă pentru ALU, care va avea o metodă pentru realizarea operațiilor aritmetico-logice

4.2. Funcționalitate: interfața simulatorului

Simulatorul va fi însoțit de o interfață grafică, care va facilita utilizarea acestuia și urmărirea execuției unui miniprogram. Fereastra aplicației va fi împărțită în mai multe secțiuni:

Schema generală este una dintre cele mai importante secțiuni ale interfeței grafice. Pe această schemă (care va fi desenată asemenător cu *Fig. 2.1.*) se va vedea calea de date pentru fiecare ciclu de ceas. Cu ajutorul unui buton (ex. Next), vom trece la următorul ciclu de ceas și vom vedea rezultate noi pe schemă.

Introducerea miniprogramului este cealaltă secțiune importantă a interfeței grafice. Este singura secțiune prin care utilizatorul poate introduce date în simulator. Utilizatorul introduce în această secțiune miniprogramul care vrea să fie executat de către simulator. Un buton de start va porni execuția simulatorului și nu îi va mai permite utilizatorului să modifice miniprogramul până la sfârșitul execuției sale.

Secțiunea de verificare a conținutului componentelor îi permite utilizatorului să vadă conținutul unor componente hardware la fiecare ciclu de ceas. Utilizatorul va putea apăsa componente pe schema miniprocesorului. Lângă schemă, se va putea vedea conținutul memoriilor sau a regiștrilor.

Secțiunea de ajutor are un rol educativ, prin explicarea acțiunilor executate de către microprocesor la fiecare ciclu de ceas.

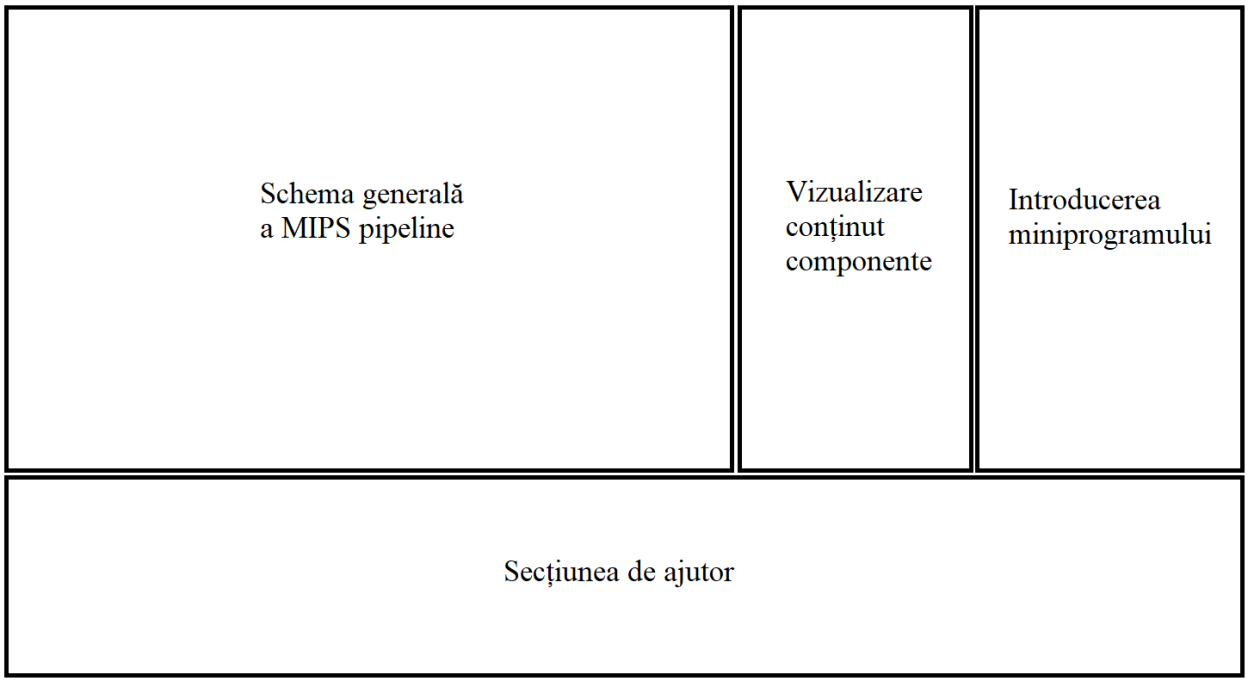


Fig. 4.1. Layout-ul ferestrei aplicației – schemă teoretică

4.3. Cazuri de utilizare

Pentru toate cazurile de utilizare prezentate, actorul este reprezentat de utilizatorul simulatorului de UCP pipeline.

4.3.1. ***Caz de utilizare**: introducerea miniprogramului*

1. Utilizatorul deschide aplicația
2. Utilizatorul introduce în secțiunea din dreapta-sus miniprogramul pe care dorește să-l simuleze, scriind instrucțiunile sub formatele descrise în capitolul 3
3. Utilizatorul apasă butonul de start pentru a porni simularea
4. Utilizatorul nu poate modifica miniprogramul pe durata execuției simulatorului

4.3.2. ***Caz de utilizare**: vizualizarea execuției miniprogramului*

1. Utilizatorul introduce miniprogramul în aplicație și pornește simularea (cazul de utilizare precedent)
2. Utilizatorul poate vedea, pe schema generală, execuția miniprogramului la ciclul de ceas curent
3. Utilizatorul poate apăsa pe butonul “Următorul” pentru a trece la următorul ciclu de ceas
4. Se repetă pasurile 2 și 3 până când este terminată ultima instrucțiune din miniprogram

4.3.3. ***Caz de utilizare**: vizualizarea conținuturilor componentelor*

1. Utilizatorul introduce miniprogramul în aplicație și pornește simularea (cazul de utilizare 4.3.1.)
2. Utilizatorul dă click, pe schema generală, pe componenta dorită
3. Se deschide un tabel în dreapta schemei generale care arată conținutul componentei selectate

4.3.4. ***Caz de utilizare****: vizualizare explicații de ajutor*

1. Utilizatorul introduce miniprogramul în aplicație și pornește simularea (cazul de utilizare 4.3.1.)
2. Pe toată durata simulării, utilizatorul poate vedea, în partea de jos a ferestrei, explicații teoretice pentru ciclul de ceas curent

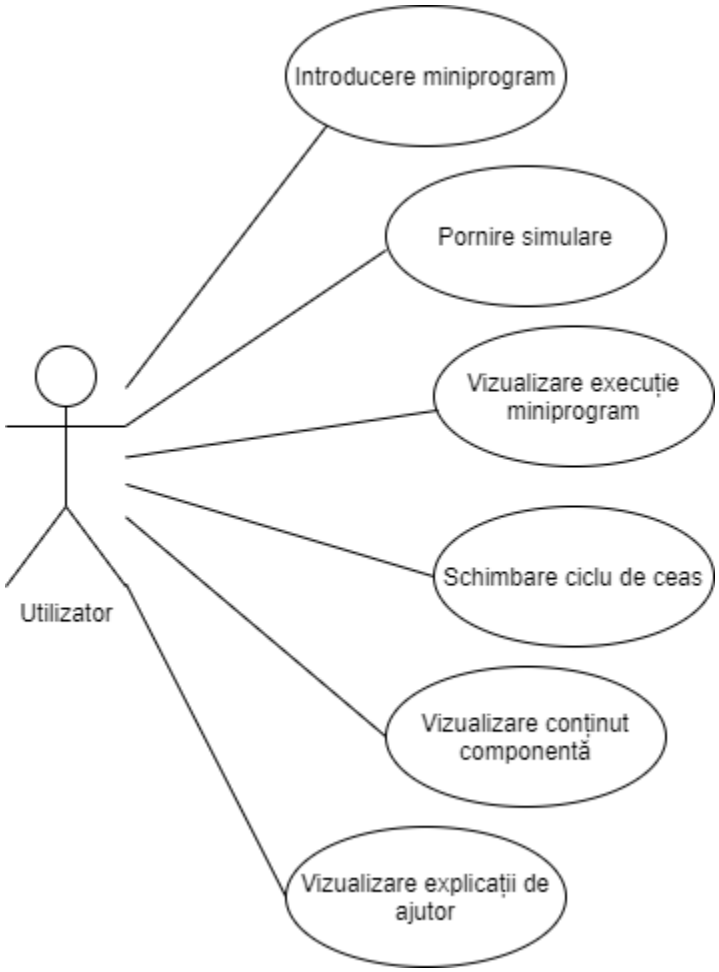


Fig. 4.2. Use-case diagram pentru simulator

4.4. Descrierea claselor

Clasele care vor fi implementate în aplicație vor fi descrise teoretic, în limbaj natural. Pentru componentele din bottom level vor fi implementate următoarele clase:

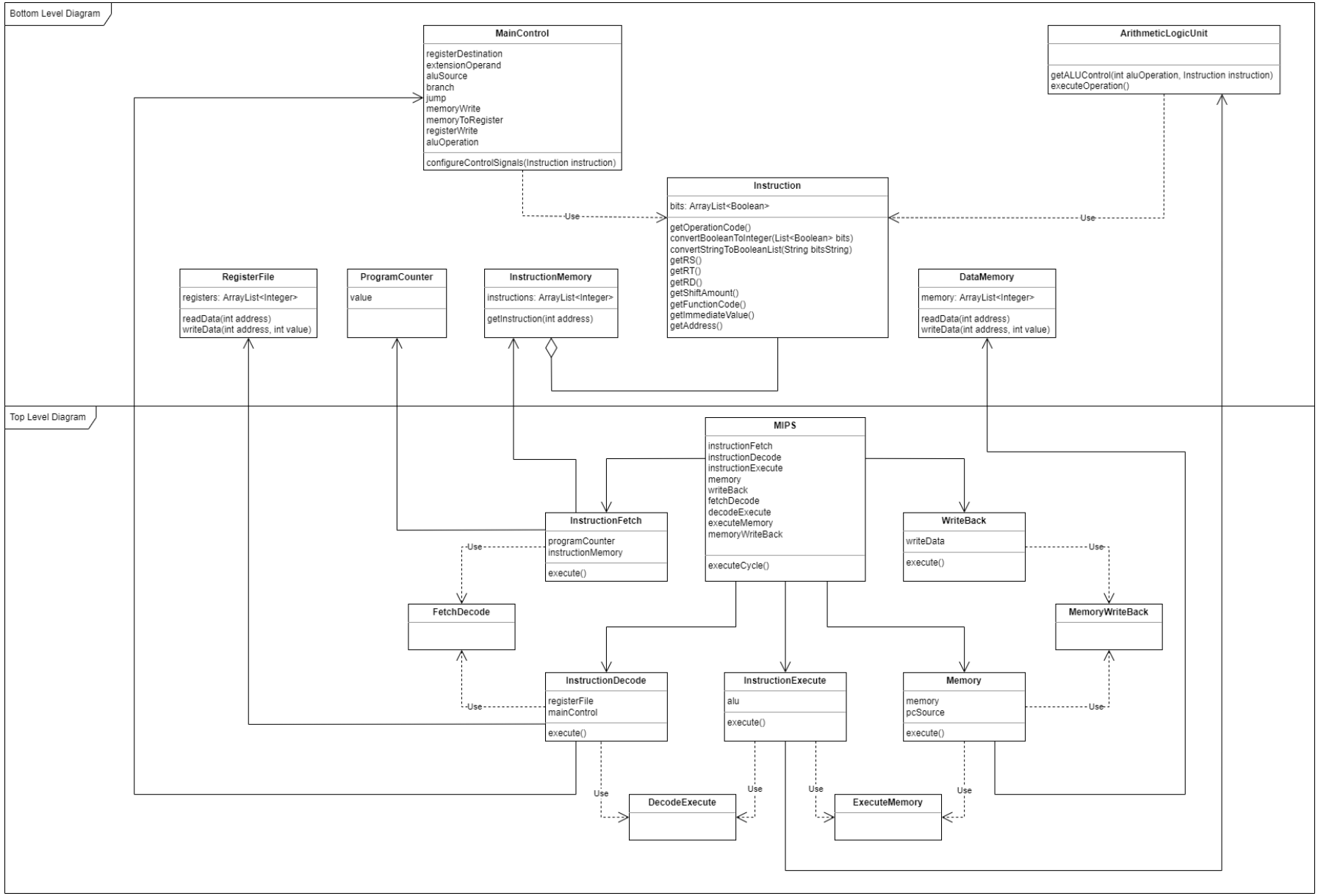
- **Instrucțiune –** conține șirul de 32 de biți care reprezintă formatul binar al instrucțiunii și metode specifice de extragere a grupărilor de biți
- **Unitatea aritmetico-logică –** conține o metodă pentru calcularea rezultatului operației
- **Memoria de date –** are ca atribut o listă care va reprezenta memoria propriu-zisă și are metode specifice de citire din memorie și scriere în memorie
- **Memoria de instrucțiuni –** are ca atribut o listă de instrucțiuni, care reprezintă miniprogramul și are o metodă de citire din memorie
- **Main Control –** are o metodă care modifică semnalele de control pentru restul componentelor în funcție de codul operației (*opcode*) dintr-o instrucțiune
- **Blocul de regiștri** - are ca atribut o listă care va reprezenta blocul de regiștri propriu-zis și are metode specifice de citire din regiștri și scriere în regiștri
- **Program Counter –** are ca atribut o valoare (*value*) ce reprezintă indicele instrucțiunii din memoria de instrucțiuni care urmează a fi executată

Pentru componentele din top level vor fi implementate următoarele clase:

- **Instruction Fetch –** conține program counter-ul și memoria de instrucțiuni
- **Instruction Decode –** conține blocul de regiștri și main control-ul
- **Instruction Execute –** conține unitatea aritmetico-logică
- **Memory –** conține memoria de date
- **Write Back –** se selectează data ce urmează a fi scrisă în blocul de regiștri
- **Registrul intermediar IF/ID –** conține instrucțiunea ce urmează a fi executată și indicele intrucțiunii următoare; face trecerea de la Instruction Fetch la Instruction Decode

- **Registrul intermediar ID/EX –** conține semnalele corespunzătoare instrucțiunii curente, datele citite din blocul de regiștri, și diferite grupări de biți din instrucțiune; face trecerea de la Instruction Decode la Instruction Execute
- **Registrul intermediar EX/MEM –** conține semnalele corespunzătoare instrucțiunii curente, rezultatul ALU și intrarile corespunzătoare pentru memoria de date; face trecerea de la Instruction Execute la Memory
- **Registrul intermediar MEM/WB -** conține semnalele corespunzătoare instrucțiunii curente, data citită din memorie și rezultatul ALU; face trecerea de la Memory la Write Back

Fig. 4.3. Diagrama de clase



5. Implementarea aplicației

În acest capitol sunt descrise toate clasele din aplicație, explicând atributele și metodele fiecăreia.

Pachetul **model.instruction**

Clasa **Instruction**

Atribute

- bits - șirul de 32 de biți care reprezintă instrucțiunea propriu-zisă

Metode

- getOperationCode( ) – returnează codul operației ca un întreg
- convertBooleanToInteger(List<Boolean> bits) – convertește un șir de valori booleene într-un număr întreg
- convertStringtoBooleanList(String bitsString) – convertește un șir de biți (string) într-un șir de valori booleene
- getRS( ) – returnează câmpul *rs* din instrucțiune ca un întreg
- getRT( ) – returnează câmpul *rt* din instrucțiune ca un întreg
- getRD( ) – returnează câmpul *rd* din instrucțiune ca un întreg
- getShiftAmount( ) – returnează câmpul *sa* din instrucțiune ca un întreg
- getFunctionCode( ) – returnează câmpul *func* din instrucțiune ca un întreg
- getImmediateValue( ) – returnează câmpul *imm* din instrucțiune ca un întreg
- getAddress( ) - returnează câmpul *address* din instrucțiune ca un întreg

Pachetul **model.component**

Clasa **ArithmeticLogicUnit**

Metode

- getALUControl(int aluOperation, Instruction instruction) – returnează semanul de control corespunzător pentru UAL
- executeOperation( )

Clasa **DataMemory**

Atribute

- memory – memoria de date propriu-zisă

Metode

- readData(int address) – returnează valoarea din memorie de la adresa indicată
- writeData(int address, int value) – scrie în memorie valoarea la adresa indicată

Clasa **RegisterFile**

Atribute

- registers – blocul de regiștri propriu-zis

Metode

- readData(int address) – returnează valoarea din registrul indicat
- writeData(int address, int value) – scrie valoarea în registrul indicat de adresă

Clasa **MainControl**

Atribute

- registerDestination
- extensionOperand
- aluSource
- branch
- jump
- memoryWrite
- memoryToRegister
- registerWrite
- aluOperation

Metode

- configureControlSignals(Instruction instruction) – configurează semnalele de control în funcție de câmpul *opcode* al intrucțiunii
- reset( ) – configurează toate semnalele la *false* sau *0* (resetează valorile semnalelor de control)

Clasa **InstructionMemory**

Atribute

- instructions – memoria de instrucțiuni propriu-zisă

Metode

- getInstruction (int address) – returnează instrucțiunea din memorie de la adresa indicată

Clasa **ProgramCounter**

Atribute

- value - valoare ce reprezintă indicele instrucțiunii din memoria de instrucțiuni care urmează a fi executată

Pachetul **model.stage**

Clasa **InstructionFetch**

Atribute

- programCounter
- instructionMemory

Metode

- execute( ) – se execută acțiunile de pe etajul Instruction Fetch: se calculează adresa instrucțiunii ce urmează a fi executată și se configurează registrul intermediar IF/ID cu datele necesare

Clasa **InstructionDecode**

Atribute

- registerFile
- mainControl

Metode

- execute( ) – se execută acțiunile de pe etajul Instruction Decode: se configurează semnalele de control pentru intrucțiunea curentă, se extrag datele din regiștri și se configurează registrul intermediar ID/EX cu datele necesare

Clasa **InstructionExecute**

Atribute

- alu – unitatea aritmetico-logică

Metode

- execute( ) – se execută acțiunile de pe etajul Instruction Execute: se configurează semnalul de control pentru ALU, se calculează rezultatul operației și se configurează registrul intermediar EX/MEM cu datele necesare

Clasa **Memory**

Atribute

- memory – memoria de date
- semnalul pcSource

Metode

- execute( ) – se execută acțiunile de pe etajul Memory: se citește din memorie, se scrie în memorie dacă semnalul specific permite și se configurează registrul intermediar MEM/WB cu datele necesare

Clasa **WriteBack**

Atribute

- writeData – data care urmează a fi scrisă în blocul de regiștri

Metode

- execute( ) – se execută acțiunile de pe etajul Write Back: se selectează data ce urmează a fi scrisă în blocul de regiștri în fucție de semnalul specific MemoryToRegister și se scrie în blocul de regiștri

Pachetul **mips.model.component.bufferregister**

Conține clasele care reprezintă regiștrii intermediari ai procesorului MIPS pipeline. Toate aceste clase implementează *design pattern-ul Singleton*.

Clasa **FetchDecodeRegister – registrul IF/ID**

Atribute

- instruction – instrucțiunea care va fi executată
- nextInstruction – indexul instrucțiunii următoare

Clasa **DecodeExecuteRegister – registrul ID/EX**

Atribute

- toate semnalele de control din **Main Control**
- nextInstruction – indexul următoarei instrucțiuni, luat din registrul IF/ID
- readData1 – data citită de la adresa indicată de RS
- readData2 – data citită de la adresa indicată de RT
- immediateValue – valoarea imediată din instrucțiune
- shiftAmount – valoarea cu care se va realiza deplasarea biților
- functionCode – codul funcției instrucțiunii R
- rt – adresa RT
- rd - adresa RD

Clasa **ExecuteMemoryRegister – registrul EX/MEM**

Atribute

- semnalele de control din blocurile WB și MEM de pe schemă, luate din registrul ID/EX

- branchAddress – adresa la care se va face saltul condiționat
- zeroSignal – semnal care are valoarea 1 dacă rezultatul scăderii este 0
- ALUResult – rezultatul operației din ALU
- writeData – data care se scrie în memorie
- writeAddress – adresa la care se va scrie în blocul de regiștri: RT sau RD în funcție de semnalul RegDst

Clasa **MemoryWriteBackRegister – registrul MEM/WB**

Atribute

- semnalele de control din blocul WB de pe schemă, luate din registrul EX/MEM
- readData – data citită din memorie
- aluResult – același din ExecuteMemoryRegister
- writeAddress – același din ExecuteMemoryRegister

Pachetul **mips.processor**

Clasa **MIPS**

Atribute

- instructionFetch – etajul Instruction Fetch
- fetchDecode – registrul intermediar IF/ID
- instructionDecode – etajul Instruction Decode
- decodeExecute – registrul intermediar ID/EX
- instructionExecute – etajul Instruction Execute
- executeMemory – registrul intermediar EX/MEM\

- memory – etajul Memory
- memoryWriteBack – registrul intermediar MEM/WB
- writeBack – etajul Write Back

Metode

- executeCycle( ) – se execută toate metodele etajelor microprocesorului în ordine inversă, imitând execuția MIPS pipeline pe un ciclu de ceas (sunt declarate două metode: una pentru a vedea ciclurile de ceas individual, cu ajutorul clasei HelpText, și una pentru execuția întregului miniprogram)
- reset( ) – resetează memoria de instrucțiuni și regiștri intermediari
- addInstruction(String bitsString) – adaugă o instrucțiune (format string 32 de biți) în memoria de regiștri
- loadProgram(ArrayList program) – adaugă o listă de șiruri de biți (instrucțiuni) în memoria de instrucțiuni
- treatHazards(ArrayList program) – tratează hazardurile miniprogramului
- clearInstructionMemory( ) – resetează memoria de instrucțiuni
- writeInRegisterFile(int address, int value) – scrie valoarea *value* în blocul de regiștri la adresa *address*
- writeInMemory (int address, int value) – scrie valoarea *value* în memoria de date la adresa *address*

Pachetul **mips.view**

Conține clasele care fac parte din interfața grafică a simulatorului.

Clasa **HelpText –** conține texte care vor fi afișate în secțiunea de ajutor a interfeței simulatorului

Atribute

- welcomeText – apare la început

- simulationText – apare în timpul simulării
- finalText – apare la sfârșitul simulării
- fetchText – apare la apăsarea etajului IF
- decodeText – apare la apăsarea etajului ID
- executeText – apare la apăsarea etajului IE
- memoryText – apare la apăsarea etajului MEM
- writeBackText – apare la apăsarea etajului WB

Clasa **Table –** conține tabelele pentru introducerea datelor în blocul de regiștri și memoria de date. Tabelele apar în secțiunea *Componente* din interfață.

Atribute

- table – tabela propriu-zisă
- model – modelul de tabel

Metode

- setRows(boolean isRegisterTable) – inițializează tabela în funcție de componentă (bloc de regiștri sau memorie de date)
- resetTable( ) – setează valorile din tabelă la 0
- updateTable(ArrayList values) – actualizează tabela cu valorile din listă; se apelează la fiecare ciclu de ceas

Clasa **View –** interfața grafică a simulatorului

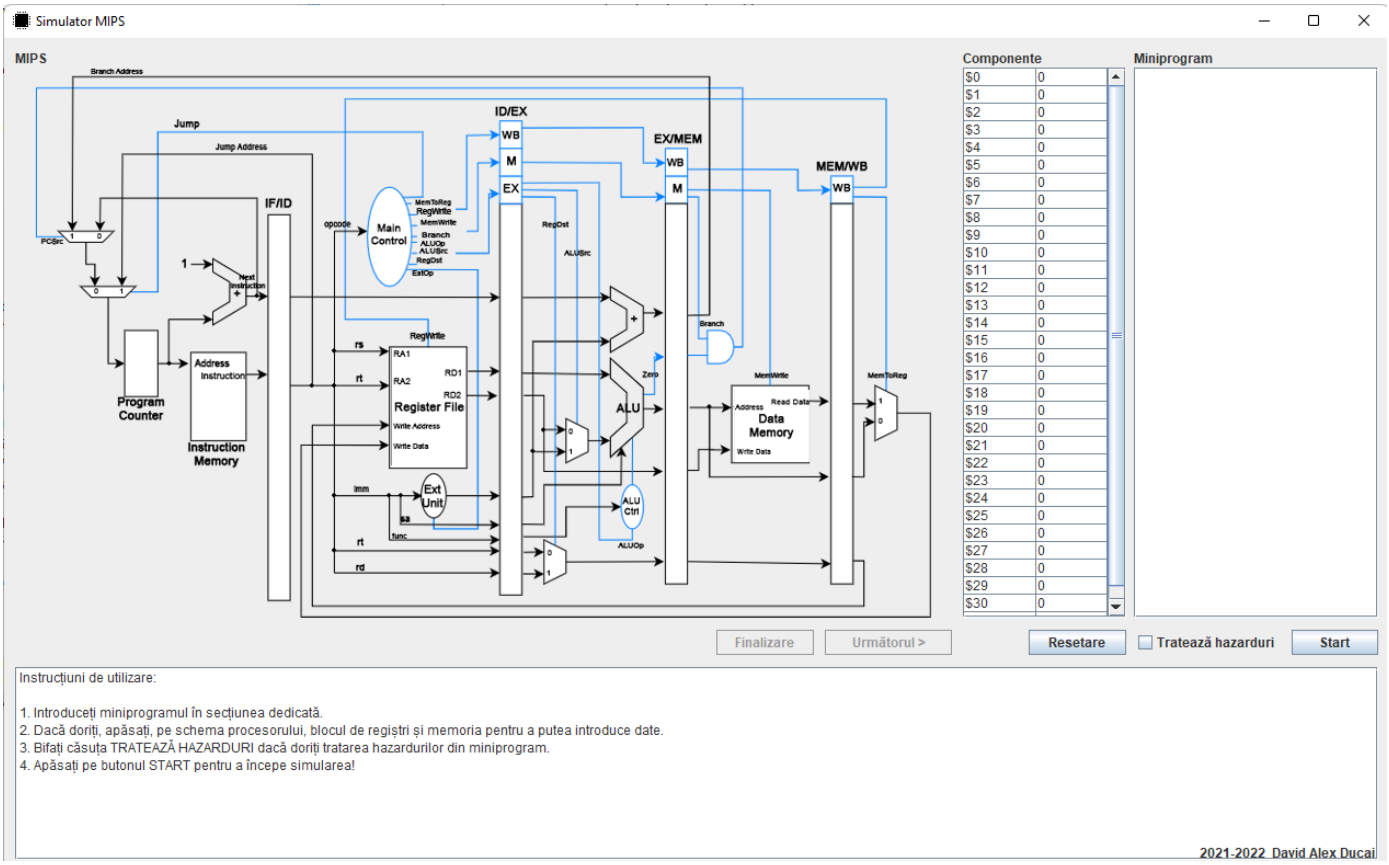


Fig. 5.1. Interfața grafică

Pachetul **mips.controller**

Clasa **Controller –** controller-ul pentru interfața aplicției

Metode

- actionPerformed(ActionEvent e) – se apelează la apăsarea unui buton și execută acțiunile specifice pentru butonul respectiv (Butonul de start, reset, pentru următorul ciclu de ceas și execuție totală/finalizare simulare)

- resetView( ) – se resetează interfața simulatorului; apelată la apăsarea butonului de resetare după terminarea unei simulări
- setSimulationDoneInterface( ) – setează interfața pentru terminarea simulării (se dezeactivează butoanele pentru simulare și se activează cel de resteare)
- updateInterface( ) – actualizează interfața la fiecare ciclu de ceas
- verifyProgram(ArrayList program) – verifică modul de scriere a programului introdus în secțiunea *Miniprogram*; simularea poate începe doar dacă s-au introdus șiruri de biți de lungime 32 pe câte un rând
- verifyTableValues(ArrayList values) – verifică datele introduse în tabelele pentru blocul de regiștri și memoria de date; simularea poate începe doar dacă s-au introdus valori numerice

Pachetul **mips.main**

Clasa **Launch –** pentru lansarea aplicației

6. Testarea aplicației

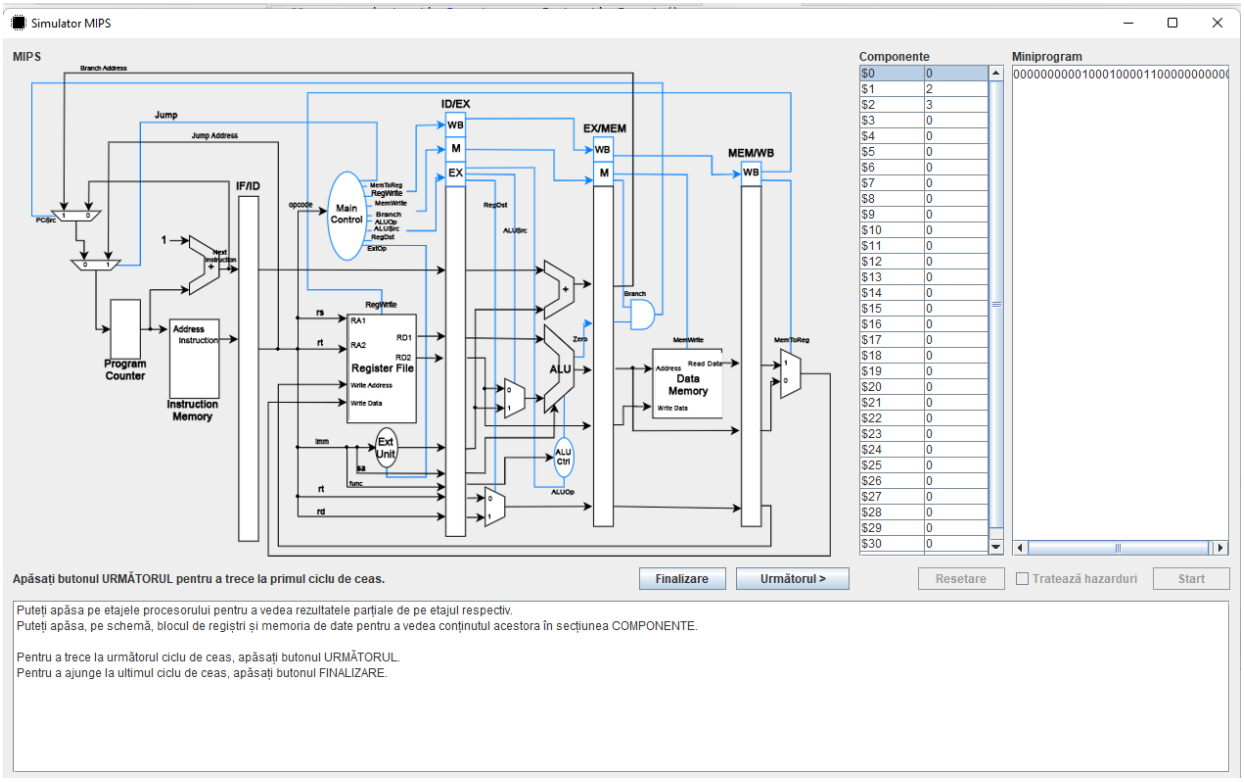
6.1. Utilizare

Secțiunea de ajutor din partea de jos a ferestrei este foarte utilă pe tot parcursul folosirii aplicației și se actualizează în funcție de necesitate:

- la început afișează modul de utilizare: introducerea miniprogramului
- la pornirea simulării indică acțiunea fiecărui buton
- la fiecare ciclu de ceas indică conținutul fiecărui etaj al procesorului
- la final afișează informații despre resetare

În următoarele rânduri se va prezenta aplicația, testând instrucțiuni.

Exemplu: instrucțiunea *add $3, $1, $2*



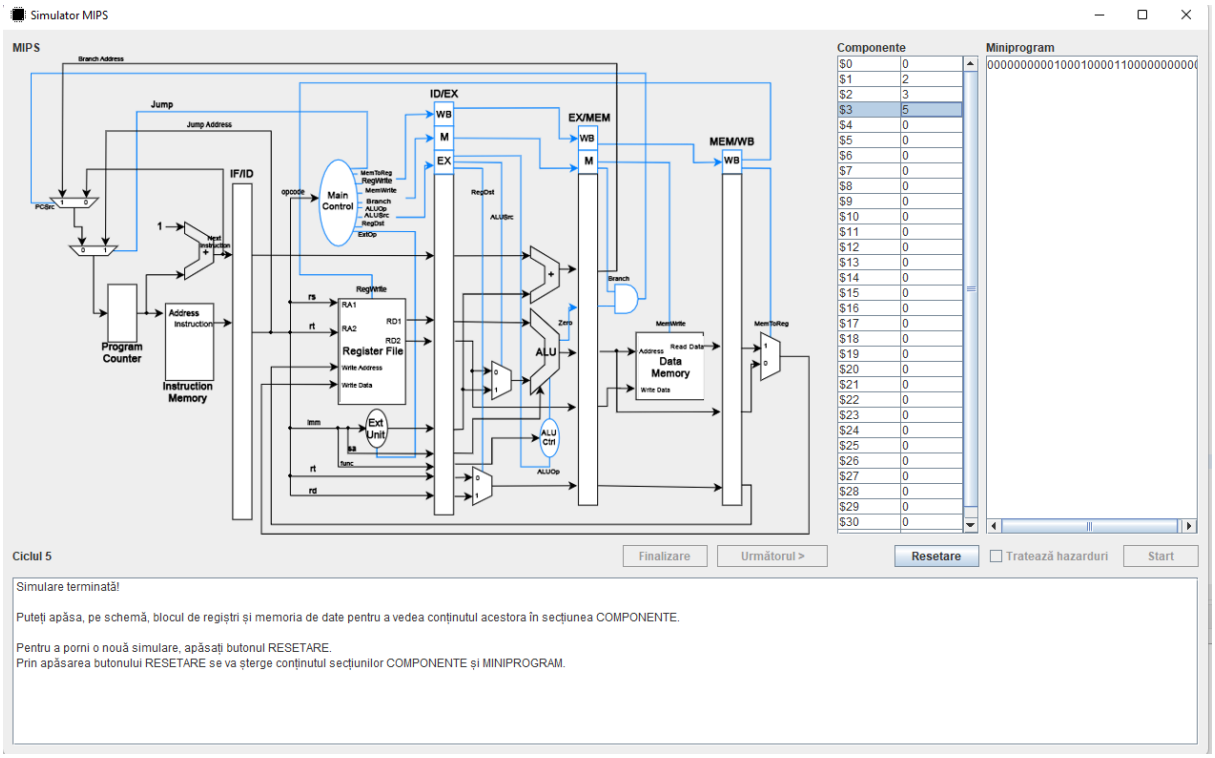


Fig. 6.1. și 6.2. Testarea instrucțiunii add

Am introdus în regiștri $1 și $2 valorile 2 și 3. Am apăsat butonul *Finalizare* pentru executarea tuturor ciclurilor de ceas. Rezultatul așteptat este valoarea 5 în registrul $3.

Exemplu: salt necondiționat

Vom simula următorul miniprogram:

```
j 2
add $6, $4, $5
add $3, $1, $2
```

Vom testa saltul la cea de-a treia instrucțiune. Am introdus în regiștri $1 și $2 valorile 2 și 3, ca în exemplul precedent, și valorile 1 și 1 în regiștri $4 și $5. În urma execuției miniprogramului ar trebui să avem valoarea 5 în registrul $3 și valoarea 0 în registrul $6, deoarece nu se execută cea de-a doua instrucțiune.

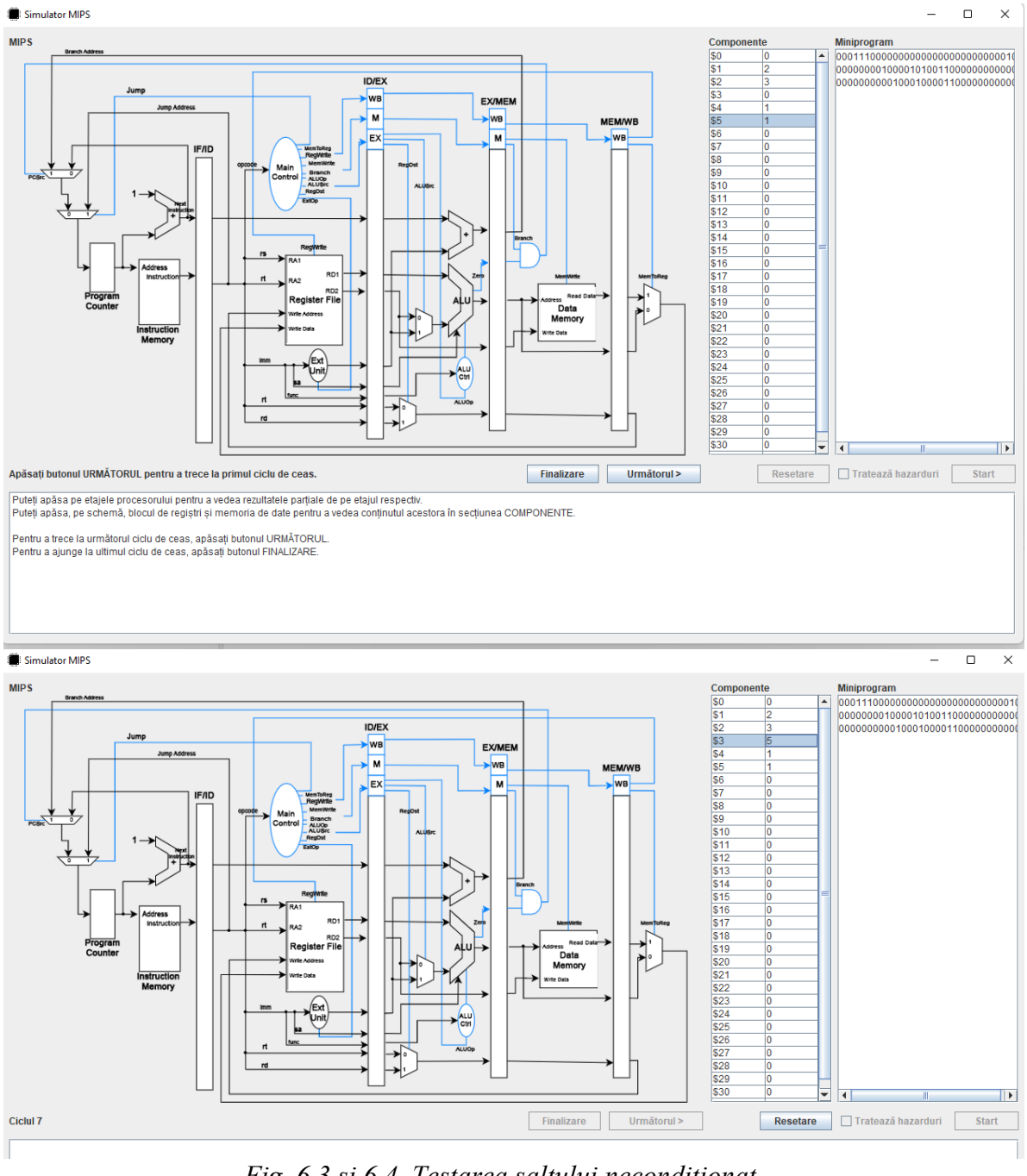


Fig. 6.3 și 6.4. Testarea saltului necondiționat

6.2. Testarea excepțiilor

Singurele excepții care pot apărea sunt la introducerea miniprogramului și a datelor în componente (blocul de regiștri și memoria de date). Secțiunea de introducere a miniprogramului permite numai introducerea de șiruri de 32 de biți. La introducerea oricăror altor caractere, nu se va permite trecerea mai departe spre simulare.

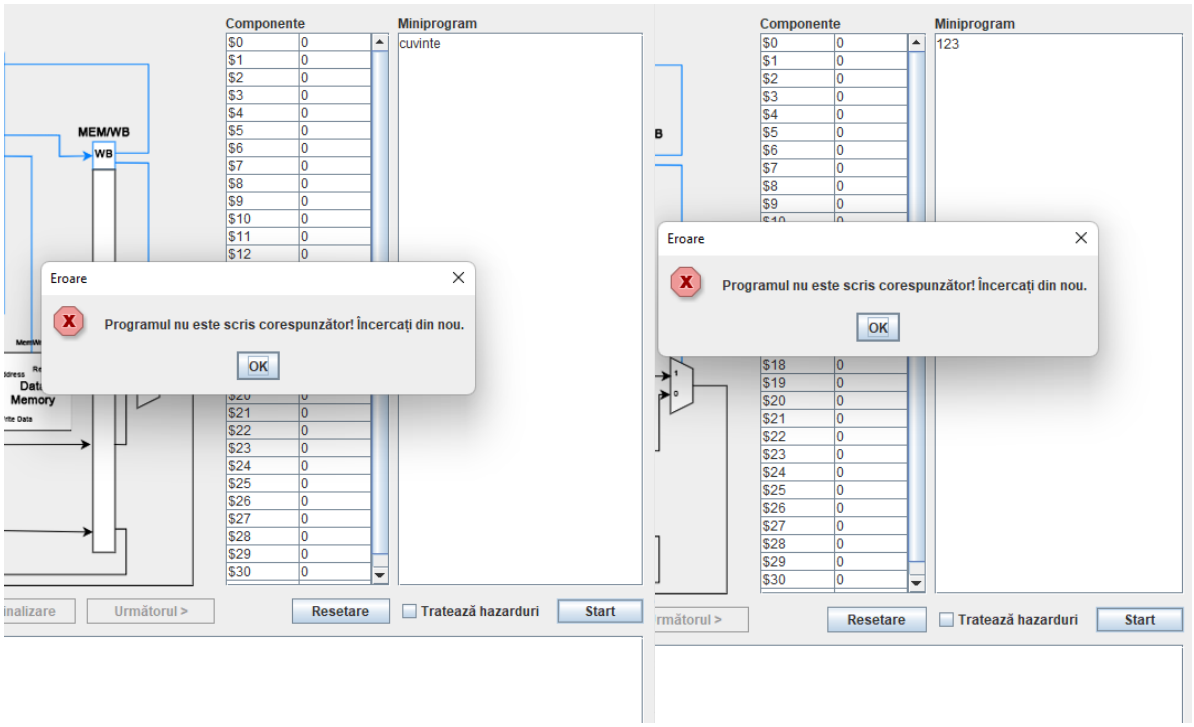


Fig. 6.5 . Introducerea greșită a miniprogramului

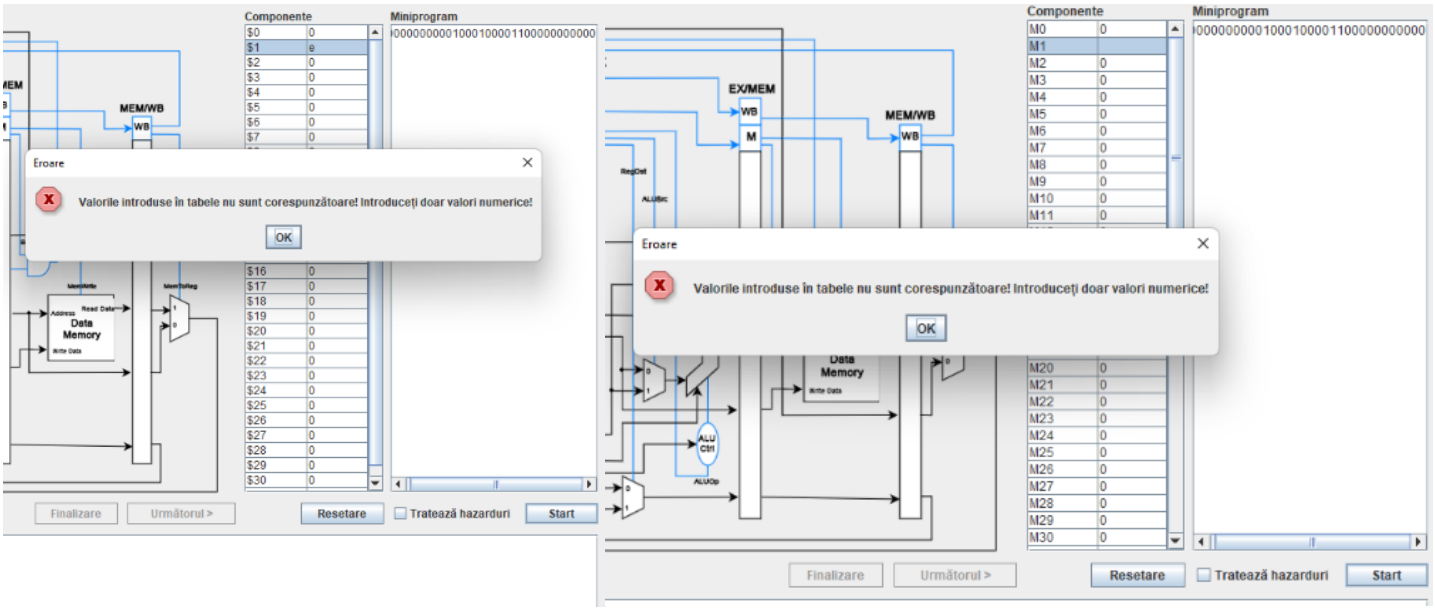


Fig. 6.6. Introducerea greșită a valorilor în tabele

7. Bibliografie

Pentru studiul teoretic și analiză:

[1] "Arhitectura MIPS", https://ro.wikipedia.org/wiki/Arhitectur%C4%83_MIPS

[2] Dr A. P. Shanthi, "10 Pipelining – MIPS Implementation" - https://www.cs.umd.edu/~meesh/411/CA-online/chapter/pipelining-mips-implementation/index.html

[3] "Classic RISC pipeline" - https://en.wikipedia.org/wiki/Classic_RISC_pipeline

[4] F. Oniga, M. Negru, „ARHITECTURA CALCULATOARELOR Îndrumător de laborator" - https://users.utcluj.ro/~onigaf/files/teaching/AC/AC_indrumator_laborator.pdf

[5] „Design Pattern - Singleton Pattern" - https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

[6] J. Jenkov, „Java Regex – Matcher" - http://tutorials.jenkov.com/java-regex/matcher.html