CS224
Section No.: 2
Spring 2019
Lab No. 5
Munib Emre Sevilgen / 21602416

#part a
Program List:
#part d: PipeDtoE, PipeEtoM, datapath, HazardUnit, and mips modules are placed in the part d.

#part b

- Compute-use (data hazard):
  - Assume that there are 2 instructions as A and B. If the instruction B tries to read a source before instruction A writes it, B gets the old value that is incorrect. i.e, add writes $s0 then sub uses $s0 too.
  - Decode and execute stages are effected.
  - Solutions:
    - Inserting nops: Insert enough nops so that the instruction A finishes the writing back to the register.
    - Reordering the instructions: Reorder the instructions so that A finishes writing back to the register before the processor starts reading the register during instruction B.
    - Forwarding: The result of the instruction A can forwarded in execute stage or memory write stage to the instruction B's execute stage.
    - Stall: We can stall the process until A writes to the register.

- Load-use (data hazard):
  - Write back, execute and decode stages are effected. We can't forward the data directly, so we need to stall.
  - Solutions:
    - Inserting nops: Insert enough nops so that the load word instruction is finished.
    - Reorder the instructions: Reorder the instructions so that load word finishes its process before other instructions reads from the register.
    - Stall, flush and forward: The fetch and decode stages should be stalled and the contents of the execute stage should be cleared, so we can forward the correct data.

- Load-store (data hazard):
  - Before storing the data, load should start writing to the register.
  - This effects the decode and execute stages.
  - Solution:
    - Inserting nops
    - Reordering the instructions
    - Stall, flush and forward: The fetch and decode stages should be stalled and the contents of the execute stage should be cleared, so we can forward the correct data.

- J-type jump (control hazard):
  - Jump decision occurs in execution stage, which effects the upcoming fetch, decode stages. In this pipeline the jump instruction is not present.
  - Solution:

        - Stall, flush and forward: The fetch and decode stages should be stalled and the contents of the execute stage should be cleared while waiting for the jump decision, then forward the jump decision to the fetch stage.

- Branch (control hazard):
    - Branch not determined until memory stage of pipeline. Instructions after the branch are fetched before the calculation of the branch decision. These instructions must be flushed.
    - Therefore every stage until calculation of branch is effected.
    - Solutions:
        - Inserting nops: Insert enough nops so that branch finishes calculating
        - Stall, flush and forward: While waiting for the branch result, processor starts to do the upcoming instructions, if the branch happens then the instructions will get cleared, then we can forward the data to the fetch stage of the next instruction.

#part c

#StallF, StallD and FlushE

```
lwStall = ((rsD==rtE) || (rtD==rtE)) && MemToRegE;
StallF = lwStall;
StallD = lwStall;
FlushE = lwStall;
```

#ForwardBE

```
if((rtE!=0) && (rtE == WriteRegM) && RegWriteM)
          ForwardBE = 2'b10;
         else
                  if((rtE!=0) &&(rtE==WriteRegW) && RegWriteW)
                           ForwardBE = 2'b01;
                  else
                              ForwardBE = 2'b00;
```

#ForwardAE

```
if((rsE!=0) && (rsE == WriteRegM) && RegWriteM)
         ForwardAE = 2'b10;
          else
                  if((rsE!=0) &&(rsE==WriteRegW) && RegWriteW)
                           ForwardAE = 2'b01;
                  else
                           ForwardAE = 2'b00;
```

#ForwardAD

```
ForwardAD = (rsD !=0) && (rsD == WriteRegM) && RegWriteM;
```

#ForwardBD

```
ForwardBD = (rtD !=0) && (rtD == WriteRegM) && RegWriteM;
```

#part d

```
module PipeDtoE(input logic clr, clk, reset, RegWriteD, MemtoRegD, MemWriteD,
        input logic[2:0] AluControlD,
        input logic AluSrcD, RegDstD, BranchD,
        input logic[31:0] RD1D, RD2D,
        input logic[4:0] RsD, RtD, RdD,
        input logic[31:0] SignImmD,
        input logic[31:0] PCPlus4D,
            output logic RegWriteE, MemtoRegE, MemWriteE,
            output logic[2:0] AluControlE,
            output logic AluSrcE, RegDstE, BranchE,
            output logic[31:0] RD1E, RD2E,
            output logic[4:0] RsE, RtE, RdE,
            output logic[31:0] SignImmE,
            output logic[31:0] PCPlus4E);

always_ff @(posedge clk, posedge reset) begin
    // ****************************************************************************
    // YOUR CODE HERE
    // ****************************************************************************
    if(clr) begin
        RegWriteE <= 0;
        MemWriteE <= 0;
        BranchE <= 0;
        MemtoRegE <= MemtoRegD;
        AluSrcE <= AluSrcD;
        RegDstE <= RegDstD;
        RsE <= 0;
        RtE <= 0;
        RdE <= 0;
        AluControlE <= AluControlD;
        SignImmE <= SignImmD;
        RD1E <= RD1D;
        RD2E <= RD2D;
        PCPlus4E <= PCPlus4D;
    end
    else if (reset) begin
        RegWriteE <= 0;
        MemWriteE <= 0;
        BranchE <= 0;
        MemtoRegE <= 0;
        AluSrcE <= 0;
        RegDstE <= 0;
        RsE <= 0;
        RtE <= 0;
        RdE <= 0;
        AluControlE <= 0;
        SignImmE <= 0;
        RD1E <= 0;
        RD2E <= 0;
        PCPlus4E <= 0;
    end
    else begin
        RegWriteE <= RegWriteD;
        MemtoRegE <= MemtoRegD;
        MemWriteE <= MemWriteD;
```

```
            AluSrcE <= AluSrcD;
            RegDstE <= RegDstD;
            BranchE <= BranchD;
            RsE <= RsD;
            RtE <= RtD;
            RdE <= RdD;
            AluControlE <= AluControlD;
            SignImmE <= SignImmD;
            RD1E <= RD1D;
            RD2E <= RD2D;
            PCPlus4E <= PCPlus4D;
        end
    end
endmodule

module PipeEtoM(input logic clk, reset, RegWriteE, MemtoRegE, MemWriteE, BranchE,
Zero,
                input logic[31:0] ALUOut,
                input logic [31:0] WriteDataE,
                input logic[4:0] WriteRegE,
                input logic[31:0] PCBranchE,
                    output logic RegWriteM, MemtoRegM, MemWriteM, BranchM, ZeroM,
                    output logic[31:0] ALUOutM,
                    output logic [31:0] WriteDataM,
                    output logic[4:0] WriteRegM,
                    output logic[31:0] PCBranchM);

    always_ff @(posedge clk, posedge reset) begin
        // ******************************************************************************
        // YOUR CODE HERE
        // ******************************************************************************
        if (reset) begin
            ZeroM <= 0;
            ALUOutM <= 0;
            WriteDataM <= 0;
            WriteRegM <= 0;
            PCBranchM <= 0;
            RegWriteM <= 0;
            MemtoRegM <= 0;
            MemWriteM <= 0;
            BranchM <= 0;
        end
        else begin
            ZeroM <= Zero;
            ALUOutM <= ALUOut;
            WriteDataM <= WriteDataE;
            WriteRegM <= WriteRegE;
            PCBranchM <= PCBranchE;
            RegWriteM <= RegWriteE;
            MemtoRegM <= MemtoRegE;
            MemWriteM <= MemWriteE;
            BranchM <= BranchE;
        end
    end
```

```
endmodule

module PipeMtoW(input logic clk, reset, RegWriteM, MemtoRegM,
            input logic[31:0] ReadDataM, ALUOutM,
            input logic[4:0] WriteRegM,
                output logic RegWriteW, MemtoRegW,
                output logic[31:0] ReadDataW, ALUOutW,
                output logic[4:0] WriteRegW);

                always_ff @(posedge clk, posedge reset) begin
        //
********************************************************************************
                    // YOUR CODE HERE
        //
********************************************************************************
        if (reset) begin
            RegWriteW <= 0;
            MemtoRegW <= 0;
            ReadDataW <= 0;
            ALUOutW <= 0;
            WriteRegW <= 0;
        end
        else begin
            RegWriteW <= RegWriteM;
            MemtoRegW <= MemtoRegM;
            ReadDataW <= ReadDataM;
            ALUOutW <= ALUOutM;
            WriteRegW <= WriteRegM;
        end
    end
endmodule



// ******************************************************************************
// End of the individual pipe definitions.
// *****************************************************************************

// ******************************************************************************
// Below is the definition of the datapath.
// The signature of the module is given. The datapath will include (not limited to) the
following items:
//  (1) Adder that adds 4 to PC
//  (2) Shifter that shifts SignImmE to left by 2
//  (3) Sign extender and Register file
//  (4) PipeFtoD
//  (5) PipeDtoE and ALU
//  (5) Adder for PCBranchM
//  (6) PipeEtoM and Data Memory
//  (7) PipeMtoW
//  (8) Many muxes
//  (9) Hazard unit
//  ...?
// ******************************************************************************
```

```
module datapath (input  logic clk, reset,
                         input logic [31:0] PCF, instr,
                         input logic RegWriteD, MemtoRegD, MemWriteD,
                         input logic [2:0] ALUControlD,
                         input logic AluSrcD, RegDstD, BranchD,
                            output logic PCSrcM, StallD, StallF,
                            output logic[31:0] PCBranchM, PCPlus4F, instrD, ALUOut,
ResultW, WriteDataM);

        // ****************************************************************
        // Here, define the wires (logics) that are needed inside this pipelined datapath
module
   // You are given the wires connecting the Hazard Unit.
   // Notice that StallD and StallF are given as output for debugging
        // ****************************************************************

        logic ForwardAD, ForwardBD,  FlushE;
        logic [1:0] ForwardAE, ForwardBE;
        // Add necessary wires (logics).
   logic BranchM;
   logic ZeroM, ZeroE;
   logic [4:0] RsD, RtD, RdD, WriteRegW, RsE, RtE, RdE, WriteRegE, WriteRegM;
   logic [31:0] WriteDataE, ALUOutM, PCBranchE;
   logic [31:0] PCPlus4D, PCPlus4E, PC;
   logic [31:0] SignImmD, SignImmE, SrcBE, SrcAE, SignImmEsll2;
   logic ALUSrcD;
   logic [31:0] RD1D, RD2D, RD1E, RD2E, ReadDataM, ReadDataW;
   logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemWriteE, ALUSrcE,
RegDstE, BranchE,
          MemtoRegM, MemWriteM, MemtoRegW, ALUOutW;
   logic [2:0] ALUControlE;

        // ****************************************************************
        // Instantiate the required modules below in the order of the datapath flow.
        // ****************************************************************

        assign PCSrcM = BranchM && ZeroM;
        assign RsD = instrD[25:21];
   assign RtD = instrD[20:16];
   assign RdD = instrD[15:11];
   //assign WriteDataE = ;

        // Below, PipeFtoD and regfile instantiations are given
   // Add other instantiations
   // BE CAREFUL ABOUT THE ORDER OF PARAMETERS!
   imem imem1(PCF[7:2], instr);

   adder adder1(PCF,4,PCPlus4F);

        PipeFtoD pfd(instr, PCPlus4F, ~StallD, clk, reset || PCSrcM, instrD, PCPlus4D);

        signext sext1(instrD[15:0], SignImmD);
        regfile rf(clk, RegWriteW, instrD[25:21], instrD[20:16], WriteRegW, ResultW,
RD1D, RD2D);

        PipeDtoE pde(FlushE, clk, reset || PCSrcM, RegWriteD, MemtoRegD, MemWriteD,
ALUControlD, ALUSrcD,
                RegDstD, BranchD, RD1D, RD2D,
 RsD, RtD, RdD,

 SignImmD, PCPlus4D,
 RegWriteE,
                MemtoRegE, MemWriteE, ALUControlE, ALUSrcE, RegDstE, BranchE,
RD1E, RD2E,
            RsE, RtE, RdE,

 SignImmE, PCPlus4E);

   mux2 #(5)mux1(RtE, RdE, RegDstE, WriteRegE);

   mux2 #(32)mux2(WriteDataE, SignImmE, ALUSrcE, SrcBE);

   mux4 #(32)mux3(RD1E,ResultW,ALUOutM, 0, ForwardAE, SrcAE);

   mux4 #(32)mux4(RD2E,ResultW,ALUOutM, 0, ForwardBE, WriteDataE);

   alu alu1(SrcAE, SrcBE, ALUControlE, ALUOut, ZeroE, reset);

   sl2 sl21(SignImmE, SignImmEsll2);

   adder adder2(SignImmEsll2, PCPlus4E, PCBranchE);


   PipeEtoM pem(clk, reset || PCSrcM, RegWriteE, MemtoRegE, MemWriteE, BranchE,
ZeroE, ALUOut,
               WriteDataE, WriteRegE, PCBranchE,
 RegWriteM, MemtoRegM, MemWriteM,
               BranchM, ZeroM,
 ALUOutM, WriteDataM, WriteRegM, PCBranchM);

   dmem dmem1(clk, MemWriteM, ALUOutM, WriteDataM, ReadDataM);

   PipeMtoW pmw(clk, reset, RegWriteM, MemtoRegM, ReadDataM, ALUOutM,
WriteRegM,
               RegWriteW, MemtoRegW, ReadDataW, ALUOutW, WriteRegW);

   mux2 #(32)mux5(ALUOutW, ReadDataW, MemtoRegW, ResultW);

   mux2 #(32)mux6(PCPlus4F, PCBranchM, PCSrcM, PC);

   PipeWtoF pwf(PC, ~StallF, clk, reset, PCF);

   HazardUnit hazard(RegWriteW, WriteRegW, RegWriteM, MemtoRegM, WriteRegM,
RegWriteE,
               MemtoRegE, RsE, RtE, RsD, RtD, ForwardAE, ForwardBE, FlushE, StallD,
StallF);
```

```
endmodule



// Hazard Unit with inputs and outputs named
// according to the convention that is followed on the book.

module HazardUnit( input logic RegWriteW,
                input logic [4:0] WriteRegW,
                input logic RegWriteM,MemToRegM,
                input logic [4:0] WriteRegM,
                input logic RegWriteE,MemtoRegE,
                input logic [4:0] rsE,rtE,
                input logic [4:0] rsD,rtD,
                output logic [1:0] ForwardAE,ForwardBE,
                output logic FlushE,StallD,StallF);

    logic lwstall;
    assign lwstall = ((rsD == rtE) || (rtE == rtE)) && MemtoRegE;
    assign StallF = lwstall;
    assign StallD = lwstall;
    assign FlushE = lwstall;

    always_comb begin

        // ********************************************************************
        // Here, write equations for the Hazard Logic.
        // If you have troubles, please study pages ~420-430 in your book.
        // ********************************************************************

        if((rtE!=0) && (rtE == WriteRegM) && RegWriteM)
            ForwardBE = 2'b10;
        else
            if((rtE!=0) &&(rtE==WriteRegW) && RegWriteW)
                ForwardBE = 2'b01;
            else
                ForwardBE = 2'b00;

        if((rsE != 0) && (rsE == WriteRegM) && RegWriteM)
            ForwardAE = 2'b10;
        else
            if((rsE != 0) &&(rsE == WriteRegW) && RegWriteW)
                ForwardAE = 2'b01;
            else
                ForwardAE = 2'b00;

    end

endmodule


module mips (input  logic        clk, reset,
             output logic[31:0]  PCF,
            input  logic[31:0]  instr,
            output logic[31:0]  aluout, resultW,
            output logic[31:0]  instrOut, WriteDataM,
            output logic StallD, StallF);

    // *****************************************************************
    // You can change the logics below but if you didn't change the signitures of
    // above modules you will need these.
    // *****************************************************************

    logic memtoreg, zero, alusrc, regdst, regwrite, jump, PCSrcM, branch, memwrite;
    logic [31:0] PCPlus4F, PCm, PCBranchM, instrD;
    logic [2:0] alucontrol;
    assign instrOut = instr;

        // *****************************************************************
        // Below, instantiate a controller and a datapath with their new (if modified)
        // signatures and corresponding connections.
    // Also, you might want to instantiate PipeWtoF and pcsrcmux here.
    // Note that this is not the only solution.
    // You can do it in your way as long as it works.
        // *****************************************************************
    controller cnt(instrD[31:26], instrD[5:0], memtoreg, memwrite, alusrc, regdst,
                    regwrite, jump, alucontrol, branch);
        datapath dp(clk, reset, PCF, instr,        regwrite, memtoreg, memwrite,
alucontrol, alusrc,
                    regdst, branch, PCSrcM, StallD, StallF, PCBranchM, PCPlus4F, instrD,
aluout,
                    resultW, WriteDataM);

endmodule
```

#part e

- No Hazard

- Compute-use hazard:
  addi $t1, $t0, 6: RsD($t0) is equal to RtE($t0) at PipeDtoE.
  add $t2, $t1, $t0: RsD($t1) is equal to RtE($t1) at PipeDtoE.

- Load-use Hazard:
  add $t2, $t1, $a0: RsE($t1) is equal to WriteRegW($t1).

- Branch Hazard: The instructions below should be flushed.
  addi $t1, $zero, 5
  addi $t1, $t1, 6
  addi $t1, $zero, 8