CS224

LAB 5

PRELIMINARY WORK

SECTION 3

MUSTAFA MERT GÜLHAN

22201895

<u>Part b)</u>

1) Data Hazards:

- Compute-use Hazard:

  This causes data hazard because the writeback operation does not complete until the next instruction needs the data before the writeback stage. Execute, Memory or Writeback stages are affected. Forwarding data to the execution stage from the previous instruction or stalling will solve the hazard.

- Load-use Hazard:

  This data hazard occurs when an instruction tries to read a register value when it is not read from previous lw instruction. Execute and memory stages are affected. Flushing the execute stage and stalling fetch, decode stages will solve the issue by repeating the flushed instruction in the next cycle by correct forwarded data this time.

- Load-store Hazard:

  This data hazard occurs when sw instruction tries to save a register into memory where the value of that register is not loaded by the previous lw instruction. Execute and memory stages are affected. Can be handled by stalling the microprocessor or forwarding the data to the memory stage.

2) Control Hazards

- Branch Hazard:

  This control hazard occurs when a branch instruction is called because branch occurs in the 4th stage of the pipeline and instructions after the branch will be fetched before the branch happens.Fetch and Decode stages are affected. Flushing the fetched instructions will solve the problem.

<u>**Part c)**</u>

- Forwarding to Execute from AE:

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

- Forwarding to Execute from BE:

```
if ((rtE!=0) && (rtE==WriteRegM) && RegWriteM)
    ForwardBE = 10
```

```
else
    if ((rtE != 0)&&(rtE==WriteRegW)&& RegWriteW)
        ForwardBE = 01
    else
        ForwardBE = 00
```

- Stalling calculation:

```
lwstall = ((rsD==rtE) || (rtD==rtE)) && MemtoRegE
StallF = lwstall
StallD = lwstall
FlushE = lwstall
```

- Forwarding to Decode:

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

- Stalling and Flushing:

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


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

Part d)

- <u>Test with no hazards:</u>

```
addi $t0, $zero, 5   # $t0 = 5
addi $t1, $zero, 10  # $t1 = 10
add $t2, $t0, $t1    # $t2 = $t0 + $t1 = 15
sub $t3, $t1, $t0    # $t3 = $t1 - $t0 = 5
sw $t2, 0($zero)     # Store $t2 to memory
lw $t4, 0($zero)     # Load from memory to $t4
```

- <u>Test for Compute-use:</u>

```
addi $t0, $zero, 5    # $t0 = 5
addi $t1, $zero, 10   # $t1 = 10
add $t2, $t0, $t1     # $t2 = 15
sub $t3, $t2, $t0     # $t3 = 10 (hazard here)

#if t3=10 this means t1,t2 value got correctly
to the last instruction and hazard is fixed
```

- <u>Test for Load-use:</u>

```
addi $t0, $zero, 10 # $t0 = 10
lw $t0, 0($zero) # Load value into $t0
add $t1, $t0, $zero # $t1 = $t0 + 0 (hazard
here)

#if t1 != 10 it means hazard is fixed
```

- <u>Test for Load-store:</u>

```
addi $t0, $zero, 10   # $t0 = 10
lw $t0, 0($zero)      # Load into $t0
```

```
sw $t0, 4($zero)      # Save $t0 (hazard here)
beq $zero, $zero, -1  # loop

#if $t0 != 10 in the last sw instruction the
hazard is fixed
```

- <u>Branch Hazard:</u>

```
addi $t0, $zero, 5    # $t0 = 5
addi $t1, $zero, 5    # $t1 = 5
addi $t2, $zero, 5    # $t2 = 5
beq  $t0, $t1, 1      # Branch if $t0 == $t1
addi $t2, $zero, 1    # skipped
addi $t2, $t2, 2      # $t2 = 5+2
beq $zero, $zero, -1  # loop

#if t2 is 7 after the last instruction branch
works correctly and hazard is fixed.
```

Part e)

```
module HazardUnit( input logic RegWriteW, BranchD,
                input logic [4:0] WriteRegW, WriteRegE,
                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,ForwardCE,
                output logic
FlushE,StallD,StallF,ForwardAD, ForwardBD
    );
    logic lwstall;
    logic branchstall;
    always_comb begin
     //Forward to Execute Stage Calculation for AE
            if ((rsE != 0) & (rsE == WriteRegM) &
RegWriteM)
                ForwardAE = 2'b10;
            else if ((rsE != 0) & (rsE == WriteRegW) &
RegWriteW)
                ForwardAE = 2'b01;
            else
                ForwardAE = 2'b00;
//Forward to Execute Stage Calculation for BE
            if ((rtE != 0) & (rtE == WriteRegM) &
RegWriteM)
                ForwardBE = 2'b10;
            else if ((rtE != 0) & (rtE == WriteRegW) &
RegWriteW)
                ForwardBE = 2'b01;
            else
                ForwardBE = 2'b00;
            //Stall calculation for Load-use or Load-store
hazards
            if (((rsD == rtE) | (rtD == rtE)) & MemToRegE)
                assign lwstall = 1;
            else
                assign lwstall = 0;

            //Checking Branch Hazard
            if ((BranchD & RegWriteE & ((WriteRegE == rsD)
| WriteRegE == rtD)) | (BranchD & MemToRegM & ((WriteRegM
== rsD) | (WriteRegM == rtD))))
                  assign branchstall = 1;
            else
                  assign branchstall = 0;

            //Stalling and flushing
```

```
              for branch
            if (lwstall | branchstall) begin
                    StallF = 1;
                    StallD = 1;
                    FlushE = 1;
            end
           //No Stall+Flush no Hazard Occurred
            else begin
                    StallF = 0;
                    StallD = 0;
                    FlushE = 0;
            end
    end
endmodule



// Clear will be FlushE

module PipeDtoE(input logic clk, clear, reset,
                input logic RegWriteD, MemtoRegD,
MemWriteD, ALUSrcD, RegDstD,
                input logic[2:0] ALUControlD,
                input logic[4:0] RsD, RtD, RdD,
                input logic[31:0] RD1, RD2, SignImmD,
                output logic[31:0] RD1E, RD2E, SignImmE,
                output logic[4:0] RsE, RtE, RdE,
                output logic RegWriteE, MemtoRegE,
MemWriteE, ALUSrcE, RegDstE,
                output logic[2:0] ALUControlE
                );

                always_ff @(posedge clk, posedge reset)
begin
                    if(clear || reset) begin
                       //If FlushE is enabled we’re
flushing the instruction
```

```
                        //clear will be Flush E
                        //Clearing the register to flush
                         RegWriteE <= 0;
                         MemtoRegE <= 0;
                         MemWriteE <= 0;
                         ALUControlE <= 0;
                         ALUSrcE <= 0;
                         RegDstE <= 0;
                         RD1E <= 0;
                         RD2E <= 0;
                         RsE <= 0;
                         RtE <= 0;
                         RdE <= 0;
                         SignImmE <= 0;
                    end
                    else begin
                        //If not flushing update the
                   register
                         RegWriteE <= RegWriteD;
                         MemtoRegE <= MemtoRegD;
                         MemWriteE <= MemWriteD;
                         ALUControlE <= ALUControlD;
                         ALUSrcE <= ALUSrcD;
                         RegDstE <= RegDstD;
                         RD1E <= RD1;
                         RD1E <= RD2;
                         RsE <= RsD;
                         RtE <= RtD;
                         RdE <= RdD;
                         SignImmE <= SignImmD;
                    end
                end
endmodule

module PipeEtoM(input logic clk, reset,
                input logic [31:0] ALUOutE,
                input logic [31:0] WriteDataE,
                input logic [4:0] WriteRegE,
                input logic RegWriteE, MemToRegE,
MemWriteE,
                output logic RegWriteM, MemWriteM,
MemToRegM,
                output logic [31:0] WriteDataM,
                output logic [31:0] ALUOutM,
                output logic [4:0] WriteRegM
                );
                //No specified input to register for
ensuring reset added
                always_ff @(posedge clk, posedge reset)
begin
                    if(reset) begin //safety reset
                        RegWriteM <= 0;
                        MemToRegM <= 0;
                        MemWriteM <= 0;
                        WriteDataM <= 0;
                        ALUOutM <= 0;
                        WriteRegM <= 0;
                    end
                    else begin //think it will always work
                         //saving new values to reg
                        RegWriteM <= RegWriteE;
                        MemToRegM <= MemToRegE;
                        MemWriteM <= MemWriteE;
                        WriteDataM <= WriteDataE;
                        WriteRegM <= WriteRegE;
                        ALUOutM <= ALUOutE;
                    end
                end
endmodule
```