BILKENT UNIVERSITY
COMPUTER ENGINEERING
CS 224
COMPUTER ORGANIZATION

PRELIMINARY DESIGN REPORT
LAB 05

BERK YILDIZ

21502040
SECTION 4

06.12.2018

c)

RAW data dependencies can occur which is a data hazard. It causes wrong execution of the values when the next instruction uses a register which assigned by a value in previous instruction. For the next instruction the read value will be wrong for the specified register and this will cause wrong calculation in the program. This problem mostly occurs in the memory stage of the pipeline datapath. There can be three possible solutions for this problem. First one is stalling which is adding some nop instructions until the register's value become updated however it will cause performance problems. Reordering the instructions by software or hardware can be another solution. However the healthiest solution is forwarding the hardware. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage.

There are load-use dependencies which is a data hazard. The problem is considering the memory stage. Differently from the RAW dependencies load-use is not fixed by forwarding hardware. So when load is followed by use of the value in the next instruction, a 1-cycle stall is needed. This stall will cause a delay in all the stages.

For the j-type jump which is a control hazard, all jumps will fetch then flush the following instruction. Jumps not decode until ID, so one flush is needed. So there have to be replacement of nop instructions instead of inserting nop instructions which is zeroing the control bits for the instruction to be flushed (zeroed all bits in the IF/ID register, before decoding the control bits).

For the branches which are a control hazard, all branches will fetch the following two instructions, and have to flush them if the branch is taken. Because there is no early branch predictions we have to consider execute stage. For reducing the branch misprediction penalties, the branch prediction could be made earlier. So adding hardware to compute branch in ID stage by adding a comparator and and gate and 3x1 mux to ID timing path would provide early branch prediction. RAW data dependencies for branch instructions are solved with the same forwarding hardware that solves the other RAW dependencies.

d)

Forwarding Logic For *ForwardAE*

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)

then
ForwardAE = 10

else

if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)

then
ForwardAE = 01

else
ForwardAE = 00
```

Forwarding Logic For *ForwardBE*

```
if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM)

then
ForwardBE = 10

else

if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW)

then
ForwardBE = 01

else
ForwardBE = 00
```

Logic For Stalling

lwstall =

((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall

Forwarding logic (Control Hazard)

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

ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND
RegWriteM
```

Stalling Logic (Control Hazard)

```
branchstall = BranchD AND RegWriteE AND

(WriteRegE == rsD OR WriteRegE == rtD)

OR

BranchD AND MemtoRegM AND

(WriteRegM == rsD OR WriteRegM == rtD)

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

e)

```
//THE CHANGES THAT I MADE IN THE CODE ARE BOLD WRITTEN.
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);

                always_ff @(posedge clk)
                    if(EN)
                        begin
                        PCF<=PC;
                        end

endmodule
```

```
module PipeDtoE();

endmodule

module PipeEtoM();
           // to be filled
endmodule

module PipeMtoW();
           // to be filled
endmodule




module datapath (input  logic clk, reset, RegWriteW,
                    input  logic[2:0]  ALUControlD,
                    input logic BranchD,
                    input logic [31:0] pcPlus4D,
                    input logic [31:0] ResultW,
                    input logic [4:0] rsD,rtD,rdD,
                    input logic [15:0] immD,                    // Add
input-outputs if necessary
                    input logic [4:0] WriteRegW,

                    output logic RegWriteE,MemToRegE,MemWriteE,
                      output logic[31:0] ALUOutE, WriteDataE,
                      output logic [4:0] WriteRegE,
                      output logic [31:0] PCBranchE,
```

```
                        output logic pcSrcE);

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

      logic stallF, stallD,  ForwardAD, ForwardBD,  FlushE, ForwardAE,
ForwardBE, MemtoRegE, RegWriteM, RegWriteW;
                        // Wires for connecting Hazard Unit
                        //Add the rest of the wires whenever necessary.




      regfile     rf (clk, regwrite, instr[25:21], instr[20:16], rsE,
rtE, rsD, rtD, writeregW,
                    result, srca, writedata);


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 [2:0] ForwardAE,ForwardBE,
                   output logic FlushE,StallD,StallF

    );

    always_comb begin
           //Forwarding Logic For ForwardAE
           if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM)
           begin
           ForwardAE = 10
           end
           else
           begin
           if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)
           begin
           ForwardAE = 01
           end
           else
           begin
           ForwardAE = 00
           end
           end
```

```
//Forwarding Logic For ForwardBE
if ((rtE != 0) & (rtE == WriteRegM) & RegWriteM)
begin
ForwardBE = 10
end
else
begin
if ((rtE != 0) & (rtE == WriteRegW) & RegWriteW)
begin
ForwardBE = 01
end
else
begin
ForwardBE = 00
end
end

//Logic For Stalling

lwstall = ((rsD==rtE) | (rtD==rtE)) & MemtoRegE

StallF = StallD = FlushE = lwstall

//Forwarding logic (Control Hazard)
ForwardAD = (rsD !=0) & (rsD == WriteRegM) & RegWriteM

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

//Stalling Logic (Control Hazard)

branchstall = BranchD & RegWriteE &
```

```
            (WriteRegE == rsD | WriteRegE == rtD) |

            BranchD & MemtoRegM &

            (WriteRegM == rsD | WriteRegM == rtD)

            StallF = StallD = FlushE = (lwstall | branchstall)


    end
endmodule
```

```
module mips (input  logic        clk, reset,
             output logic[31:0]  pc,
             input  logic[31:0]  instr,
             output logic        memwrite,
             output logic[31:0]  aluout, resultW,
             output logic[31:0] instrOut,
             input  logic[31:0]  readdata);

  logic        memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump;
  logic [2:0]  alucontrol;
  assign instrOut = instr;


endmodule
```

```
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;     // 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
```

```
//Test for no hazards
8'h4c: instr = 32'h21280007;
8'h50: instr = 32'h02328020;
8'h54: instr = 32'h018d5820;
8'h58: instr = 32'h02327022;
8'h5c: instr = 32'h030fc020;

//Test for RAW Dependency
8'h60: instr = 32'h21280007;
8'h64: instr = 32'h010d6022;
8'h68: instr = 32'h010f7024;
8'h6c: instr = 32'h0119c025;
8'h70: instr = 32'h012dc027;

//Test for jump
8'h74: instr = 32'h21280007;
8'h78: instr = 32'h08000005;
8'h7c: instr = 32'h01Cf6822;
8'h80: instr = 32'h016c6020;
8'h84: instr = 32'h01cf5820;
8'h88: instr = 32'h02328020;
8'h8c: instr = 32'h02538820;
8'h90: instr = 32'h02329020;

//Test for branch
8'h94: instr = 32'h11090004;
8'h98: instr = 32'h012a4820;
8'h9c: instr = 32'h016c5020;
8'h100: instr = 32'h01ae6020;
8'h104: instr = 32'h08000000;
8'h108: instr = 32'h02328020;
```

```
            8'h10c: instr = 32'h02538820;
            8'h110: instr = 32'h02329020;
           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);

    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
```

g)

No Hazard		
Address (in decimal)	**Instruction**	**Machine Code**
0	addi $t0, $t1, 7	0x21280007
4	add $s0, $s1, $s2	0x02328020
8	add $t3, $t4, $t5	0x018D5820
12	sub $t6, $s1, $s2	0x02327022
16	add $t8, $t8, $t7	0x030FC020

Raw Dependency		
Address (in decimal)	**Instruction**	**Machine Code**
0	addi $t0, $t1, 7	0x21280007
4	sub $t4, $t0, $t5	0x010D6022
8	and $t6, $t0, $t7	0x010F7024
12	or $t8, $t0, $t9	0x0119C025
16	nor $t4, $t1, $t5	0x012DC027

Jump		
Address (in decimal)	**Instruction**	**Machine Code**
0	addi $t0, $t1, 7	0x21280007
4	j end	0x08000005
8	sub $t5, $t6, $t7	0x01CF6822
12	add $t4, $t3, $t4	0x016C6020
16	add $t3, $t6, $t7	0x01CF5820
20	end: add $s0, $s1, $s2	0x02328020
24	add $s1, $s2, $s3	0x02538820
28	add $s2, $s1, $s2	0x02329020

Branch		
Address (in decimal)	**Instruction**	**Machine Code**
0	loop: beq $t0, $t1, out	0x11090004
4	add $t0, $t1, $t2	0x012A4820
8	add $t2, $t3, $t4	0x016C5020
12	add $t4, $t5, $t6	0x01AE6020
16	j loop	0x08000000
20	out: add $s0, $s1, $s2	0x02328020
24	add $s1, $s2, $s3	0x02538820
28	add $s2, $s1, $s2	0x02329020

CS224
Section 4
Fall 2018
Lab 05
Berk Yıldız / 21502040