Lucrarea 3

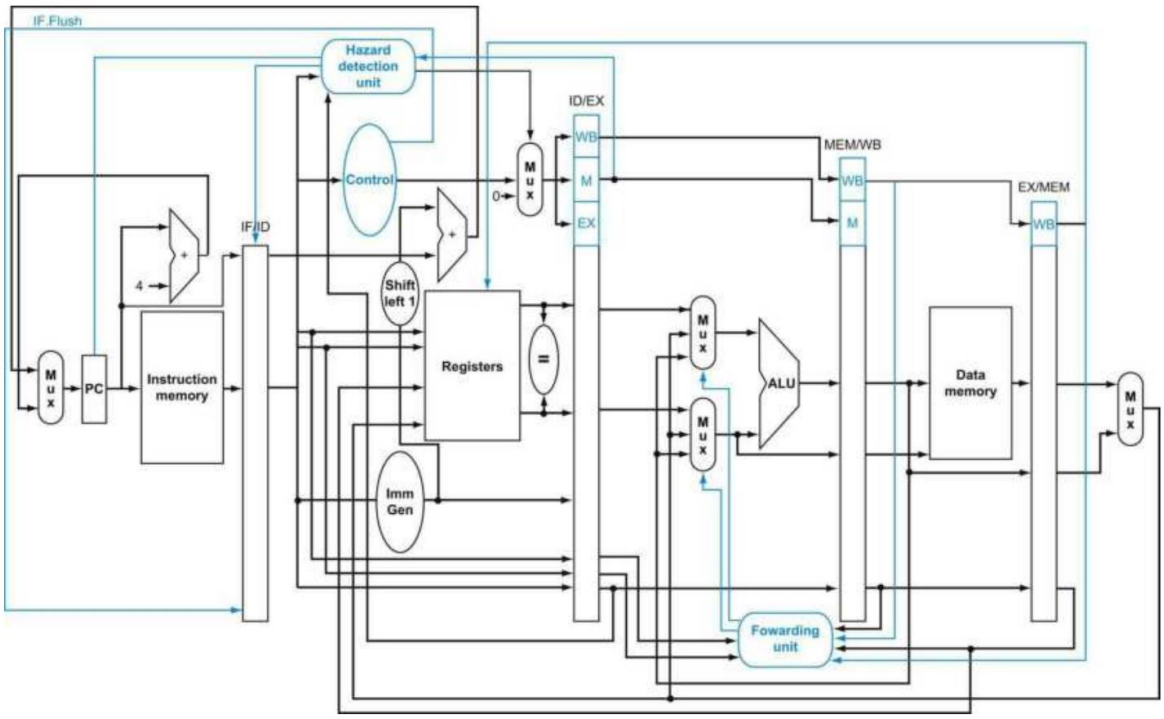


Fig. 1: Procesorul RISC-V – varianta pipeline

Lucrarea curentă își propune implementarea stagiilor EX, MEM, WB, plus două unități folosite în detecția hazardurilor și unitatea de forwarding, astfel realizându-se un prototip complet a procesorului RISC-V.

Vom începe cu implementarea unității de control din stagiul ID, care este responsabilă cu generarea semnalelor de comandă pentru întreg procesorul. Aceste semnale depind de tipul instrucțiunii decodificate(R-type, S-type, etc.), după cum poate fi observat în Fig. 2, iar acestea trebuie propagate până în stagiul în care sunt folosite după cum se prezintă în Fig. 3.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Fig. 2: Semnalele de control generate în funcție de instrucțiunea decodificată

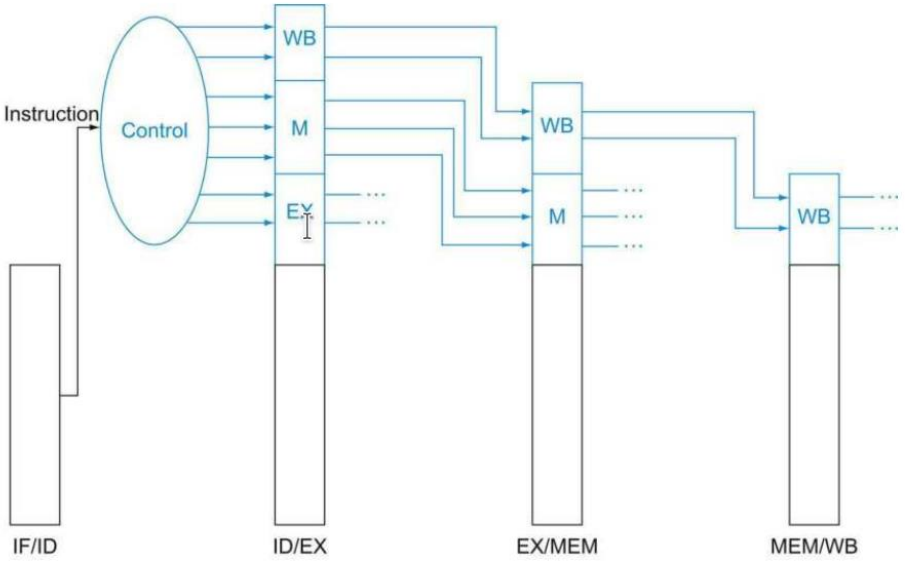


Fig. 3: Semnalele de control până în stagiile în care vor fi folosite

În stagiul EX, unitatea ALU se va ocupa de executarea propriu-zisa a operației aritmetico-logice din instrucțiunea curentă. Un prototip minimal de ALU este descris în Fig. 4, unde in funcție de câmpurile ***funct3***, ***funct7*** și ***AluOp*** (generat de unitatea de control), se generează semnalele de control corespunzătoare operației.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Fig. 4: Tabelul logic pentru generarea semnalelor de control a unității ALU pentru instrucțiunile AND, OR, ADD, SUB

Stagiul MEM se ocupă de scrierea/citirea în memoria de date (a cărei dimensiune se va considera ca fiind de 4KB), similară cu memoria de instrucțiuni prezentată în laboratorul 2. Aceasta suportă, pe lângă operația de citire asincronă, operația de scriere sincronă.

Stagiul WB se ocupă de scrierea rezultatului final în bancul de registre. Rezultatul final va fi selectat de un multiplexor; se va considera valoarea citită din memoria de date (dacă avem de a executăm instrucțiunea LW), sau valoarea calculată de ALU (dacă avem de executat o instrucțiune de tipul R-type sau I-type). Semnalul de control ***MemtoReg*** funcționează ca semnal de selecție pentru multiplexorul menționat.

Unitatea de Forwarding este necesară pentru soluționarea hazardurilor de tip RAW. Aceasta va face bypass la valorile calculate de ALU ce nu au fost încă scrise în bancul de registre, și se află încă în pipeline (în stagiul MEM sau WB). Aceasta este prezentată în Fig. 5, și generează semnalele ***ForwardA*** și ***ForwardB*** ce sunt necesare pentru alegerea operanzilor trimiși spre ALU (mai multe detalii de la pagina 569 din cartea prezentă în Bibliografie).

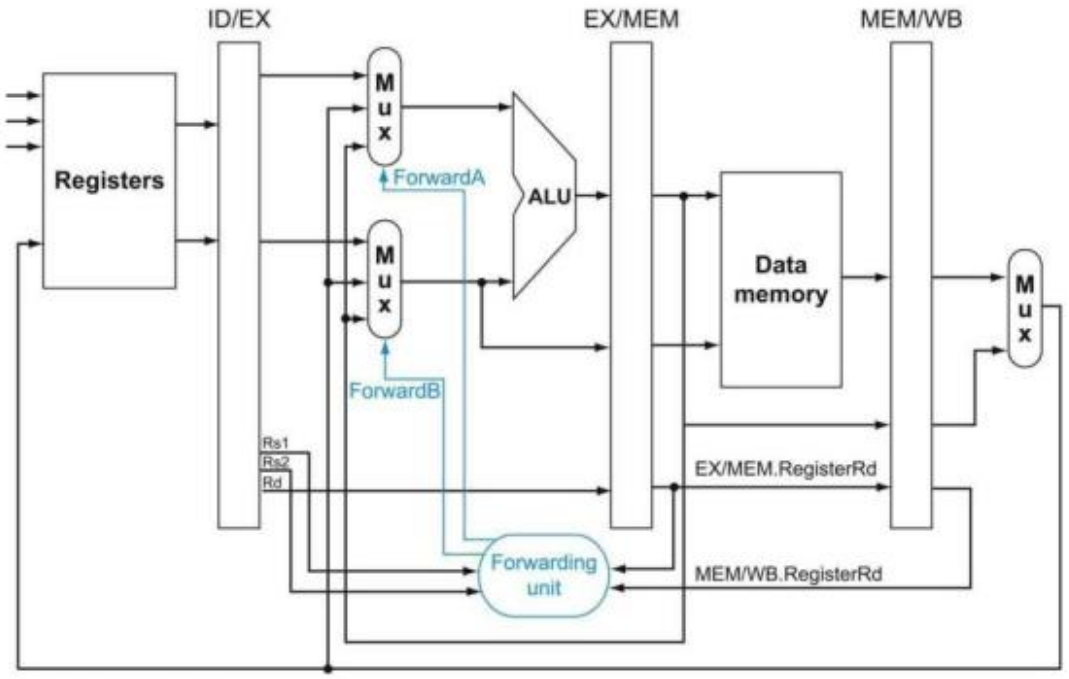


Fig. 5: Unitatea de Forwarding

Multiplexoarele din Fig. 5 selectează valorile registrelor sursă ce vor fi trimiși spre ALU (fie cei din stagiul ID, fie bypass-ate din stagiile MEM/WB). Multiplexorul nou introdus în diagramă selectează valoarea de input pentru ALU, având pe intrări o valoare imediata sau un registru. În Fig. 6 avem logica pentru a putea realiza operații atât între 2 registre, cât și între un registru și o valoare imediata.

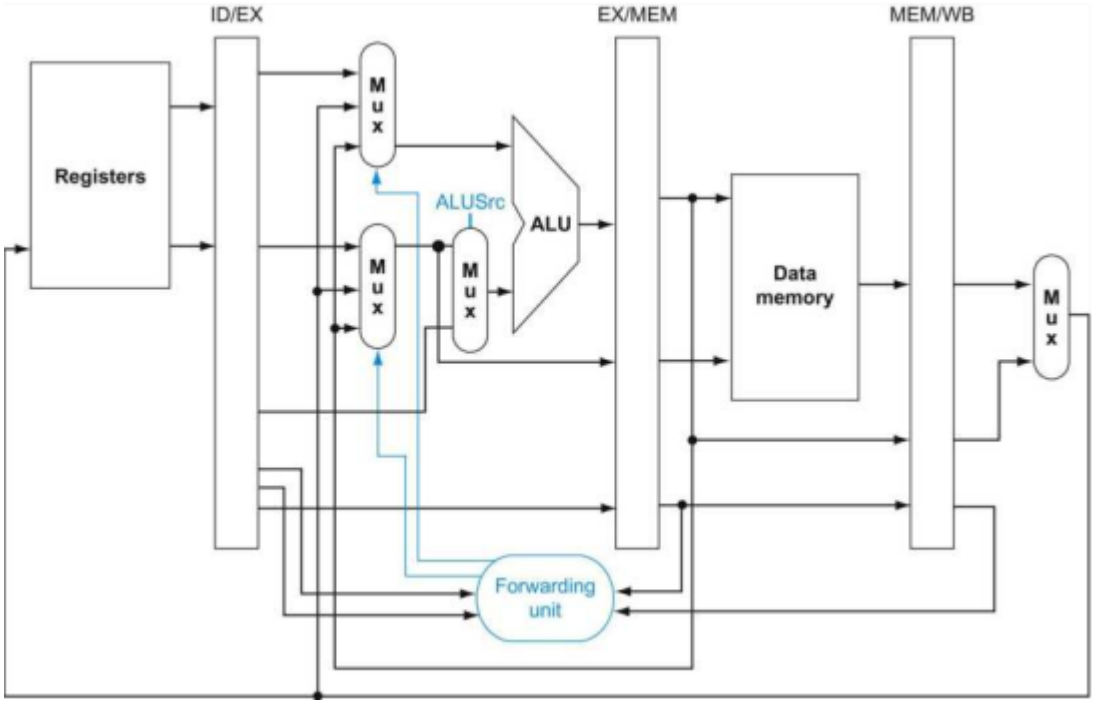


Fig. 6: Stagiul EX cu posibilitatea de a executa operații între registre și valori imediate.

Pentru a completa stagiul EX, este necesară o logică ce calculează adresa de salt pentru instrucțiunile de tip branch. Această adresă este calculată prin adunarea adresei furnizată de PC-ul curent al instrucțiunii și valoarea imediată codificată în aceasta. Fig. 7 prezinta această logică.

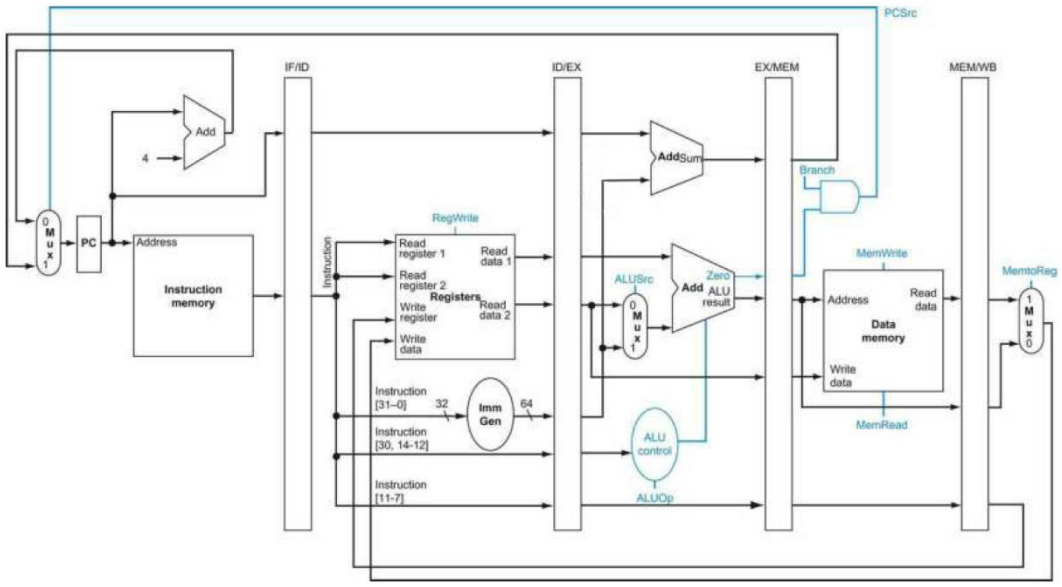


Fig. 7: Logica pentru calculul adresei de salt adăugată în etapa EX, al cărui rezultat este propagat în stagiul MEM al benzii de asamblare

Unitatea de detecție a hazardurilor(Hazard Detection Unit) inserează stall-uri în pipeline atunci când hazardul RAW nu poate fi rezolvat cu tehnica de forwarding (mai multe detalii de la pagina 582 din cartea prezentată în Bibliografie).

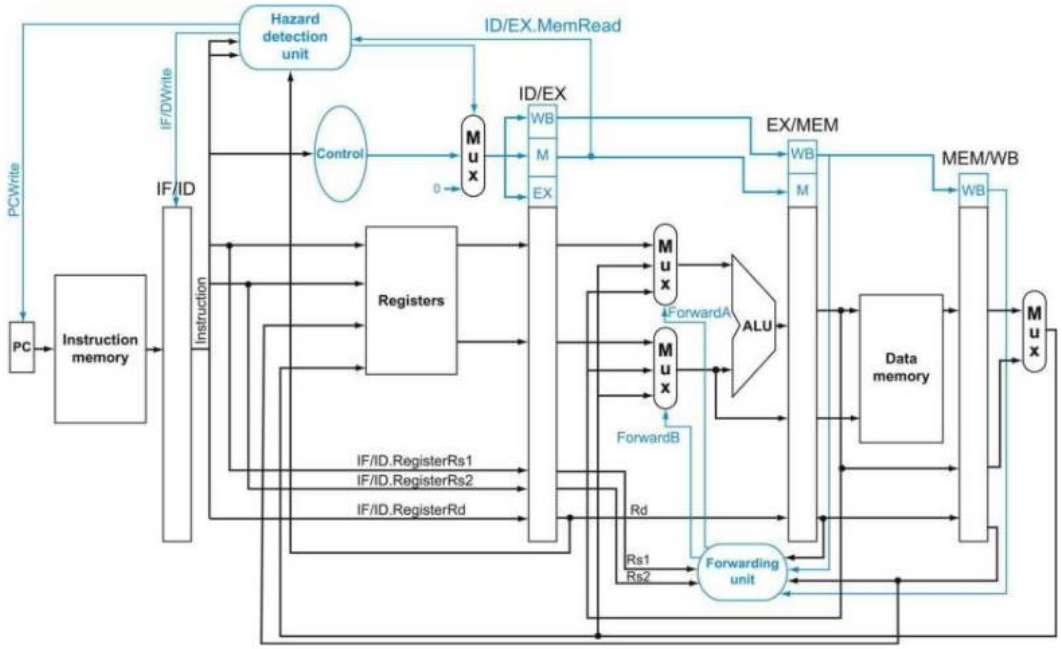


Fig. 8: Procesorul RISC-V complet cu unitățile de forwarding și detecție a hazardurilor

CERINȚE

1. Să se implementeze unitatea de control:

```
module control_path(input [6:0] opcode, //opcode-ul instructiunii
                    input control_sel,  //semnal provenit din unitatea de detectie a hazardurilor
                    output reg MemRead,MemtoReg,MemWrite,RegWrite,Branch,ALUSrc, //semnale de control
                    output reg [1:0] ALUop); //semnal de control pentru ALU
```

Semnalul ***control_sel*** provine din Hazard Detection Unit. Dacă acesta este 1 logic, toate semnalele de control din output vor fi 0 logic.

2. Sa se implementeze unitatea ALUcontrol:

```
module ALUcontrol(input [1:0] ALUop,
                  input [6:0] funct7,
                  input [2:0] funct3,
                  output reg [3:0] ALUinput);
```

Semnalul ALUinput pe 4 biți va desemna operația executată de ALU, în funcție de instrucțiunea curentă (se vor folosi câmpurile din instrucțiune ***funct3*** și ***funct7*** precum și semnalul de control ***ALUop*** pentru a identifica ce operație trebuie executată).

Se va folosi următorul tabel pentru generarea semnalului ALUinput în funcție de instrucțiune:

Instrucțiune	Cod ALUinput
ld, sd	0010
add	0010
sub	0110
and	0000
or	0001
xor	0011
srl, srli	0101
sll, slli	0100
sra, srai	1001
sltu	0111
slt	1000
beq, bne	0110
blt, bge	1000
bltu, bgeu	0111

3. Să se implementeze unitatea ALU:

```
module ALU(input [3:0] ALUop,      //ALUinput ce selecteaza operatia
           input [31:0] ina,inb,   //operanzii ce iau parte la operatie
           output zero,            //semnal ce verifica daca rezultatul este 0
           output reg [31:0] out); //rezultatul operatiei
```

4. Să se implementeze modulul EX ce cuprinde unitatea ALU, ALUcontrol, multiplexoarele necesare pentru alegerea operanzilor și sumatorul care calculează adresa de salt:

```
module EX(input [31:0] IMM_EX,         //valoarea immediate in EX
          input [31:0] REG_DATA1_EX,   //valoarea registrului sursa 1
          input [31:0] REG_DATA2_EX,   //valoarea registrului sursa 2
          input [31:0] PC_EX,          //adresa instructiunii curente in EX
          input [2:0] FUNCT3_EX,       //funct3 pentru instructiunea din EX
          input [6:0] FUNCT7_EX,       //funct7 pentru instructiunea din EX
          input [4:0] RD_EX,           //adresa registrului destinatie
          input [4:0] RS1_EX,          //adresa registrului sursa 1
          input [4:0] RS2_EX,          //adresa registrului sursa 2
          input RegWrite_EX,           //semnal de scriere in bancul de registri
          input MemtoReg_EX,           //...
          input MemRead_EX,            //semnal pentru activarea citirii din memorie
          input MemWrite_EX,           //semnal pentru activarea scrierii in memorie
          input [1:0] ALUop_EX,        //semnalul de control ALUop
          input ALUSrc_EX,             //semnal de selectie intre RS2 si valoarea imediata
          input Branch_EX,             //semnal de identificare a instructiunilor de tip branch
          input [1:0] forwardA,forwardB, //semnalele de selectie pentru multiplexoarele de forwarding

          input [31:0] ALU_DATA_WB,    //valoarea calculata de ALU, prezenta in WB
          input [31:0] ALU_OUT_MEM,    //valoarea calculata de ALU, prezenta in MEM

          output ZERO_EX,              //flag-ul ZERO calculat de ALU
          output [31:0] ALU_OUT_EX,    //rezultatul calculat de ALU in EX
          output [31:0] PC_Branch_EX,  //adresa de salt calculata in EX
          output [31:0] REG_DATA2_EX_FINAL //valoarea registrului sursa 2 selectata dintre
          );                               //valorile prezente in etapele EX, MEM si WB
```

5. Să se implementeze memoria de date din stagiul MEM:

```
module data_memory(input clk,
                   input mem_read,    //semnal de activare a citirii din memorie
                   input mem_write,   //semnal de activare a scrierii in memorie
                   input [31:0] address,    //adresa de scriere/citire
                   input [31:0] write_data, //valoarea scrisa in memorie
                   output reg [31:0] read_data //valoarea citita din memorie
                   );
```

6. Să se implementeze unitatea de detecție a hazardurilor:

```
module hazard_detection(input [4:0] rd, //adresa registrului destinatie in etapa EX
                        input [4:0] rs1, //adresa registrului sursa 1 decodificat in etapa ID
                        input [4:0] rs2, //adresa registrului sursa 2 decodificat in etapa ID
                        input MemRead,   //semnalul de control MemRead din etapa EX
                        output reg PCwrite, //semnalul PCwrite ce controleaza scrierea in registrul PC
                        output reg IF_IDwrite, //semnal ce controleaza scrierea in registrul de pipeline IF_ID
                        output reg control_sel); //semnal transmis spre unitatea de control
```

Unitatea de detecție a hazardurilor va fi implementata după următoarea logică:

```
if (ID/EX.MemRead and
  ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
   (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
   stall the pipeline
```

Semnalele de output vor fi puse în locul liniei “stall the pipeline” astfel încât în momentul inserării unui stall, registrul PC din IF și registrul de pipeline IF_ID să nu mai permită actualizarea valorilor interne până când hazardul nu este rezolvat, iar semnalul ***control_sel*** va seta semnalele de control generate de unitatea de control pe 0 logic.

7. Sa se implementeze unitatea de forwarding:

```
module forwarding(input [4:0] rs1,         //adresa registrului sursa 1 in etapa EX
                  input [4:0] rs2,         //adresa registrului sursa 2 in etapa EX
                  input [4:0] ex_mem_rd,   //adresa registrului destinatie in etapa MEM
                  input [4:0] mem_wb_rd,   //adresa registrului destinatie in etapa WB
                  input ex_mem_regwrite,   //semnalul de control RegWrite in etapa MEM
                  input mem_wb_regwrite,   //semnalul de control RegWrite in etapa WB
                  output reg [1:0] forwardA,forwardB); //semnalele de selectie a multiplexoarelor
                                                       //ce vor alege valoarea ce trebuie bypassata
```

Unitatea de forwarding va fi implementată după următoarea logică:

1. Pentru hazard în stagiul EX:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

2. Pentru hazard în stagiul MEM:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

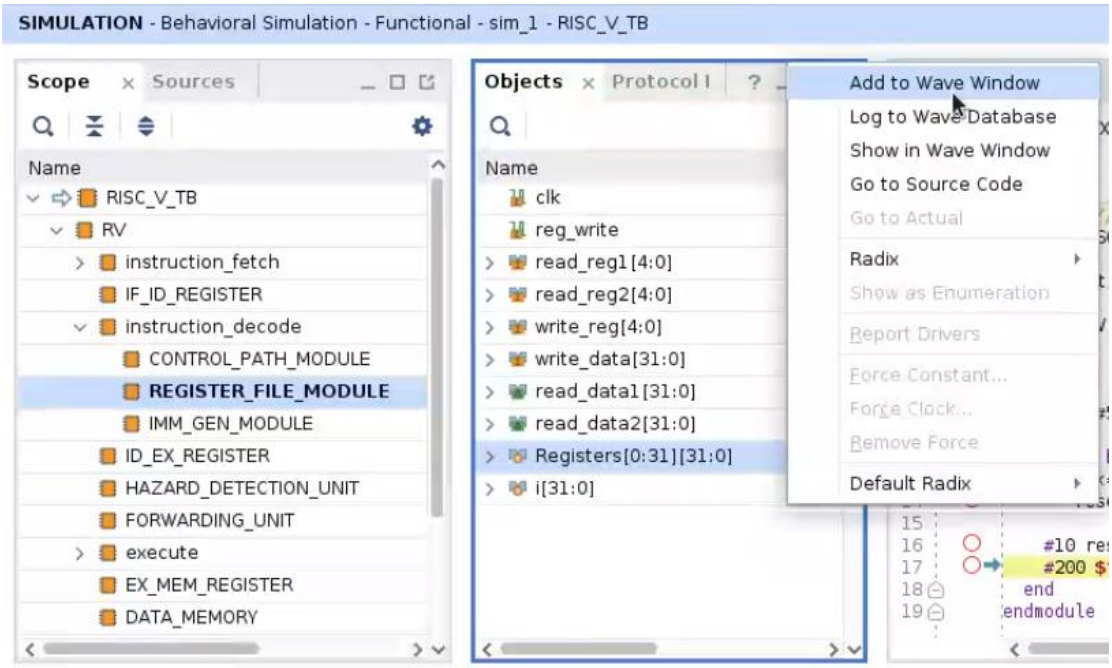
8. Să se implementeze registrele de pipeline dintre stagiile EX și MEM, respectiv MEM și WB

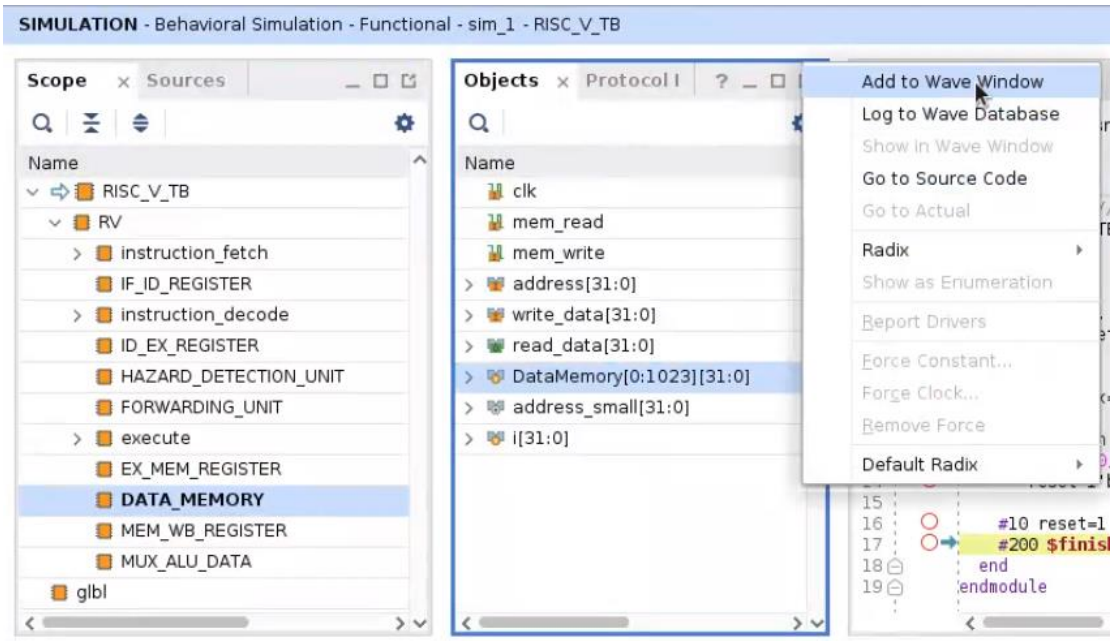
9. Să se implementeze microprocesorul RISC-V complet ce conține cele 5 stagii de pipeline și cele 2 module pentru detecția hazardurilor, respectiv unitatea de forwarding:

```
module RISC_V(input clk,   //semnalul de ceas global
              input reset, //semnalul de reset global

              output [31:0] PC_EX,        //adresa PC in etapa EX
              output [31:0] ALU_OUT_EX,   //valoarea calculata de ALE in etapa EX
              output [31:0] PC_MEM,       //adresa de salt calculata
              output PCSrc,               //semnal de selectie pentru PC
              output [31:0] DATA_MEMORY_MEM, //valoarea citita din memoria de date in MEM
              output [31:0] ALU_DATA_WB,     //valoarea finala scrisa in etapa WB
              output [1:0] forwardA, forwardB, //semnalele de forwarding
              output pipeline_stall            //semnal de stall la detectia de hazarduri
             );
```

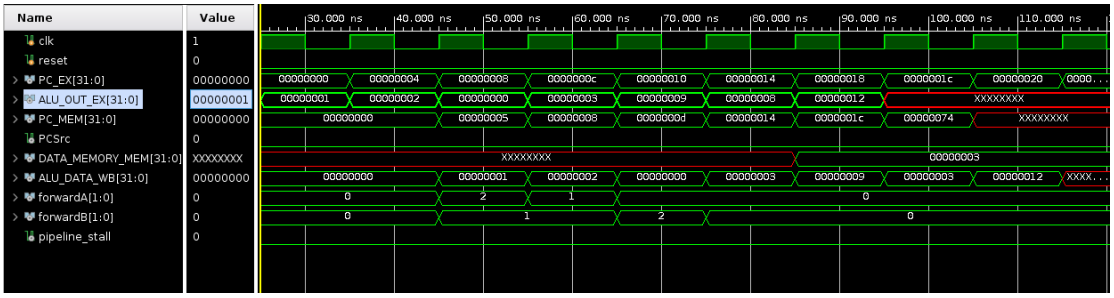
Pentru simulare se va adăuga bancul de registre și memoria de date în fereastra pentru formele de undă astfel:





Simularea se va face pe baza codului asamblare folosit in lucrarea precedenta, iar valorile finale rezultate în bancul de registre, respectiv memoria de date vor fi următoarele:

Simularea completa a modulului RISC-V:



Bibliografie:

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