CS224 Computer Organization
Preliminary Report

Lab 05

Section 2

Burak Ozturk

21901841

Part B:

Load-Use Hazard:

Type: Data hazard

Happens when there are two consecutive instructions such as first one is a load instruction and it's rt register is one second one's input registers. Since load instruction's result is not generated when second instruction needs it, this causes a hazard.

Affected pipeline stages: Execute stage and possibly Memory stage in case of memory write.

Solution: One solution is stalling until load instruction's result is available.

Load-Store Hazard:

Type: Data hazard

Happens when there are consecutive load and store instructions that has the same rt register. Since the data loaded by load instruction will be needed by store instruction before it is available, this will cause a hazard.

Affected pipeline stages: Memory stage of store instruction.

Solution: One solution is stalling until load instruction's result is available.

Branch Hazard:

Type: Control hazard

Happens when a branch that will normally cancel the three instructions that came after it is taken. This is because of when branch writes new address to PC register, that branch's following instructions will already be in pipeline illegally, following branch. This causes running instructions that are shouldn't be run, thus a hazard.

Affected pipeline stages: Three instructions that came after branch, if that branch is taken and should normally inherit those three instructions.

Solution: Moving branch decision to a earlier stage of pipeline can decrease the number of incorrectly fetched instructions. After that the solution is flushing incorrect instructions from pipeline registers.

Compute-Use Hazard:

Type: Data hazard

Happens when there are two consecutive instructions such as first one's output is one of second one's inputs. Since first instruction's result is not generated when second instruction needs it, this causes a hazard.

Affected pipeline stages: Forwarding first instruction's result to second instruction's Execute stage without having to wait first instruction's Writeback stage.

Solution: One solution is stalling until load instruction's result is available.

Part C:

Stalling and Flushing Logic:

```
lwstall = ( (rsE == rtD) | (rtD == rtE) ) & MemtoRegE

lwstall = StallF = StallD = FlushE
```

Forwarding Logic:

```
if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM)
      ForwardAE = 10
else if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)
      ForwardAE = 01
else
      ForwardAE = 00
```

Part D:

```
// Define pipes that exist in the PipelinedDatapath.
// The pipe between Writeback (W) and Fetch (F), as well as Fetch (F) and
Decode (D) is given to you.
// Create the rest of the pipes where inputs follow the naming conventions in
the book.


module PipeFtoD(input logic[31:0] instr, PcPlus4F,
                input logic EN, clk,        // StallD will be connected as this
EN
                output logic[31:0] instrD, PcPlus4D);

                always_ff @(posedge clk)
                    if(EN)
                        begin
                        instrD<=instr;
                        PcPlus4D<=PcPlus4F;
                        end

endmodule

// Similarly, the pipe between Writeback (W) and Fetch (F) is given as follows.

module PipeWtoF(input logic[31:0] PC,
                input logic EN, clk,        // StallF will be connected as this
EN
                output logic[31:0] PCF,
                input logic reset);

                always_ff @(posedge clk)
                    if (reset)
                        begin
                        PCF <= 32'b0;
                        end
                    else if(EN)
                        begin
                        PCF<=PC;
                        end

endmodule

//
*******************************************************************************
// Below, write the modules for the pipes PipeDtoE, PipeEtoM, PipeMtoW
yourselves.
// Don't forget to connect Control signals in these pipes as well.
//
*******************************************************************************


module PipeDtoE(
                      // Data inputs
                      input logic[31:0] RD1_D, RD2_D, PCPlus4D, SignImmD,
                      input logic[4:0] RsD, RtD, RdD,
                      // Control inputs
                      input logic RegWriteD, MemToRegD, MemWriteD, ALUSrcD, 
RegDstD, BranchD,
                      input logic[2:0] ALUControlD,
                              
                      // Control signals
                input logic CLR, clk,        // FlushE will be connected as this 
CLR
                              
                      // Data outputs
                output logic[31:0] RD1_E, RD2_E, PCPlus4E, SignImmE,
                      output logic[4:0] RsE, RtE, RdE,
                              
                      // Control outputs
                      output logic RegWriteE, MemToRegE, MemWriteE, ALUSrcE, 
RegDstE, BranchE,
                      output logic[2:0] ALUControlE);

                always_ff @(posedge clk)
                    if(CLR)
                        begin
                                    RD1_E <= 0;
                                    RD2_E <= 0;
                                    PCPlus4E <= 0;
                                    RsE <= 0;
                                    RtE <= 0;
                                    RdE <= 0;
                                    SignImmE <= 0;
                                             
                                    RegWriteE <= 0;
                                    MemToRegE <= 0;
                                    MemWriteE <= 0;
                                    ALUSrcE <= 0;
                                    RegDstE <= 0;
                                    BranchE <= 0;
                                    ALUControlE <= 0;
                        end
                            else
                                    begin
                                    RD1_E <= RD1_D;
                                    RD2_E <= RD2_D;
                                    PCPlus4E <= PCPlus4D;
                                    RsE <= RsD;
                                    RtE <= RtD;
                                    RdE <= RdD;
                                    SignImmE <= SignImmD;
                                             
                                    RegWriteE <= RegWriteD;
                                    MemToRegE <= MemToRegD;
                                    MemWriteE <= MemWriteD;
                                    ALUSrcE <= ALUSrcD;
                                    RegDstE <= RegDstD;
                                    BranchE <= BranchD;
                                    ALUControlE <= ALUControlD;
                                    end

endmodule
```

```
module PipeEtoM(
                        // Data inputs
                        input logic[31:0] ALUOutE, WriteDataE, PCBranchE,
                        input logic[4:0] WriteRegE,

                        // Control inputs
                        input logic RegWriteE, MemToRegE, MemWriteE, BranchE,
ZeroE,

                        // Control signals
                  input logic clk,      // FlushE will be connected as this CLR

                        // Data outputs
                  output logic[31:0] ALUOutM, WriteDataM, PCBranchM,
                        output logic[4:0] WriteRegM,

                        // Control outputs
                        output logic RegWriteM, MemToRegM, MemWriteM, BranchM,
ZeroM);

                  always_ff @(posedge clk)
                              begin
                              ALUOutM <= ALUOutE;
                              WriteDataM <= WriteDataE;
                              PCBranchM <= PCBranchE;
                              WriteRegM <= WriteRegE;
                              RegWriteM <= RegWriteE;
                              MemToRegM <= MemToRegE;
                              MemWriteM <= MemWriteE;
                              BranchM <= BranchE;
                              ZeroM <= ZeroE;
                              end

endmodule

module PipeMtoW(
                        // Data inputs
                        input logic[31:0] ReadDataM, ALUOutM,
                        input logic[4:0] WriteRegM,

                        // Control inputs
                        input logic RegWriteM, MemToRegM,

                        // Control signals
                  input logic clk,      // FlushE will be connected as this CLR

                        // Data outputs
                  output logic[31:0] ReadDataW, ALUOutW,
                        output logic[4:0] WriteRegW,

                        // Control outputs
                        output logic RegWriteW, MemToRegW);

                  always_ff @(posedge clk)
                              begin
                              ReadDataW <= ReadDataM;
```

```
                                ALUOutW <= ALUOutM;
                                WriteRegW <= WriteRegM;
                                RegWriteW <= RegWriteM;
                                MemToRegW <= MemToRegM;
                                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 RegWriteD, MemtoRegD, MemWriteD, AluSrcD, RegDstD,
BranchD,
                    input logic[2:0]  ALUControlD,

                    input logic StallF, StallD, FlushE, ForwardAE, ForwardBE,

                    output logic MemToRegE, RegWriteM,
                    output logic[5:0] opcode, funct,
                    output logic[4:0] RsD, RtD, RsE, RtE, WriteRegW);

      // ********************************************************************
      // Here, define the wires that are needed inside this pipelined datapath
module
      // ********************************************************************

      // Add the rest of the wires whenever necessary.

    // Fetch stage
      logic[31:0] PC, PCF, PCPlus4F, InstrF;

      // Decode stage
```

```
	logic[31:0] InstrD, PcPlus4D, RD1D, RD2D;

	// Execute stage
	logic       RegWriteE, MemWriteE, ALUSrcE, RegDstE, BranchE;
	logic[ 2:0] AluControlE;
	logic[31:0] RD1E, RD2E, PCPlus4E, SignImmE;

	// Memory stage
	logic       PCSrcM;

	// Writeback stage
	logic       RegWriteW;
	logic[31:0] ResultW, SignImmD;

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

	// Fetch
	mux2 #(32)	pcMux(PCPlus4F, PCBranchM, RegWriteW, PC);

	PipeWtoF	wfPipe(PC, StallF, clk, PCF, reset);

	imem		imem(PCF, InstrF);

	adder		pcAdder(PCF, 3'b100, PCPlus4F);

	// Decode
	PipeFtoD	fdPipe(InstrF, PcPlus4F, StallD, clk, InstrD, PcPlus4D);

	assign		opcode = InstrD[31:26];
	assign		funct = InstrD[5:0];

	regfile		rf(clk, RegWriteW, InstrD[25:21], InstrD[20:16],
WriteRegW, ResultW, RD1D, RD2D);

	signext		immext(InstrD[15:0], SignImmD);

	// Execute
	PipeDtoE	dePipe(RD1D, RD2D, PCPlus4D, SignImmD, InstrD[25:21],
InstrD[20:16], InstrD[15:11],
				RegWriteD, MemToRegD, MemWriteD, ALUSrcD, RegDstD,
BranchD, ALUControlD,
				FlushE, clk,
				RD1E, RD2E, PCPlus4E, SignImmE, RsE, RtE, RdE,
				RegWriteE, MemToRegE, MemWriteE, ALUSrcE, RegDstE,
BranchE, ALUControlE);

	mux4 #(32)	AluMuxA(RD1E, ResultW, ALUOutM, 32'b0, ForwardAE, SrcAE);
	mux4 #(32)	AluMuxB(RD2E, ResultW, ALUOutM, 32'b0, ForwardBE, SrcBE0);

	mux2 #(32)	Alu2MuxB(SrcBE0, SignImmE, ALUSrcE, SrcBE);

	alu		alu(SrcAE, SrcBE, ALUControlE, ALUOutE, ZeroE);

	mux2 #(5)	regMux(RtE, RdE, RegDstE, WriteRegE);
```

```
    sl2            brchShift(SignImmE, SignImmE2);

    adder          brchAdder(SignImmE2, PCPlus4E, PCBranchE);

    // Memory
    PipeEtoM       emPipe(ALUOutE, WriteDataE, PCBranchE, WriteRegE,
                            RegWriteE, MemToRegE, MemWriteE, BranchE, ZeroE,
                            clk,
                            ALUOutM, WriteDataM, PCBranchM, WriteRegM,
                            RegWriteM, MemToRegM, MemWriteM, BranchM, ZeroM);

    assign PCSrcM = BranchM & ZeroM;

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

    // Writeback
    PipeMtoW       mwPipe(ReadDataM, ALUOutM, WriteRegM,
                            RegWriteM, MemToRegM,
                            clk,
                            ReadDataW, ALUOutW, WriteRegW,
                            RegWriteW, MemToRegW);

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

endmodule



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

module HazardUnit(input logic MemtoRegE, RegWriteM, RegWriteW,
                  input logic[4:0] RsD, RtD, RsE, RtE, WriteRegM, WriteRegW,
                  output logic StallF, StallD, FlushE,
                  output logic[1:0] ForwardAE, ForwardBE);

   logic lwstall;
   always_comb begin

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

      lwstall = ( (RsE == RtD) | (RtD == RtE) ) & MemtoRegE;
      StallF = !lwstall;   StallD = !lwstall;   FlushE = lwstall;

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

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

    end
endmodule


module mips (input logic clk, reset);

      // ********************************************************************
      // Below, instantiate a controller and a datapath with their new (if
modified) signatures
      // and corresponding connections.
      // ********************************************************************

  controller        ctrl(opcode, funct, MemtoRegD, MemWriteD, AluSrcD, RegDstD,
RegWriteD, jump, ALUControlD, BranchD);

  datapath          dp(clk, reset,
                        RegWriteD, MemtoRegD, MemWriteD, AluSrcD, RegDstD,
BranchD,
                           ALUControlD,
                           StallF, StallD, FlushE, ForwardAE, ForwardBE,
                           MemToRegE, RegWriteM,
                           opcode, funct,
                           rsD, rtD, rsE, rtE, WriteRegW);

  HazardUnit            hz(MemtoRegE, RegWriteM, RegWriteW,
                        RsD, RtD, RsE, RtE, WriteRegM, WriteRegW,
                        StallF, StallD, FlushE,
                        ForwardAE, ForwardBE);
endmodule


// External instruction memory used by MIPS single-cycle
// processor. It models instruction memory as a stored-program
// ROM, with address as input, and instruction as output
// Modify it to test your own programs.

module imem ( input logic [5:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
      always_comb
         case ({addr,2'b00})                 // word-aligned fetch
//
//
      ************************************************************************
**
```

```
//    Here, you can paste your own test cases that you prepared for the part 1-
g.
//    Below is a program from the single-cycle lab.
//
      **********************************************************************
**
//
//          address          instruction
//          -------          -----------
            8'h00: instr = 32'h20020005;     // disassemble, by hand
            8'h04: instr = 32'h2003000c;     // or with a program,
            8'h08: instr = 32'h2067fff7;     // to find out what
            8'h0c: instr = 32'h00e22025;     // this program does!
            8'h10: instr = 32'h00642824;
            8'h14: instr = 32'h00a42820;
            8'h18: instr = 32'h10a7000a;
            8'h1c: instr = 32'h0064202a;
            8'h20: instr = 32'h10800001;
            8'h24: instr = 32'h20050000;
            8'h28: instr = 32'h00e2202a;
            8'h2c: instr = 32'h00853820;
            8'h30: instr = 32'h00e23822;
            8'h34: instr = 32'hac670044;
            8'h38: instr = 32'h8c020050;
            8'h3c: instr = 32'h08000011;
            8'h40: instr = 32'h20020001;
            8'h44: instr = 32'hac020054;
            8'h48: instr = 32'h08000012;     // j 48, so it will loop here
           default:  instr = {32{1'bx}};     // unknown address
         endcase
endmodule


//
      **********************************************************************
**
//    Below are the modules that you shouldn't need to modify at all..
//
      **********************************************************************
**

module controller(input  logic[5:0] op, funct,
                  output logic     memtoreg, memwrite,
                  output logic     alusrc,
                  output logic     regdst, regwrite,
                  output logic     jump,
                  output logic[2:0] alucontrol,
                  output logic branch);

   logic [1:0] aluop;

   maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
         jump, aluop);

   aludec  ad (funct, aluop, alucontrol);

endmodule
```

```
// External data memory used by MIPS single-cycle processor

module dmem (input  logic        clk, we,
             input  logic[31:0]  a, wd,
             output logic[31:0]  rd);

   logic  [31:0] RAM[63:0];

   assign rd = RAM[a[31:2]];    // word-aligned  read (for lw)

   always_ff @(posedge clk)
     if (we)
       RAM[a[31:2]] <= wd;      // word-aligned write (for sw)

endmodule

module maindec (input logic[5:0] op,
                   output logic memtoreg, memwrite, branch,
                   output logic alusrc, regdst, regwrite, jump,
                   output logic[1:0] aluop );
   logic [8:0] controls;

   assign {regwrite, regdst, alusrc, branch, memwrite,
                memtoreg,  aluop, jump} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000100; // R-type
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100010; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000001; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule

module aludec (input    logic[5:0] funct,
               input    logic[1:0] aluop,
               output   logic[2:0] alucontrol);
  always_comb
    case(aluop)
      2'b00: alucontrol  = 3'b010;  // add  (for lw/sw/addi)
      2'b01: alucontrol  = 3'b110;  // sub   (for beq)
      default: case(funct)          // R-TYPE instructions
          6'b100000: alucontrol  = 3'b010; // ADD
          6'b100010: alucontrol  = 3'b110; // SUB
          6'b100100: alucontrol  = 3'b000; // AND
          6'b100101: alucontrol  = 3'b001; // OR
          6'b101010: alucontrol  = 3'b111; // SLT
          default:   alucontrol  = 3'bxxx; // ???
        endcase
    endcase
endmodule

module regfile (input    logic clk, we3,
                input    logic[4:0]  ra1, ra2, wa3,
                input    logic[31:0] wd3,
                output   logic[31:0] rd1, rd2);

  logic [31:0] rf [31:0];

  // three ported register file: read two ports combinationally
  // write third port on rising edge of clock. Register0 hardwired to 0.

  always_ff @(negedge clk)
     if (we3)
         rf [wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;

endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucont,
           output logic [31:0] result,
           output logic zero);

    always_comb
        case(alucont)
            3'b010: result = a + b;
            3'b110: result = a - b;
            3'b000: result = a & b;
            3'b001: result = a | b;
            3'b111: result = (a < b) ? 1 : 0;
            default: result = {32{1'bx}};
        endcase

    assign zero = (result == 0) ? 1'b1 : 1'b0;

endmodule

module adder (input  logic[31:0] a, b,
              output logic[31:0] y);

     assign y = a + b;
endmodule

module sl2 (input  logic[31:0] a,
            output logic[31:0] y);

     assign y = {a[29:0], 2'b00}; // shifts left by 2
endmodule

module signext (input  logic[15:0] a,
                output logic[31:0] y);

  assign y = {{16{a[15]}}, a};    // sign-extends 16-bit a
endmodule

// parameterized register
module flopr #(parameter WIDTH = 8)
```

```
               (input logic clk, reset,
             input logic[WIDTH-1:0] d,
               output logic[WIDTH-1:0] q);

  always_ff@(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule


// paramaterized 2-to-1 MUX
module mux2 #(parameter WIDTH = 8)
              (input  logic[WIDTH-1:0] d0, d1,
               input  logic s,
               output logic[WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux4 #(parameter WIDTH = 8)
              (input  logic[WIDTH-1:0] d00, d01, d10, d11,
               input  logic[1:0] s,
               output logic[WIDTH-1:0] y);

   assign y = s[1] ? (s[0] ? d11 : d10) : (s[0] ? d01 : d00);
endmodule
```

Part E:

No hazards:

```
addi $s0, $zero, 0x4
addi $s1, $zero, 0x7
addi $s2, $zero, 0x13
addi $s3, $s0, 0x6
and $t0, $s0, $s1
or $t1, $s1, $s2
sub $t2, $s2, $s3
slt $v0, $s1, $s2
sw $t3, 0x44($t0)
lw $s3, 0x32($t1)
ori $s0, $s1, 0x44
```

Hex:

```
0x20100004
0x20100004
0x20120013
0x22130006
0x02114024
0x02324825
0x02535022
0x0232102A
0xAD0B0044
0x8D330032
0x36300044
```

Compute-use hazard:

```
addi $t0, $zero, 0x10
addi $t1, $t0, 0x0
add $t2, $t3, $t4
add $t3, $t1, $t0
add $t3, $t1, $t0
```

Hex:

```
0x20080010
0x21090000
0x016C5020
0x01285820
0x01285820
```

Load-use hazard:

```
addi $t0, $zero, 0x10
addi $t2, $zero, 0x10
ori $t3, $zero, 0x5
andi $t4, $zero, 0x5
sw $t2, 0x0($t0)
lw $t0, 0x0($t0)
add $t0, $t2, $t0
```

Hex:

```
0x20080010
0x200A0010
0x340B0005
0x300C0005
0xAD0A0000
0x8D080000
0x01484020
```

Branch hazard:

```
addi $s0, $zero, 0x4
addi $s1, $zero, 0x5
beq $s0, $s1, 0x1
addi $s2, $s2, 0x45

// s2 here should be 16
addi $s2, $s2, 0x10

// This loops on itself
beq $zero, $zero, 0xFFFFFFFF
```

Hex:

```
0x20100004
0x20110005
0x12110001
0x22520045
0x22520010
0x1000FFFF
```