BILKENT UNIVERSITY

CS 224

Preliminary Design Report

Lab 05

Sec 04

Alper Kandemir

21703062

3.12.21

B)

<u>Raw hazards:</u>

Load use hazard: Register value not yet written back to register file because the memory stage didn't end. So, the next instruction cannot read the new data.

lw $s0, 40($0)

and $t0, $s0, $s1

The lw instruction receives data from memory at the end of memory stage but the and instruction needs that data as a source operand.

Effected stages: Execude

Solution: There is no way to solve this hazard with forwarding. The alternative solution is to stall until the new data available.

Load-store hazard: If store word instruction that uses same rt register with load word instruction comes after lw this hazard occurs.

lw $s0, 40($0)

sw $s0, 40($0)

Effected stages: Memory

Solution: Stalling.

Compute-use:

This occurs when one instruction writes a register and subsequent instructions read this register. The subsequent register tries to read the register value before the WB stage is done.

Effected stages: In the decode stage wrong values are read, and in the WB stage wrong values are written.

Solution: Forwarding to new value of the register to other instructions execute stage.

add $s0, $s2, $s3

and $t0, $s0, $s1

The add instruction writes a result into $s0 in the first half of cycle 5. However, the and instruction reads $s0 on cycle 3, obtaining the wrong value.

Control Hazards:

Branch: It occurs when the branch label is taken. Whether the branch will be taken or not will be decided in the memory stage so, previous stages have other instructions values. It causes a delay.

Effected stages: F, D, E

Solution: Adding an equality comparison unit decode stage and flushing

```
beq $t1, $t2, loop

and $t0, $s0, $s1

…

…

loop:

slt $t3, $s2, $s3
```

The and instruction is flushed and the slt instruction is fetched. Now the branch misprediction penalty is reduced to only one instruction rather than three.

J-type jump: It occurs when the j is used.

Effected stages: F, D, E

Solution: adding some extra parts to check whether the instruction is jump or not so, we can update the pc earlier than old version, and flush the next instruction

```
j loop

and $t0, $s0, $s1

…

…

loop:

slt $t3, $s2, $s3
```

C)

Data Forwarding & Stalling Logic

Forwading:

```
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
```

Stall and flush:

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

StallF = StallD = FlushE = lwstall

Control Forwarding & Stalling Logic

Forwarding logic:

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)

D)asdads

```
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
```

```
// *****************************************************************************
// 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(input logic clk,
		input logic RegWriteD,
		input logic MemtoRegD,
		input logic MemWriteD,
		input logic [2:0] ALUControlD,
		input logic ALUSrcD,
		input logic RegDstD,
		input logic BranchD,
		input logic [31:0] rd1D,
		input logic [31:0] rd2D,
		input logic [4:0] rsD,
		input logic [4:0] rtD,
		input logic [4:0] rdD,
		input logic [31:0] signImmD,
		input logic [31:0] PCPlus4D,
		input logic CLR,
		output logic RegWriteE,
		output logic MemtoRegE,
		output logic MemWriteE,
		output logic [2:0] ALUControlE,
		output logic ALUSrcE,
output logic RegDstE,
output logic BranchE,
output logic [31:0] rd1E,
output logic [31:0] rd2E,
output logic [4:0] rsE,
output logic [4:0] rtE,
output logic [4:0] rdE,
output logic [31:0] signImmE,
output logic [31:0] PCPlus4E
);

always_ff @(posedge clk)
   if(CLR)
      begin

      RegWriteE   <= 0;
      MemtoRegE   <= 0;
      MemWriteE   <= 0;
      ALUControlE <= 0;
      ALUSrcE     <= 0;
      RegDstE     <= 0;
      BranchE     <= 0;
      rd1E        <= 0;
      rd2E        <= 0;
      rsE         <= 0;
      rtE         <= 0;
      rdE         <= 0;
      signImmE    <= 0;
      PCPlus4E    <= 0;
```

```
          end

        else
          begin

          RegWriteE    <=  RegWriteD;
          MemtoRegE    <=  MemtoRegD ;
          MemWriteE    <=  MemWriteD;
          ALUControlE  <=  ALUControlD;
          ALUSrcE      <=  ALUSrcD ;
          RegDstE      <=  RegDstD ;
          BranchE      <=  BranchD ;
          rd1E         <=  rd1D;
          rd2E         <=  rd2D;
          rsE          <=  rsD;
          rtE          <=  rtD;
          rdE          <=  rdD;
          signImmE     <=  signImmD;
          PCPlus4E     <=  PCPlus4D;

          end
endmodule




module PipeEtoM(input logic clk,
        input logic RegWriteE,
        input logic MemtoRegE,
        input logic MemWriteE,
        input logic BranchE,
input logic ZeroE,
input logic [31:0] ALUOutE,
input logic [31:0] WriteDataE,
input logic [4:0] WriteRegE,
input logic [31:0] PCBranchE,
output logic RegWriteM,
output logic MemtoRegM,
output logic MemWriteM,
output logic BranchM,
output logic 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)

    begin

    RegWriteM     <=  RegWriteE;
    MemtoRegM     <=  MemtoRegE ;
    MemWriteM     <=  MemWriteE;
    BranchM       <=  BranchE ;
    ZeroM         <=  ZeroE;
    ALUOutM       <=  ALUOutE;
    WriteDataM    <=  WriteDataE;
    WriteRegM     <=  WriteRegE;
    PCBranchM     <=  PCBranchE;
```

```
                end

endmodule



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

      always_ff @(posedge clk)

                begin

                RegWriteW    <=  RegWriteM;
                MemtoRegW    <=  MemtoRegM ;
                ReadDataW    <=  ReadDataM;
                ALUOutW      <=  ALUOutM ;
                WriteRegW    <=  WriteRegM;

                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, 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,
        input logic [4:0] WriteRegW,
        input  logic[31:0] instr,
        output logic RegWriteE,MemToRegE,MemWriteE,
                output logic[31:0] ALUOutE, WriteDataE,
                output logic [4:0] WriteRegE,
                output logic [31:0] PCBranchE,
                output logic pcSrcE);


// *********************************************************************
            logic RegWriteW,ALUSrcD,ALUSrcE,RegDstD,RegDstE;
            logic MemWriteD;
            logic RegWriteM,RegWriteD;
               logic[2:0]  ALUControlD,ALUControlE;
         logic BranchD,BranchE,BranchM;
         logic [31:0] pcPlus4D,PCPlus4E;
         logic [31:0] ResultW,ReadDataM,ReadDataW;
         logic [4:0] rsD,rtD,rdD,rsE,rtE,rdE;
         logic [15:0] immD;
         logic [4:0] WriteRegW;
logic RegWriteE, MemToRegE, MemWriteE, MemToRegM, MemtoRegD, MemWriteM,
MemtoRegW;
              logic[31:0] ALUOutM,ALUOutE,ALUOutW, WriteDataE;
              logic [4:0] WriteRegE,WriteRegM;
              logic [31:0]
PCBranchE,rd1D,rd2D,signImmD,signImmE,rd1E,rd2E,WriteDataM,PCBranchM;
```

```
        logic pcSrcE,ZeroE,ZeroM;

        logic EN;

        logic [31:0] PC,PCF,PcPlus4F,instrD;


// ********************************************************************


logic stallF, stallD,  ForwardAD, ForwardBD,  FlushE, ForwardAE, ForwardBE;            //
Wires for connecting Hazard Unit

// Add the rest of the wires whenever necessary.

  HazardUnit hazard(RegWriteW,WriteRegW, RegWriteM,MemToRegM, WriteRegM,
RegWriteE,MemToRegE,rsE,rtE,

        rsD,rtD, BranchD, ForwardAE,ForwardBE,ForwardAD,ForwardBD,
FlushE,StallD,StallF);

  PipeDtoE(clk,RegWriteD, MemtoRegD, MemWriteD, ALUControlD,ALUSrcD,RegDstD,
BranchD,

        rd1D, rd2D, rsD, rtD, rdD, signImmD,  PCPlus4D,CLR, RegWriteE,
MemtoRegE,MemWriteE, ALUControlE,ALUSrcE,

        RegDstE,BranchE, rd1E,rd2E,rsE,rtE,rdE,signImmE,PCPlus4E);

  PipeEtoM(clk,RegWriteE,MemtoRegE, MemWriteE, BranchE, ZeroE,
ALUOutE,WriteDataE,WriteRegE,PCBranchE,RegWriteM, MemtoRegM,MemWriteM,

        BranchM,ZeroM, ALUOutM,WriteDataM, WriteRegM,PCBranchM);

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

PipeWtoF(PC,EN, clk, PCF);

PipeFtoD(instr, PcPlus4F,EN, clk,instrD, PcPlus4D);

// ********************************************************************

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

// ********************************************************************


regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writeregW,                    //
Instantiated register file.

            result, srca, writedata);                                                // Add the
rest.
```

```
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,
                input logic BranchD,
                output logic [2:0] ForwardAE,ForwardBE,ForwardAD,ForwardBD,
                output logic FlushE,StallD,StallF

    );

    always_comb begin



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

    StallF <= StallD <= FlushE <= lwstall ;

    if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM)

      ForwardAE = 2;
```

```
else if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)

  ForwardAE = 1;

else
 ForwardAE = 0;

 if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM)

  ForwardBE = 2;

else if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW)

  ForwardBE = 1;

else
 ForwardBE = 0;



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);



    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 RegWriteW;
                  logic[2:0]  ALUControlD;
          logic BranchD;
          logic [31:0] pcPlus4D;
          logic [31:0] ResultW;
          logic [4:0] rsD,rtD,rdD;
          logic [15:0] immD;
          logic [4:0] WriteRegW;

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

```
  assign instrOut = instr;

// ***************************************************************************
// Below, instantiate a controller and a datapath with their new (if modified) signatures
// and corresponding connections.
// ***************************************************************************
    controller ctrl(instrD[31:26], instrD[5:0], RegWriteD, MemtoRegD, MemWriteD,
ALUControlD,ALUSrcD,
                      RegDstD,BranchD);
    datapath dp(clk, reset, RegWriteW,
ALUControlD,BranchD,pcPlus4D,ResultW,rsD,rtD,rdD,
         immD,WriteRegW,RegWriteE,MemToRegE,MemWriteE,ALUOutE,
WriteDataE,WriteRegE, PCBranchE,
                 pcSrcE);



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'h8C100028;             //lw $s0, 40($0)
8'h04: instr = 32'h02114020;             //add $t0, $s0, $s1
8'h08: instr = 32'h02904822;             //sub $t1, $s4, $s0
8'h0c: instr = 32'h02155024;     //and $t2, $s0, $s5
8'h10: instr = 32'h8C100028;      //lw $s0, 40($0)
8'h14: instr = 32'hAC100028;      //sw $s0, 40($0)
8'h18: instr = 32'h02538020;      //add $s0, $s2, $s3
8'h1c: instr = 32'h02114024;      //and $t0, $s0, $s1
8'h20: instr = 32'h02904825;      //or $t1, $s4, $s0
8'h24: instr = 32'h02155022;      //sub $t2, $s0, $s5
8'h28: instr = 32'h112A0004;      //beq $t1, $t2, 0x0000004
8'h2c: instr = 32'h02114024;      //  and $t0, $s0, $s1
8'h30: instr = 32'h20840001;      //addi  $a0, $a0,1
8'h34: instr = 32'h20A50001;      //addi  $a1, $a1,1
8'h38: instr = 32'h0253582A;      //  slt $t3, $s2, $s3
8'h3c: instr = 32'h08000004;      //J 0x0000004
8'h40: instr = 32'h02114024;      //and $t0, $s0, $s1
8'h44: instr = 32'h20840001;      //addi  $a0, $a0,1
8'h48: instr = 32'h20A50001;      //addi  $a1, $a1,1
8'h4c: instr = 32'h0253582A;      //slt $t3, $s2, $s3
```

```
//NO HAZARD
8'h50: instr = 32'h20840001;      //addi $a0, $a0,1
8'h54: instr = 32'h8C100028;      //lw $s0, 40($0)
8'h58: instr = 32'h018D5824;      //and $t3, $t4, $t5
8'h5c: instr = 32'h012A4025;      //or $t0, $t1, $t2
8'h60: instr = 32'h00A42820;      //add $a1, $a1, $a0
8'h64: instr = 32'h01F87022;      //sub $t6, $t7, $t8
8'h68: instr = 32'hAC030000;      //sw $v1, 0($0)
8'h6c: instr = 32'h112A0004;      //beq $t1, $t2, 0x0000004
     default:  instr = {32{1'bx}};          // unknown address
   endcase
endmodule
```

E)

Load-use hazard test:

lw $s0, 40($0)	0x8C100028
add $t0, $s0, $s1	0x 02114020
sub $t1, $s4, $s0	0x02904822
and $t2, $s0, $s5	0x02155024

Load-store hazard test:

lw $s0, 40($0)	0x8C100028
sw $s0, 40($0)	0xAC100028

Compute-use hazard test:

add $s0, $s2, $s3	0x02538020
and $t0, $s0, $s1	0x02114024
or $t1, $s4, $s0	0x02904825
sub $t2, $s0, $s5	0x02155022

Branch test:

beq $t1, $t2, 0x0000004	0x112A0004

and $t0, $s0, $s1	0x02114024
addi $a0, $a0,1	0x20840001
addi $a1, $a1,1	0x20A50001
loop:	
slt $t3, $s2, $s3	0x0253582A

Jump test:

J 0x0000004	0x08000004
and $t0, $s0, $s1	0x02114024
addi $a0, $a0,1	0x20840001
addi $a1, $a1,1	0x20A50001
loop:	
slt $t3, $s2, $s3	0x0253582A

No Hazard test:

addi $a0, $a0,1	0x20840001
lw $s0, 40($0)	0x8C100028
and $t3, $t4, $t5	0x018D5824
or $t0, $t1, $t2	0x012A4025
add $a1, $a1, $a0	0x00A42820
sub $t6, $t7, $t8	0x01F87022
sw $v1, 0($0)	0xAC030000
beq $t1, $t2, 0x0000004	0x112A0004