CS224

SECTION 03

LAB05 PRELIMINARY REPORT

AYBÜKE CEREN DURAN/21302686

b-) List of hazards that can occur are below:

**DATA HAZARDS:**

* **Compute-use hazard:**

**Reason for this hazard:** The data is not written into write back stage after the execute stage is completed. Therefore, wrong data which belongs to rd of previous instruction can be read by the current instruction which is in the decode stage.

**Affected stages of pipeline:** Decode stage of the instruction will retrieve wrong data (reason is above). Therefore, execute and writeback stages will be affected by using wrong data (retrieved in Decode stage) in their operations.

* **Load-Use hazard:**

**Reason for this hazard: :** Due to requiring data from memory, the instruction cannot perform reading from memory. As a result of this, current instruction cannot perform using the data in its execute stage because the data is not retrieved from memory by previous instruction.

**Affected stages of pipeline:**

- execute

- memory write ( these 2 stages will have 2 late clock cycle).

* **Load-Store hazard:**

**Reason for this hazard:**  This hazard occurs after the data is retrieved/loaded from the memory, and then the data is stored in memory.

**Affected stages of pipeline:** storing instruction will be affected because of storing wrong data.

**CONTROL HAZARD:**

* **Branch hazard:**

**Reason for this hazard:** branch decision is not done until the next instruction is retrieved from the inst. memory.

**Affected stages of pipeline:** this hazard will cause delay. Not necessary 3 instructions will be retrieved in case branch misprediction.

c-) Solutions of hazards and when the hazards occur are below:

**DATA HAZARDS:**

* **Compute-use hazard:**

**Reason for this hazard:** The data is not written into write back stage after the execute stage is completed. Therefore, wrong data which belongs to rd of previous instruction which is in the decode stage can be read by the next instruction.

**When this hazard occurs:**  We can consider add instruction as an example for this hazard. rd value of the add instruction is not computed because rs or rt of the previous instruction is not computed.

**Solution of this hazard:** we can forward the data into the next instruction’s execute stage. Stalling is also another solution for this hazard.

* **Load-Use hazard:**

**Reason for this hazard: :** Due to requiring data from memory, the instruction cannot perform reading from memory. As a result of this, next instruction cannot perform using the data in its execute stage because the data is not retrieved from memory by previous instruction.

**When this hazard occurs:**  When next instruction tries to retrieve data in memory which is loaded by previous instruction.

**Solution of this hazard:** Forwarding does not solve this problem. Stalling is a one type of solution where pipeline hold until data is available.

* **Load-Store hazard:**

**Reason for this hazard:**  This hazard occurs after the data is retrieved/loaded from the memory, and then the data is stored in memory.

**When this hazard occurs:**  Consecutive lw and sw instructions are being used with same rt register.

**Solution of this hazard:** Stalling is an efficient solution for allowing loading from memory done until next instruction retrieves data from same register at its decoder stage.

**CONTROL HAZARD:**

* **Branch hazard:**

**Reason for this hazard:** branch decision is not done until the next instruction is retrieved from the inst. memory.

**When this hazard occurs:** when a branch decision is required.

**Solution of this hazard:** Pipeline can be stalled for 3 cycles or with additional hardware branch decision can be done at earlier stage. Flushing the retrieved instructions can be considered to solve the issue of branch mispredictions.

d-)

FORWARDING: (for rt is identical with rs so there is no need to show it since it would be a duplicate of the lines written below)

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

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

StallF = StallD = FlushE = lwstall

FORWARDING (Decode stage):

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

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

Stalling logic:

*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)*

OR;

branchstall =

BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD) OR

BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)

load or a branch hazard:

StallF = StallD = FlushE = lwstall OR branchstall

e-)

module maindec (input logic[5:0] op,

                              output logic memtoreg, memwrite, branch,

                              output logic alusrc, regdst, regwrite, jump,

                              output logic[1:0] aluop );

   logic [11:0] controls; // the added control signals according to the table

   assign {regwrite, regdst, alusrc, branch, memwrite,

                memtoreg,  aluop, jump} = controls;

  always\_comb

    case(op)

      6'b000000: controls <= 12'b101000001X00; // R-type

      6'b100011: controls <= 12'b100100010000; // LW

      6'b101011: controls <= 12'b0XX101XX0000; // SW

      6'b000100: controls <= 12'b0XX010XX0100; // BEQ

      6'b001000: controls <= 12'b100100000000; // ADDI

      6'b000010: controls <= 12'b0XXXX0XXXX01; // J

      6'b011101: controls <= 12'b110X0010XX10; // JALR

      6'b011111: controls <= 12'b111001110000; // PUSH

      default:   controls <= 9'bxxxxxxxxx; // illegal op

    endcase

endmodule

module datapath (input  logic clk, reset, pcsrc, alusrc,

 input  logic regwrite, [1:0] memtoreg, [1:0] regdst, [1:0] jump,

                                 input  logic[2:0]  alucontrol,

                 output logic zero,

                                 output logic[31:0] pc,

                         input  logic[31:0] instr,

                 output logic[31:0] aluout, writedata,

                         input  logic[31:0] readdata);

  logic [4:0]  writereg;

  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch, srcaAdderResult;

  logic [31:0] signimm, signimmsh, srca, srcb, result;

  // next PC logic

  flopr #(32) pcreg(clk, reset, pcnext, pc);

  adder       pcadd1(pc, 32'b100, pcplus4);

  sl2         immsh(signimm, signimmsh);

  adder       pcadd2(pcplus4, signimmsh, pcbranch);

  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,

                      pcnextbr);

  mux3 #(32)  pcmux(pcnextbr, {pcplus4[31:28], // needed for push

                    instr[25:0], 2'b00},srca, jump, pcnext);

// register file logic

   regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writereg,

                   result, srca, writedata);

   adder     srcaAdder(srca, 32’b100, srcaAdderResult); // needed for push

   mux2 #(5)    wrmux (instr[20:16], instr[15:11], regdst, writereg);

   mux4 #(32)  resmux (aluout, readdata,pcplus4, srcaAdderResult, // needed for push

memtoreg, result);

   signext         se (instr[15:0], signimm);

  // ALU logic

   mux2 #(32)  srcbmux (writedata, signimm, alusrc, srcb);

   alu         alu (srca, srcb, alucontrol, aluout, zero);

endmodule

module alu(input  logic [31:0] a, b,

           input  logic [2:0]  alucont,

           output logic [31:0] result,

           output logic zero);

  // details of the model need to be

  // filled in by you, the designer !

// classic adder module

always @ (\*) begin

case (alucont)

3’b000: result <= (a&&b); //and

3’b001: result <= (a||b); //or

3’b010: result <= (a+b); //add

3’b100: result <= (a+~b); //add

3’b101: result <= (a-b); //subtract

3’b111: result <= if (a<b)

result <= 32’b1;

else

result = 32’b0;//slt

default: result <= 31’b0; //default

endcase

always @ (\*)

if (result==32’b0)

zero = 1;

else

zero = 0;

endmodule

module hazard\_handling(input  logic        clk, reset, ForwardAD, ForwardBD, ForwardAE, ForwardBE, writeregE, writeregM, writeregW

             input  logic[31:0]  instr,

             output logic       branch, memtoregE, regwriteE, memtoregM, regwriteM, regwriteW, stallF, stallD,  memwrite ForwardAD, ForwardBD, ForwardAE, ForwardBE, writeregE, writeregM, writeregW,

             output logic[31:0]  aluout, writedata,

             input  logic[31:0]  readdata)

//to be further implemented

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, writedata,

             input  logic[31:0]  readdata);

  logic        regwrite, pcsrc, zero, alusrc, [1:0] regdst, [1:0] memtoreg, [1:0] jump;

  logic [2:0]  alucontrol;

  controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,

                        alusrc, regdst, regwrite, jump, alucontrol);

  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,

                          alucontrol, zero, pc, instr, aluout, writedata, readdata);

endmodule

module controller(input  logic[5:0] op, funct,

                  input  logic     zero,

                  output logic[1:0]     memtoreg, memwrite, regdst, jump

                  output logic     pcsrc, alusrc, regwrite,

                  output logic[2:0] alucontrol);

   logic [1:0] aluop;

   logic       branch;

   maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,

                                 jump, aluop);

   aludec  ad (funct, aluop, alucontrol);

   assign pcsrc = branch & zero;

endmodule

f-)

Code with hazard

addi $s0, $s2,$s3

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5

Code without hazard

addi $s0, $s2,$s3

nop

nop

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5