UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Simulator de UCP in arhitectura pipeline

Iepure Denisa-Alexandra
grupa: 30233

Conținut:

1. Introducere

1.1 Context

Unitatea centrală de prelucrare (UCP) sau unitatea centrală de procesare (CPU) reprezintă componenta hardware a unui sistem informatic care efectuează execuția instrucțiunilor unui program de calculator, efectuând operații aritmetice și logice, precum și gestionând operațiile de intrare/ieșire (I/O - input/output) ale sistemului.

Proiectul are ca scop implementarea unui CPU in arhitectura pipeline. Tehnica pipeline (pipeline = bandă de execuţie/ conductă de execuţie) este o tehnică de introducere a paralelismului în execuţia instrucţiunilor. Aceasta se face prin împărţirea ciclului de execuţie al unei instrucţiuni în mai multe faze de prelucrare, iar execuţia paralelă a mai multor instrucţiuni se bazează pe faptul că fiecare dintre aceste instrucţiuni se află în altă fază de prelucrare.Arhitectura tip pipeline îmbunătățește performanța prin suprapunerea operațiilor pentru diferite instrucțiuni, astfel încât, în mod ideal, fiecare

etapă a fiecărei instrucțiuni să fie folosită în fiecare ciclu de ceas, conducând la o creștere a ratei de execuție a instrucțiunilor.

Programul va permite simularea a 15 instructiuni pe numere de 32 de biti , simularea este realizata in Java.

1.1 Specificații

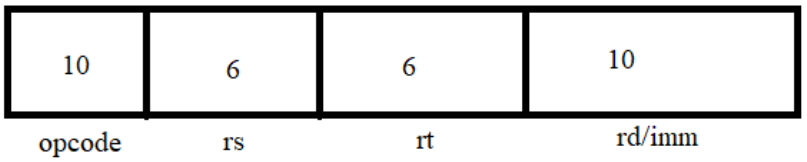
În limbajul C voi realiza un simulator care sa modeleze functionarea CPU -ului cu arhitectură de tip pipeline. Acest simulator va simula execuția instrucțiunilor în diferite etape , astfel simulând logica specifică pentru fiecare instrucțiune. Astfel, pe parcursul unei instrucțiuni , în fiecare etapă voi avea acces la un rezultat temporar, verificând corectitudinea din fiecare registru folosit.

1.1 Obiective

Obiectivul este proiectarea unui UCP in arhitectură pipeline. Voi defini 15 instructiuni în cod mașină (add, , sub, mul etc.) și simularea acestora în diferite etape pipeline, respectând logica specifică fiecărei etape.

2.Studiu Bibliografic

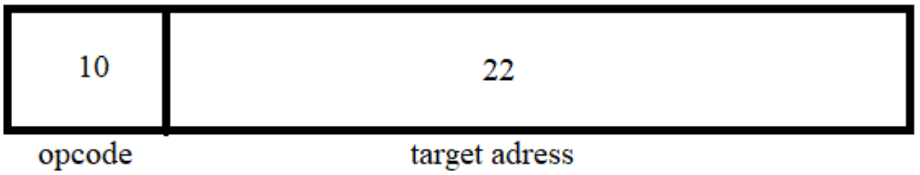
Formatele de instructiuni sunt folosite pentru logica de folosire propriu zisa a instructiunii. Astfel, am codificat in cod masina (limbajul de bază al mașinilor și calculatoarelor) cele 15 instructiuni . Am ales sa codific dupa acest format :



- **opcode-ul :** identifică operația specifică pe care CPU trebuie să o efectueze. De exemplu, un opcode specific poate indica o operațiune aritmetică (cum ar

fi adunarea sau înmulțirea), o operațiune logică (cum ar fi AND sau OR), sau o operațiune de transfer de date (cum ar fi încărcarea sau salvarea în memorie).

- **rs/rt/rd(imm)** rs=registrul sursa;
  rt= este indexul celui de al doilea registru sursă / indexul registrului destinație, rd= indexul registrului destinatie ;
  imm(imediatul) este in functie de tipul de instructiune :
  -> sa (cantitatea de deplasare)
  -> immediat (constanta folosita in operatiile aritmetice)
  -> target adress pentru instructiunea de tip J care are urmatorul format:



În procesarea instrucțiunilor de tip I în arhitecturile de calculatoare, precum MIPS, se întâlnește adesea necesitatea de a face o extensie cu semn asupra imediatului asociat. Aceasta se datorează faptului că registrele implicate în instrucțiuni au dimensiuni diferite în biți, iar operațiunile necesită coerență în gestionarea semnelor.

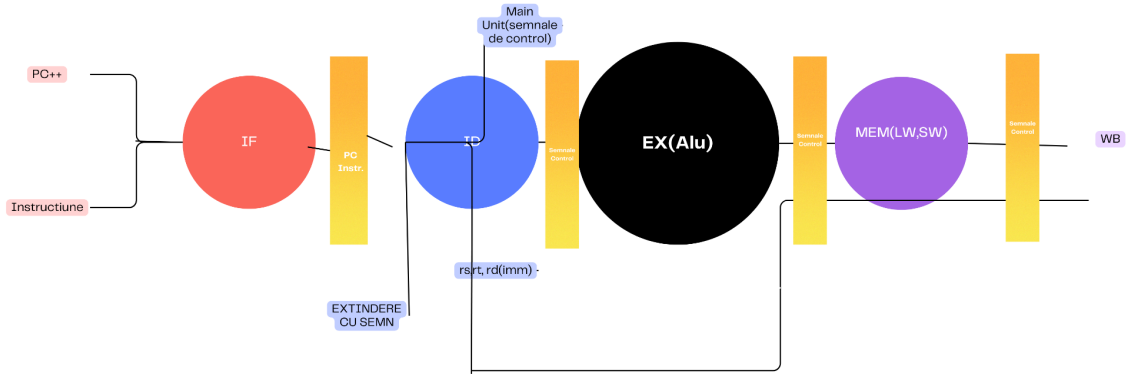
De exemplu, în cazul instrucțiunilor de tip I, cum ar fi `ori` (sau logic cu imediat/constantă) și `addi` (adunare cu imediat/constantă), este esențial să se extindă semnul imediatului pentru a asigura coerența semnelor în execuție. Acest lucru se face prin extinderea semnului imediatului la dimensiunea corespunzătoare a registrelor implicate în instrucțiune.

Astfel, în timpul executării instrucțiunilor de acest tip, se efectuează operații de extensie cu semn pentru a asigura interpretarea corectă a valorilor și pentru a preveni erorile legate de semn.

2.Analiză

în analiză am proiectat acestă schiță a programului

Pipeline architecture



Extragerea instrucțiunii (IF - Instruction Fetch):

- Memoria: Memoria program este accesată pentru a prelua instrucțiunea de la adresa specificată de PC.
- Registrul Contor de Program (PC): Se adaugă 4 la valoarea curentă a PC pentru a obține adresa următoarei instrucțiuni.
- Sumator: Realizează operația PC + 4 pentru a calcula adresa următoarei instrucțiuni.

Interpretarea instrucțiunii (ID - Instruction Decode):

- Blocul de Registre (RF - Register File): Se citesc valorile din registrele specificate în instrucțiune (de exemplu, `rs`, `rt`) pentru a fi folosite în execuție.
- Unitate de Extensie cu sau fără Semn: Extinde valoarea imediatului asociat instrucțiunii la 32 de biți, fie cu semn, fie fără, în funcție de tipul instrucțiunii.
- Unitatea de Control: Gestionează semnalele de control pentru diferitele componente ale execuției instrucțiunii.

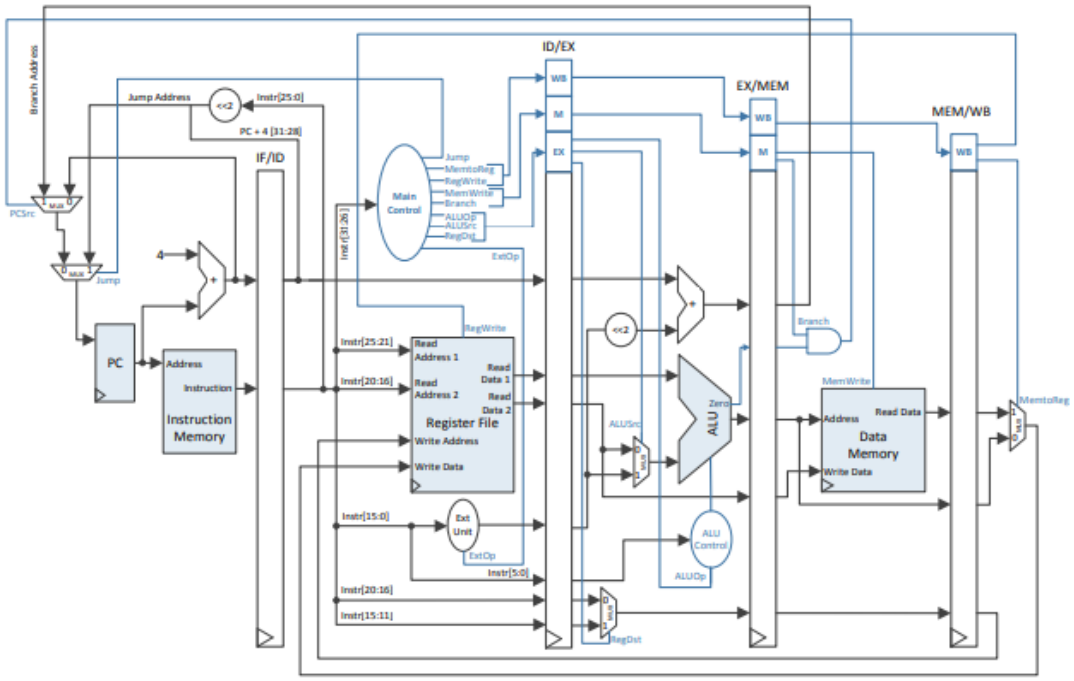
Execuția instrucțiunii (EX - Execute):

- ALU (Aritmetic-Logic Unit): Pentru instrucțiunile de tip R, se efectuează operații aritmetice și logice între valorile din registre.
- Calcul adrese: Pentru instrucțiunile de lucru cu memorie, se calculează adresa efectivă a locației de memorie (de exemplu, în cazul instrucțiunilor `lw` și `sw`).
- Evaluarea condiției de salt (`beq`): Se efectuează o comparație între valorile din două registre pentru a determina dacă condiția de salt este îndeplinită.

Aceste etape constituie un ciclu de execuție al unei instrucțiuni, iar în continuare, în etapele `MEM` și `WB`, se vor gestiona operațiunile de acces la memorie și scrierea

rezultatelor înapoi în registre. Este important de menționat că pentru fiecare tip de instrucțiune, se efectuează anumite operații specifice în aceste etape, și anume, accesul la memorie pentru `lw` și `sw`, respectiv scrierea rezultatelor înapoi în registre pentru instrucțiunile de tip `R` sau `I`.

4.Design



Pentru design , am ales ca sursă de inspirație o figură din îndrumătorul de laborator de la AC, pe care ulterior am adaptat-o pentru a evidenția beneficiile aduse de limbajul Java în comparație cu VHDL. Etapele procesului respectă informațiile din figura inițială.

IF (Instrucțiunea de Fetch):

- În această etapă, se produce creșterea conținutului registrului program (Pc) și se identifică instrucțiunea corespunzătoare.

ID (Instrucțiunea de Decodificare):

- Decodificarea instrucțiunii are loc bazându-ne pe logica specifică fiecărui tip de instrucțiune.
- Se generează semnalele de control corespunzătoare fiecărui tip de instrucțiune.
- Se extinde imediatul cu semn, acolo unde este necesar.

EX (Instrucțiunea de Execuție):

- În această fază, se desfășoară operațiile care implică Unitatea Aritmetică și Logică (ALU).

MEM (Instrucțiuni de Lucru cu Memoria):

- Pentru instrucțiunile care implică lucrul cu memoria (de exemplu, lw și sw), se accesează blocul de memorie corespunzător.
- Notă importantă: În această etapă, nu este necesar să avem două blocuri distincte de memorie pentru instrucțiuni și date; putem utiliza un singur bloc comun.

WB (Write-Back):

- Rezultatul execuției se scrie înapoi în blocul de registre (RF), asigurându-ne că datele actualizate sunt disponibile pentru viitoarele operațiuni.

Această arhitectură respectă principiile limbajului Java, aducând beneficiile sale într-un context specific de procesare a instrucțiunilor. Modificările aduse în comparație cu VHDL sunt menite să optimizeze și să evidențieze caracteristicile avantajoase ale limbajului Java în cadrul acestui design

5.Implementare

1.1 Implementarea efectivă a arhitecturii pipeline

Implementarea conceptului de arhitectură pipeline o realizez prin intermediul unei liste de instrucțiuni care sunt active , adica in interiorul unui stage (IF, ID, EX,MEM, WB). O instructiune este caracterizată printr-o adresă , cea a pc-ului, și un stage, care reprezintă starea actuală a instrucțiunii.

```java
public class Instruction {
    1 usage
    public String codificare;
    7 usages
    public int stage;

    1 usage
    public Instruction(String codificare) {
        this.codificare = codificare;
        this.stage = 1;
    }
}
```

```java
public static ArrayList<Instruction> running = new ArrayList<~>();
```

Pentru a avea acel paralelism adaug instructiunile din `MemoryInstruction` și le adaug pe rând în listă , iar în funcție de stage apelez metoda necesară realizării stage-ulu.

```java
for (int i = 0; i < running.size(); i++) {
    if (!running.isEmpty() && i >= 0) {
        System.out.println("IF.pc" +IF.pc);
        switch (running.get(i).stage) {
            case 1:
                IF.IF(IF.pc);
                running.get(i).stage += 1;
                break;
            case 2:
                ID.ID();
                running.get(i).stage += 1;
                break;
            case 3:
                EX.ex();
                running.get(i).stage += 1;
                break;
            case 4:
                Mem.mem();
                running.get(i).stage += 1;
                break;
            case 5:
                WB.writeBack();
                running.get(i).stage += 1;
                break;
            case 6:
                running.remove(i);
                i--;
                break;
        }
```

1.2 Implementarea componentelor principale: ALU, RegFile, MemoryData

REGISTER FILE reprezintă o implementare simplă a unui set de 32 de registri, utilizând un limbaj de programare Java. Fiecare registru este simulat printr-un șir de caractere de lungime fixă de 32 de biți, reprezentând astfel o valoare binară.

```
    public static String[] registers = new String[32];

static {
    registers[0]  = "00000000000000000000000000000000";
    registers[1]  = "00000000000000000000000000000001";
    registers[2]  = "00000000000000000000000000000010";
    registers[3]  = "00000000000000000000000000000011";
    registers[4]  = "00000000000000000000000000000100";
    registers[5]  = "00000000000000000000000000000101";
    registers[6]  = "00000000000000000000000000000110";
    registers[7]  = "00000000000000000000000000000111";
    registers[8]  = "00000000000000000000000000001000";
    registers[9]  = "00000000000000000000000000001001";
    registers[10] = "00000000000000000000000000001010";
    registers[11] = "00000000000000000000000000001011";
    registers[12] = "00000000000000000000000000001100";
    registers[13] = "00000000000000000000000000001101";
    registers[14] = "00000000000000000000000000001110";
    registers[15] = "00000000000000000000000000001111";
    registers[16] = "00000000000000000000000000010000";
    registers[17] = "00000000000000000000000000010001";
    registers[18] = "00000000000000000000000000010010";
    registers[19] = "00000000000000000000000000010011";
    registers[20] = "00000000000000000000000000010100";
    registers[21] = "00000000000000000000000000010101";
    registers[22] = "00000000000000000000000000010110";
    registers[23] = "00000000000000000000000000010111";
```

Alu (Unitatea Aritmetică și Logică) în limbajul de programare Java este implementată prin metode care primesc registrii ca parametri. Aceste metode efectuează operații aritmetice și logice pe valorile primite și returnează rezultatele acestor operații.

Memoria (RAM) definește o structură de memorie de date și include un bloc de inițializare statică pentru a seta anumite valori predefinite.

1.3 Implementarea registrilor intermediari

În implementarea arhitecturii pipeline, este esențială utilizarea registrelor intermediare pentru stocarea datelor între diferitele etape ale procesului. Acești registri intermediari permit transmiterea eficientă a rezultatelor de la o etapă la alta, ceea ce contribuie la îmbunătățirea performanței și eficienței sistemului.

Imaginați-vă un flux continuu de date care parcurge diferite etape ale procesorului. Pentru a asigura sincronizarea și corectitudinea operațiilor, rezultatele intermediare dintr-o etapă sunt stocate în registri la adrese corespunzătoare. Acești registri servesc ca punți între etapele procesului, permițând transferul controlat și ordonat datelor.

```
4 usages
public static String[] IFID = new String[500000];
83 usages
public static String[] IDEX = new String[500000];
90 usages
public static String[] EXMEM = new String[500000];
75 usages
public static String[] MEMWB = new String[500000];

// index
4 usages
public static int IFIDindex = 0;
32 usages
public static int IDEXindex = 0;
144 usages
public static int EXMEMindex = 0;
36 usages
public static int MEMWBindex = 0;
2 usages
public static int cycle = 0;
```

6.Testare

```
instMemory[0] = "00000000010000010000100000000000"; // add  r1=1 , r2=2 => r0=3
instMemory[1] = "00000000110001110000110000001101"; // sub  r7=7 , r3=3 => r13=4
instMemory[2] = "00000000100000110110000000000100"; // addi r3=3 , r24=24 , r4=4 => r24 = 7
instMemory[3] = "00000001000001000000110000001111"; // mult r4=4 , r3=3 , r15=15 => r15=12
instMemory[4] = "00000001010010110011000000010001"; // and  r11=11 , r12=12, r17=17 => r17= 1000 (8)
instMemory[5] = "00000001100001100101110000000011"; // ori r6=6 , r23= 23 , r3=3 => r23=7
instMemory[6] = "00000001110010100000000000000010"; // sll  3*4=12 r10=10= d, r0=3 =t, h= 2 => rd(10)<= 3*2*2 =12
instMemory[7] = "00000010000110000101000000000001"; // srl 20/2=10 r24=7(d) , r20=20 =t , h=1 => rd(24) <= 20/2 =10
instMemory[8] = "00000010010010000000000000000111"; // lw  r8=8(s) , r0=3(t) h=7 => r0=mem(8+7) -> r0=0
instMemory[9] = "00000010100011000000010000001111"; // sw   r12=12(s) , r1=1 (t) h=15 => mem(12+15) = 1 => mem(27)=1
instMemory[10] ="00000011010111010011110000010010"; // slt 29 mai mare ca 15 =: rd=0
instMemory[11]= "10000000000101110100110000001110 ";//or r[14] <= r[23]=7 or r[19]=19 10111
instMemory[12]= "10000000010101110100110000011111";//xor r[31]<= 23 xor 19 10100
instMemory[13]= "10000000100011010010110000001010";//andi rs13=4 , imm=1010(10) => rt11= 0100 & 1010 => 0000
instMemory[14]= "00000010110001010001110000001010"; // branch not equal r5=5, r7= 7 5!=7 BranchAddress=10
//if $s != $t PC ← PC + 4 + (offset << 2); else PC ← PC + 4;
```

- **ADD**

```
ADD r1 r2 0 in WriteBack stage:
RD 0register new value 00000000000000000000000000000011
ALUres 00000000000000000000000000000011
```

```
instMemory[0] = "00000000010000010000100000000000"; // add  r1=1 , r2=2 => r0=3
```

- **SUB**

```
SUB r7 r3 13 in WriteBack stage:
RD 13register new value 00000000000000000000000000000100
ALUres 00000000000000000000000000000100
```

```
instMemory[1] = "00000000110001110000110000001101"; // sub  r7=7 , r3=3 => r13=4
```

- **ADDI**

```
ADDi r3 r24 4 in WriteBack stage:
RT24 register new value 00000000000000000000000000000111
ALUres 00000000000000000000000000000111
```

```
instMemory[2] = "00000000100000110110000000000100"; // addi r3=3 , r24=24 , r4=4 => r24 = 7
```

- **MULT**

```
MULT r4 r3 15 in WriteBack stage:
RD 15register new value 00000000000000000000000000001100
ALUres 00000000000000000000000000001100
```

```
instMemory[3] = "00000001000001000000110000001111"; // mult r4=4 , r3=3 , r15=15 => r15=12
```

- **AND**

```
AND r11 r12 17 in WriteBack stage:
RD 17register new value 00000000000000000000000000001000
ALUres 00000000000000000000000000001000
```

```
instMemory[4] = "00000001010010110011000000010001"; // and  r11=11 , r12=12, r17=17 => r17= 1000 (8)
```

- **ORI**

```
ORi r6 r23 3 in WriteBack stage:
RT23 register new value 00000000000000000000000000000111
ALUres 00000000000000000000000000000111
```

```
instMemory[5] = "00000001100001100101110000000011"; // ori r6=6 , r23= 23 , r3=3 => r23=7
```

- **SLL**

```
SLL r10 r0 2 in WriteBack stage:
RD 10register new value 00000000000000000000000000001100
ALUres 00000000000000000000000000001100
```

```
instMemory[6] = "00000001110010100000000000000010"; // sll  3*4=12 r10=10= d, r0=3 =t, h= 2 => rd(10)<= 3*2*2 =12
```

- **SRL**

```
SRL r24 r20 1 in WriteBack stage:
RD 24register new value 00000000000000000000000000001010
ALUres 00000000000000000000000000001010
WB Controls : MemToReg : 0 RegWrite : 1
MEM Controls : MemRead : 0 MemWrite : 0 PCSrc : 0
EXT Controls : RegDst : 1
```

```
instMemory[7] = "00000010000110000101000000000001"; // srl 20/2=10 r24=7(d) , r20=20 =t , h=1 => rd(24) <= 20/2 =10
```

- **LW**

```
LW  rt 0 MEM[rs+offser]00000000000000000000000000001111 in WriteBack stage:
rt new data : 0
ALUresult : 00000000000000000000000000001111
```

```
instMemory[8] = "00000010010010000000000000000111"; // lw  r8=8(s) , r0=3(t) h=7 => r0=mem(8+7) -> r0=0
```

- **SW**

```
SW  memory new value 1 in WriteBack stage:
```

```
instMemory[9] = "00000010100011000000010000001111"; // sw  r12=12(s) , r1=1 (t) h=15 => mem(12+15) = 1 => mem(27)=1
```

- **SLT**

```
SLT r29 r15 r0 in WriteBack stage:
rd new data : 00000000000000000000000000000000
ALUresult : 00000000000000000000000000000000
```

```
instMemory[10] ="00000011010111010011110000010010"; // slt 29 mai mare ca 15 =: rd=0
```

- **OR**

```
OR rS 23 rT 19 rD 14 in WriteBack stage:
RD14 register new value 00000000000000000000000000010111
ALUres 00000000000000000000000000010111
```

```
instMemory[11]= "10000000000101110100110000001110 ";//or r[14] <= r[23]=7 or r[19]=19 10111
```

- **XOR**

```
XOR rS 23 rT 19 rD 31 in WriteBack stage:
RD31 register new value 00000000000000000000000000010100
ALUres 00000000000000000000000000010100
```

```
instMemory[12]= "10000000010101110100110000011111";//xor r[31]<= 23 xor 19 10100
```

- **ANDI**

```
ANDi rS 13 rT 11 rD 10 in WriteBack stage:
RT11 register new value 00000000000000000000000000000000
ALUres 00000000000000000000000000000000
```

```
instMemory[13]= "10000000100011010010110000001010";//andi rs13=4 , imm=1010(10) => rt11= 0100 & 1010 => 0000
```

- **BNE**

```
BNE rs5 rt7 10 in Execute stage:
pc new  value  :
BranchAddressResult : 0000001010
```

```
instMemory[14]= "00000010110001010001110000001010"; // branch not equal r5=5, r7= 7 5!=7 BranchAddress=10
```

7.Concluzie

Scopul acestui proiect a fost implementarea unei arhitecturi pipeline pentru o Unitate Centrală de Procesare (UCP) în limbajul de programare Java. Această arhitectură pipeline aduce beneficii semnificative în ceea ce privește performanța și eficiența execuției instrucțiunilor, prin suprapunerea operațiilor pentru diferite instrucțiuni.Beneficiile limbajului Java în cadrul acestui proiect au fost evidențiate prin flexibilitatea și ușurința în manipularea obiectelor, gestionarea listelor și gestionarea fluxului de control. Java oferă instrumente puternice pentru lucrul cu obiecte, ceea ce a permis implementarea eficientă a logicii specifice fiecărei etape a arhitecturii pipeline. De asemenea, gestionarea memoriei și a listelor a fost simplificată, contribuind la claritatea și concizia codului.

Comparativ cu limbajul VHDL, limbajul Java aduce avantaje în ceea ce privește sintaxa clară și modernă, gestionarea mai ușoară a memoriei și o integrare mai bună cu concepte de programare orientată pe obiect. Aceste caracteristici au facilitat dezvoltarea și înțelegerea proiectului.

În final, implementarea arhitecturii pipeline în Java a demonstrat eficiența limbajului în contextul procesării instrucțiunilor într-un mod paralel și optimizat, conform principiilor unei arhitecturi pipeline. Acest proiect reprezintă o simulare utilă și educativă a funcționării unui CPU cu arhitectură de tip pipeline într-un mediu de programare modern.

8.Bibliografie

https://users.utcluj.ro/~onigaf/files/teaching/AC/AC_indrumator_laborator.pdf
https://biblioteca.utcluj.ro/files/carti-online-cu-coperta/366-0.pdf