

Lucrarea 2

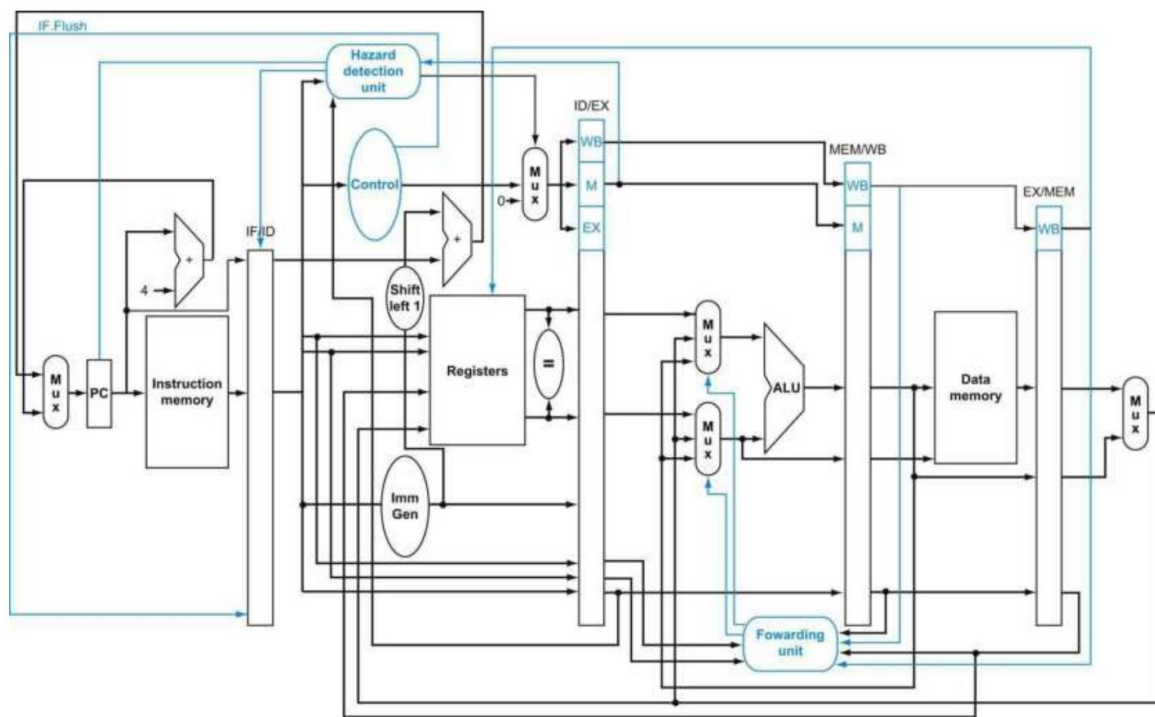


Fig. 1: Pipeline RISC-V

Fig. 1 prezinta schema bloc a unui procesor RISC-V cu 5 etape de pipeline:

1. IF(Instruction Fetch): citirea instructiunii curente
2. ID(Instruction Decode): decodarea instructiunii si citirea registrilor
3. EX(Execute): executarea instructiunii
4. MEM(Memory): operatii de citire/scriere cu memoria de date
5. WB(Write Back): scrierea rezultatelor finale inapoi in registri

Lucrarea de fata isi propune implementarea primelor 2 etape de pipeline: IF+ID.

Pentru a executa o instructiune, primii pasi pe care trebuie sa-i facem, sunt sa citim instructiunea din memoria (mai precis codificarea acesteia), si sa o decodificam pentru a intelege anumite aspecte despre ea: ce tip de instructiune este, ce operatie trebuie executata, ce registri trebuie folositi, etc.

Pentru a ne pregati de executia urmatoarei instructiuni, trebuie de asemenea sa incrementam contorul de program(program counter - PC), astfel incat adresa sa de iesire sa reprezinte urmatoarea instructiune din memorie, aflata la o distanta de 4 octeti de instructiunea curenta(mai multe detalii la Cap 4.3 din [1])

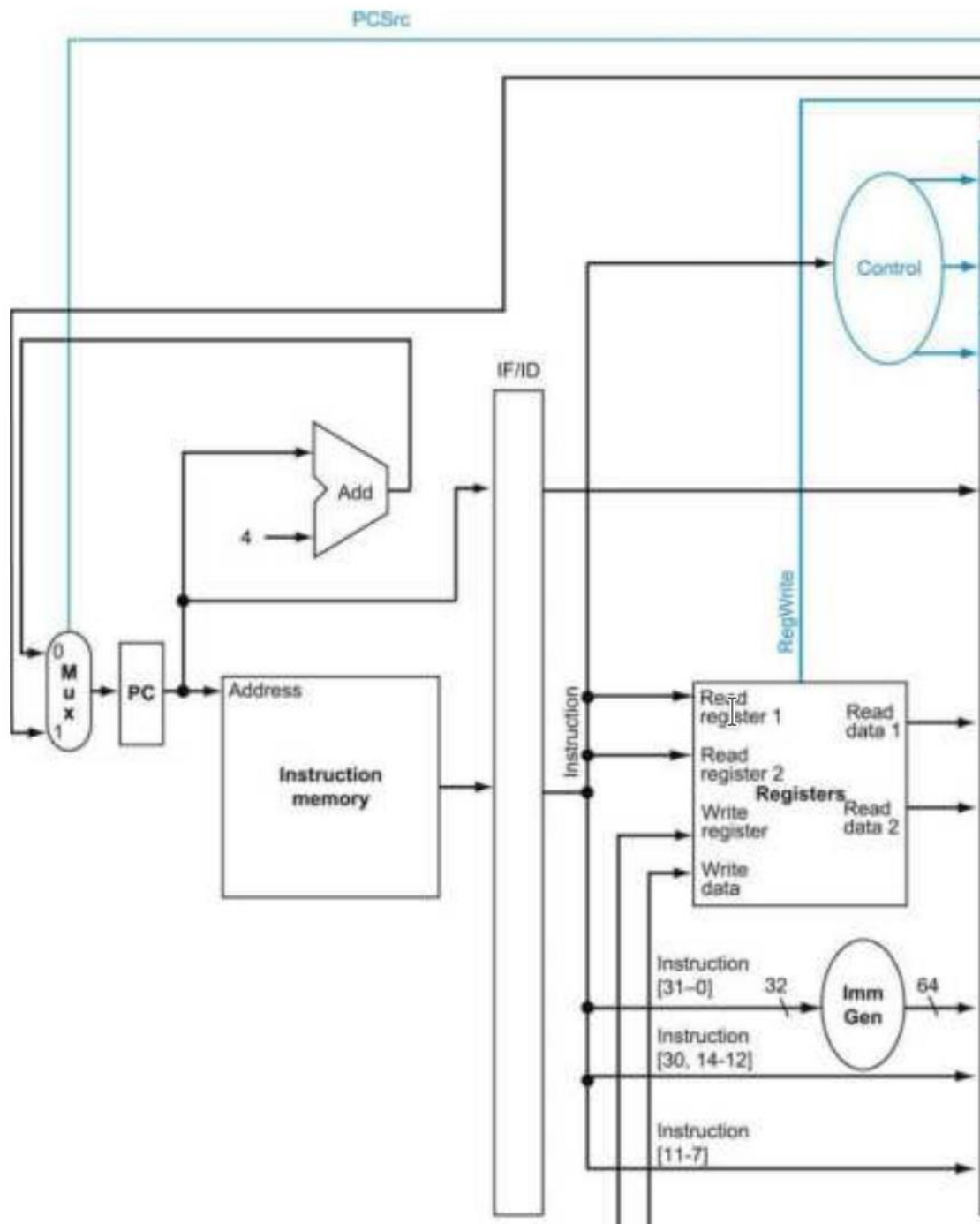


Fig. 2: Etapele de IF si ID (scopul lucrarii curente)

Cerințe:

1. Sa se implementeze urmatoarele module din IF:

a) Mux:

```
module mux2_1(input [31:0] ina, inb,  
             input sel,  
             output [31:0] out);
```

b) PC – un registru de tipul D. Semnalul **clk** și semnalul **res** sunt semnale unice la nivelul pipe-ului.

```
module PC(input clk,res,write,  
          input [31:0] in,  
          output reg [31:0] out);
```

c) Memorie Instrucțiuni - cu citire asincronă. În acest caz nu este nevoie de semnal **clk**. (hint: folosiți \$readmemh pt a încărca conținutul memoriei dintr-un fișier; pentru codificările instrucțiunilor și opcode-urile aferente, se va consulta documentul *riscv-spec-v2.2.pdf*, **paginile 104-107**):

```
module instruction_memory(input [9:0] address,  
                          output reg [31:0] instruction);
```

Exemplu de folosire a directivei \$readmemh pentru încărcarea conținutului fișierului *code.mem* în memoria **codeMemory** (codul este folosit în modulul *instruction_memory*):

```
reg [31:0] codeMemory [0:1023];  
  
initial $readmemh("code.mem", codeMemory);
```

De asemenea, în cazul de față cuvântul memoriei noastre este pe 32 de biți (o linie de memorie va cuprinde toți cei 32 de biți ai unei instrucțiuni). Procesorul nostru RISC-V știe că o instrucțiune este formată din 4 octeți, și astfel se așteaptă ca instrucțiunea următoare să se afle la 4 octeți de cea curentă (de aici și adunarea cu 4 a PC-ului).

O problemă ce apare în varianta propusă mai sus a memoriei, este că Verilog va accesa instrucțiunile din memorie la distanță de o linie, una de cealaltă (prima instrucțiune va fi pe prima linie, a doua instrucțiune pe cea de-a doua linie ș.a.m.d).

Pentru a rezolva această problemă, va trebui să shiftăm adresa noastră PC la dreapta de două ori ($PC \gg 2$) înainte de a extrage ultimii 10 biți pentru adresa instrucțiunii noastre. Motivul este următorul: adresa noastră PC va fi un multiplu de 4, din cauza adunarilor repetate cu 4, astfel pentru a trece de la o secvență de adrese multiple de 4, la o secvență de adrese consecutive, cu distanță 1, vom face o împărțire la 4 (sau o shiftare la dreapta cu 2).

Obs: Cunoașteți o metodă în Verilog pentru a extrage direct cei 10 biți de adresă corecți fără a folosi nici măcar shiftarea?

d) Sumator:

```
module adder(input [31:0] ina,inb,  
            output reg [31:0] out);
```

2. Sa se completeze modulul IF care conecteaza modulele definite la cerința 1 avand urmatoarele intrari si iesiri:

```
module IF(input clk, reset,
          input PCSrc, PC_write,
          input [31:0] PC_Branch,
          output [31:0] PC_IF, INSTRUCTION_IF);
```

3. Sa se implementeze si simuleze bancul de registri (Registers) din ID, avand urmatoarele intrari si iesiri:

```
module registers(input clk, reg_write,
                 input [4:0] read_reg1, read_reg2, write_reg,
                 input [31:0] write_data,
                 output [31:0] read_data1, read_data2);
```

Hint: Bancul de registri pentru procesorul RISC-V contine 32 de registri pe 32 de biti, notati de la x0 la x31 (x0 va fi mereu 0). Bancul reprezinta o memorie multi-port cu 2 porturi de citire pentru citirea celor 2 operanzi si un port de scriere unde va fi scris registrul rezultat.

Semnalul sincron **reg_write** activ pe 1 logic, declanseaza scrierea in bancul de registri la adresa data de **write_reg** a valorii **write_data**.

Citirea este asincrona si se realizeaza imediat cu schimbarea adreselor de citire **read_reg1** respectiv **read_reg2**, valorile citite aflandu-se in **read_data1**, respectiv **read_data2**

Pentru o mai buna simulare, folositi un ciclu for(intr-un bloc initial definit in modulul registers) in care initializati fiecare registru cu numere de la 0 la 31.

4. Sa se implementeze unitatea de generare a datelor imediate cu urmatoarea definitie a modulului:

```
module imm_gen(input [31:0] in,
               output reg [31:0] out);
```

Generarea valorilor imediate se va face pentru urmatoarele instructiuni:

- lw
- addi
- andi
- ori

- xori
- slti
- sltiu
- srli,srai
- slli
- sw
- beq,bne,blt,bge,bltu,bgeu

Hint: urmariti generarea valorii imediate specificata in tabelul de mai jos, in functie de tipul instructiunii: I,S,B,U,J(se va consulta documentul *riscv-spec-v2.2.pdf*, **pagina 12, Fig 2.4**):

| | | | | | | | | | | | |
|--------------|-------------|----|-------------|----|----------|-------------|-------------|----------|-------------|---|-------------|
| 31 | 30 | 20 | 19 | 12 | 11 | 10 | 5 | 4 | 1 | 0 | |
| — inst[31] — | | | | | | inst[30:25] | inst[24:21] | inst[20] | I-immediate | | |
| — inst[31] — | | | | | | inst[30:25] | inst[11:8] | inst[7] | S-immediate | | |
| — inst[31] — | | | | | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate | | |
| inst[31] | inst[30:20] | | inst[19:12] | | — 0 — | | | | | | U-immediate |
| — inst[31] — | | | inst[19:12] | | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate | | |

5. Sa se completeze modulul ID care conecteaza modulele definite la cerințele 3 si 4 avand urmatoarele intrari si iesiri:

```

module ID(input clk,
          input [31:0] PC_ID,INSTRUCTION_ID,
          input RegWrite_WB,
          input [31:0] ALU_DATA_WB,
          input [4:0] RD_WB,
          output [31:0] IMM_ID,
          output [31:0] REG_DATA1_ID,REG_DATA2_ID,
          output [2:0] FUNCT3_ID,
          output [6:0] FUNCT7_ID,
          output [6:0] OPCODE_ID,
          output [4:0] RD_ID,
          output [4:0] RS1_ID,
          output [4:0] RS2_ID);

```

6. Sa se implementeze registrul de pipeline ce leaga IF de ID
7. Sa se implementeze si simuleze un modul top care sa uneasca cele 2 module IF si ID cu urmatoarele intrari si iesiri, folosind fisierul *ID.v* dat:

```

module RISC_V_IF_ID(input clk, //semnalul de ceas global
input reset, //semnalul de reset global

//semnale provenite din stagii viitoare
//sunt pre-setate pentru aceasta lucrare
//vor fi discutate in lucrarile urmatoare
input IF_ID_write, //semnal de scriere pentru registrul de pipeline IF_ID
input PCSrc,PC_write, //semnale de control pentru PC
input [31:0] PC_Branch, //PC-ul calculat in etapa EX pentru instructiunile de salt
input RegWrite_WB, //semnal de activare a scrierii in bancul de registri
input [31:0] ALU_DATA_WB, //rezultatul calculat de ALU
input [4:0] RD_WB, //registrul rezultat in care se face scrierea

//semnale de iesire din ID
//vor fi vizualizate pe simulare
output [31:0] PC_ID, //adresa PC a instructiunii din etapa ID
output [31:0] INSTRUCTION_ID, //instructiunea curenta in etapa ID
output [31:0] IMM_ID, //valoarea calculata
output [31:0] REG_DATA1_ID, //valoarea primului registru sursa citit
output [31:0] REG_DATA2_ID, //valoarea celui de-al doilea registru sursa citit

output [2:0] FUNCT3_ID, //funct3 din codificarea instructiunii
output [6:0] FUNCT7_ID, //funct7 din codificarea instructiunii
output [6:0] OPCODE_ID, //opcode-ul instructiunii
output [4:0] RD_ID, //registru destinatie
output [4:0] RS1_ID, //registru sursa1
output [4:0] RS2_ID); //registru sursa2

```

8. Codificati instructiunile RISC-V urmatoare conform informatiilor din Fig. 3 (pentru codificarile instructiunilor si opcode-urile aferente, se va consulta documentul *riscv-spec-v2.2.pdf*, **paginile 104-107**):

add x2,x1,x0

addi x1,x1,1

and x3,x1,x2

ori x4,x1,1

sw x4,4(x5)

lw x12,8(x0)

beq x18,x0,5c

| | | | | | | | | | | | | | | | |
|------------|-----------|----|----|---------|------------|-----|--------|----|----------|---------|--------|--------|--------|--------|--|
| 31 | 30 | 25 | 24 | 21 | 20 | 19 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 0 | |
| funct7 | | | | rs2 | | rs1 | funct3 | | rd | | | opcode | | R-type | |
| imm[11:0] | | | | | | rs1 | funct3 | | rd | | | opcode | | I-type | |
| imm[11:5] | | | | rs2 | | rs1 | funct3 | | imm[4:0] | | | opcode | | S-type | |
| imm[12] | imm[10:5] | | | rs2 | | rs1 | funct3 | | imm[4:1] | imm[11] | opcode | | B-type | | |
| imm[31:12] | | | | | | | | | rd | | | opcode | | U-type | |
| imm[20] | imm[10:1] | | | imm[11] | imm[19:12] | | | rd | | | opcode | | J-type | | |

Fig. 3: Formatul instructiunilor aritmetice(R), imediate(I), salt conditionat(B), lucru cu memoria(S), salt neconditionat(J), upper-immediate(U) ale procesorului RISC-V

Pentru a verifica corectitudinea cerintelor, folositi fisierul de test oferit care ar trebui sa aiba urmatoarea simulare:

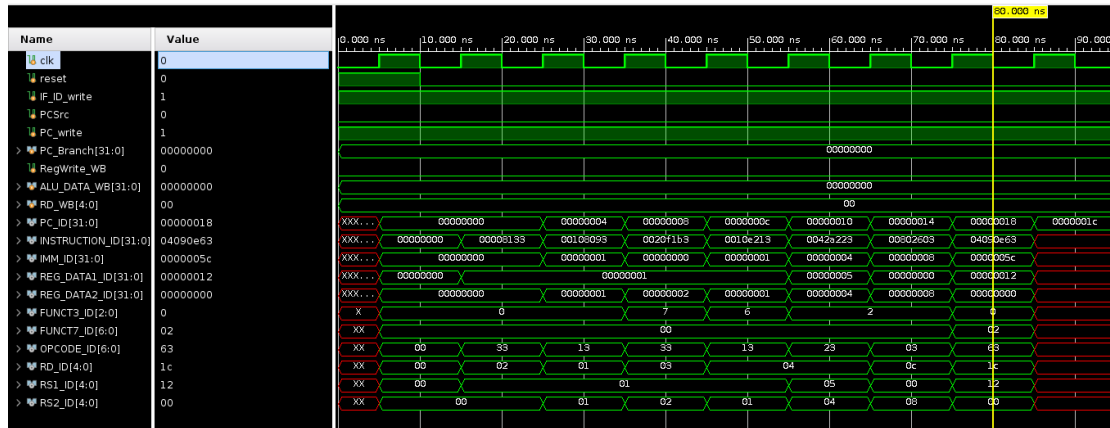


Fig. 4: Formele de unda rezultate in urma simularii instructiunilor specificate la cerinta 8, folosind fisierul de test dat

Bibliografie:

[1] David A. Patterson, John L. Hennessy, *Computer Organization and Design RISC-V edition*, 2018