PRELIMINARY REPORT

Lab 05

MUHAMMAD ARHAM KHAN

21701848 - CS

SECTION 06

SPRING 2019

Dated: 15 April, 2019

Part (b)

- **Branch Control Hazard:**

Branch hazard, a control hazard, occurs when we do not know what instruction is to taken into PC because we calculate the address during execution stage and provide it to PC after it is computed. The solution to this is us assuming that there is no branch and move on with the processing. Because if there is no branch instruction, the program continues execution but when there is a branch instruction, its address is calculated as to where it is to branch in memory stage and then flush the three instructions before that in fetch, decode and execute stages, and therefore our all stages are affected if the instructions are not flushed. After that this new instruction is executed at new address provided by branch. So, for flushing one signal is dispatched to the registers after fetch and decode from the hazard unit, and this calculates the next signals and sends them back to the datapath.

Part (c)

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

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


branchstall = BranchD && RegWriteE && (WriteRegE == rsD || WriteRegE == rtD)
      || BranchD && MemtoRegM && (WriteRegM == rsD || WriteRegM == rtD)


StallF = StallD = FlushE = (lwstall || branchstall)
```

Part (d)

```
`timescale 1ns / 1ps

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

    always_ff @(posedge clk, posedge reset)begin
        if (reset)begin
            instrD <= 0;
            PcPlus4D <= 0;
        end
        else if(EN)
            begin
            instrD<=instr;
            PcPlus4D<=PcPlus4F;
            end
        else begin
            instrD <= instrD;
            PcPlus4D <= PcPlus4D;
        end
    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, reset,          // StallF will be connected as
this EN
                output logic[31:0] PCF);

        always_ff @(posedge clk, posedge reset)begin
            if (reset)
                PCF <= 0;
            else if(EN)
                begin
                PCF<=PC;
                end
            else
                PCF <= PCF;
        end
endmodule


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
        if (reset || clr) begin
            RegWriteE <= 0;
            MemtoRegE <= 0;
            MemWriteE <= 0;
            AluSrcE <= 0;
            RegDstE <= 0;
            BranchE <= 0;
            AluControlE <= 0;
            RD1E <= 0;
            RD2E <= 0;
            RsE <= 0;
            RtE <= 0;
            RdE <= 0;
            SignImmE <= 0;
            PCPlus4E  <= 0;
        end
        else
            begin
                RegWriteE <= RegWriteD;
                MemtoRegE <= MemtoRegD;
                MemWriteE <= MemWriteD;
                AluSrcE <= AluSrcD;
                RegDstE <= RegDstD;
                BranchE <= BranchD;
                AluControlE <= AluControlD;
                RD1E <= RD1D;
                RD2E <= RD2D;
                RsE <= RsD;
                RtE <= RtD;
                RdE <= RdD;
                SignImmE <= SignImmD;
                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
        if (reset) begin
            RegWriteM <= 0;
            MemtoRegM <= 0;
            MemWriteM <= 0;
            BranchM <= 0;
```

```
            ZeroM <= 0;
            ALUOutM <= 0;
            WriteDataM  <= 0;
            WriteRegM <= 0;
            PCBranchM <= 0;
           end
        else begin
                RegWriteM <= RegWriteE;
                MemtoRegM <= MemtoRegE;
                MemWriteM <= MemWriteE;
                BranchM <= BranchE;
                ZeroM <= Zero;
                ALUOutM <= ALUOut;
                WriteDataM  <= WriteDataE;
                WriteRegM <= WriteRegE;
                PCBranchM <= PCBranchE;
            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
        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


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);

       logic ForwardAE, ForwardBE, FlushE, ALUSrcE;
       logic RegWriteE, MemtoRegE, MemWriteE;
       logic[2:0] ALUControlE;
```

```
       logic[31:0] RegWriteW, WriteRegW, RD1D, RD2D, SignImmD;
       logic[31:0] PCPlus4D;
       logic[4:0] RsD, RtD, RdD, RsE, RtE, RdE;
       logic AluSrcE, RegDstE, BranchE, ZeroE, RegWriteM, MemtoRegM, MemWriteM,
BranchM, ZeroM, MemtoRegW;
    logic[31:0] RD1E, RD2E, SignImmE, PCPlus4E, ALUOutM, SrcAE, SrcBE, WriteDataE,
ALUOutE, PCBranchE, ReadDataM, ReadDataW, ALUOutW;

    logic[4:0] WriteRegE, WriteRegM;

    adder add1( PCF, 32'd4, PCPlus4F);


       PipeFtoD ftd(instr, PCPlus4F, ~StallD, clk, reset, instrD, PCPlus4D);
       regfile rf (clk, RegWriteW, instrD[25:21], instrD[20:16],
            WriteRegW, ResultW, RD1D, RD2D);
    signext sign1( instrD[15:0], SignImmD);

    assign RsD = instrD[25:21];
    assign RtD = instrD[20:16];
    assign RdD = instrD[15:11];

    PipeDtoE dte( FlushE, clk, reset, 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 m21( RtE, RdE, RegDstE, WriteRegE);
    mux4 m41( RD1E, ResultW, ALUOutM, 32'b0, ForwardAE, SrcAE);
    mux4 m42( RD2E, ResultW, ALUOutM, 32'b0, ForwardBE, WriteDataE);
    mux2 m22( WriteDataE, SignImmE, ALUSrcE, SrcBE);

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

    adder add2( {SignImmE[29:0], 2'b00}, PCPlus4E, PCBranchE);

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

    assign PCSrcM = BranchM & ZeroM;

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

    PipeMtoW mtw(clk, reset, RegWriteM, MemtoRegM,
```

```
                                ReadDataM, ALUOutM,
                                WriteRegM,
                                RegWriteW, MemtoRegW,
                                ReadDataW, ALUOutW,
                                WriteRegW);
    mux2 m23( ReadDataW, ALUOutW, MemtoRegW, ResultW);

    HazardUnit( 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;

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

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

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

    // stalling the value of D
    assign FlushE = StallD;

endmodule


module top(input logic clk, reset,
```

```
            output logic[31:0] WriteDataM, dataadr, PCF, instr);

   logic [31:0] ResultW, instrOut, ALUOut;
   logic StallD,StallF;

    mips mips ( clk, reset, PCF, instr,
             ALUOut, ResultW,
             instrOut, WriteDataM,
             StallD, StallF);
    imem imem (PCF[7:2], instr);

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);


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

//     imem im( PCF[7:2], instr);

       controller control( 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);

       mux2 mux24( PCPlus4F, PCBranchM, PCSrcM, PCm);
    PipeWtoF wtf( PCm, ~StallF, clk, reset, PCF);


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
//
//            address             instruction
//            -------             -----------
              8'h00: instr = 32'h20020005;
              8'h04: instr = 32'h2003000c;
```

```
              8'h08: instr = 32'h2067fff7;
              8'h0c: instr = 32'h00e22025;
              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

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, input logic reset);
```

```
    always_comb begin
        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
        if(reset)
            result <= 0;
        end

    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

// paramaterized 4-to-1 MUX
```

```
module mux4 #(parameter WIDTH = 8)
             (input  logic[WIDTH-1:0] d0, d1, d2, d3,
              input  logic[1:0] s,
              output logic[WIDTH-1:0] y);

   assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
endmodule
```

Part (e)

LEGEND: **<XYZ>** means hazard

- Compute hazard

Since $t0 is not being updated entirely in previous cycle, so it causes the hazard.

addi $t0, $zero, 5	IF	DEC	EX	MEM	WB		
addi $t1, $t0, 6		IF	**<DEC>**	EX	MEM	WB	
add $t2, $t1, $t0			IF	DEC	EX	MEM	WB

- Load-use hazard

$t1 register causes the hazard as it is being utilized after it has been loaded in the previous instruction yet it is being called in the next instruction before it is written in the register memory.

addi $t0, $zero, 5	IF	DEC	EX	MEM	WB							
addi $t1, $zero, 6		IF	DEC	EX	MEM	WB						
addi $a0, $zero, 1			IF	DEC	EX	MEM	WB					
addi $a1, $zero, 2				IF	DEC	EX	MEM	WB				
sw $t0, 0($t1)					IF	DEC	EX	MEM	WB			
lw $t1, 1($t0)						IF	DEC	EX	MEM	WB		
add $t2, $t1, $a0							IF	**<DEC>**	EX	MEM	WB	
sub $t2, $t1, $a1								IF	DEC	EX	MEM	WB

- Branch hazard

Since the processor is clueless if it should take the branch or not, this hazard is caused as the processor proceeds to the next instruction and begins decoding it and when its calculated whether the branch was

required or not and if it is jumped, then flush it otherwise flush isn't required and then decides after EX of beq instruction has been executed.

addi $t1, $zero, 2	IF	DEC	EX	MEM	WB							
beq $zero, $zero, 2		IF	DEC	EX	MEM	WB						
addi $t1, $zero, 5			IF	DEC	EX	MEM	WB					
addi $t1, $t1, 6				IF	**<DEC>**	EX	MEM	WB				
addi $t1, $zero, 8					IF	DEC	EX	MEM	WB			
addi $a0, $zero, 0						IF	DEC	EX	MEM	WB		
addi $a1, $zero, 0							IF	DEC	EX	MEM	WB	
sw $t1, 0($zero)								IF	DEC	EX	MEM	WB