Bilkent University

CS 224 - Section 1

Preliminary Design Report - Lab 5

Halil Arda Özongun

22202709

27 November 2024

Part B- Potential Hazards

1) Compute-use Hazard: Data Hazard

Pipeline Stages Affected: EX (Execute), MEM (Memory)
Reason of hazard: When there are two consecutive instructions (one instruction arrives before the other one ends), these instructions may try to use the same registers. In such a case, if the data has been processed, we will experience this hazard because data is not ready.
Solution: Forwarding

2) Load-use Hazard: Data Hazard

Pipeline Stages Affected: ID (Instruction Decode), EX
Reason of hazard: This hazard occurs because the load word instruction cannot prepare the data until it is retrieved from the data memory. By the end of the MEM stage pipeline previous depentent instructions must be stalled.
Solution: Forwarding, stalling, flushing

3) Jump Hazard: Control Hazard

Pipeline Stages Affected: ID

- **Reason of hazard:** This happens when a jump instruction changes the program's flow, but the pipeline has already started fetching subsequent instructions.
  Solution: Branch Prediction and stalling, flushing

4) Branch Hazard: Control Hazard

Pipeline Stages Affected: ID, EX

- **Reason of hazard:** This occurs when a branch instruction depends on a condition that is not yet resolved, leading to uncertainty in the program flow.
  Solution: Branch Prediction and stalling, flushing

Part C- Hazard Unit's Logical Expressions

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

if ((rtE != 0) && (rtE == WriteRegM) && RegWriteM):
        ForwardBE = 10
else if ((rtE != 0) && (rtE == WriteRegW) && RegWriteW):
        ForwardBE = 01
else:
        ForwardBE = 00

lwstall = ((rsD = = rtE) || (rtD = = rtE)) && MemtoRegE
ForwardAD = (rsD != 0) && (rsD == WriteRegM) && RegWriteM
ForwardBD = (rtD != 0) && (rtD == WriteRegM) && RegWriteM
branchstall = (BranchD && (RegWriteE && ((WriteRegE == rsD) ||
(WriteRegE== rtD))) || (MemToRegM && ((WriteRegM == rsD) || (WriteRegM == rtD))));
StallF = StallD = ( lwstall || branchstall);
FlushE = (lwstall || branchstall|| JumpD);
```

Part D- Test Programs in Mips

Test 1: No Hazards

Main:

```
lw $t0, 0($s0)       # 0x8C100000
add $t1, $s1, $s2    # 0x02328020
sw $s3, 4($s0)       # 0xAC130004
sub $t2, $s4, $s5    # 0x02954022
add $t3, $s6, $s7    # 0x02D78020
```

Test 2: Compute-use Hazard

main:

```
add $t0, $s1, $s2    # 0x02328020
sub $t1, $t0, $s3    # 0x010B4822
and $t2, $t0, $s4    # 0x010C5024
or  $t3, $t1, $s5    # 0x012D5825
```

Test 3: Load-use Hazard

main:

```
lw $t0, 0($s0)       # 0x8C100000
add $t1, $t0, $s1    # 0x01098820
sub $t2, $t0, $s2    # 0x010A9022
sw $t1, 4($s0)       # 0xAC110004
```

Test 4: Branch Hazard

main:

```
add $t0, $s1, $s2       # 0x02328020
beq $t0, $zero, target  # 0x11000002
add $t1, $s3, $s4       # 0x02748820
sub $t2, $s5, $s6       # 0x02B5A022
```

target:

```
        or $t3, $s7, $t0        # 0x02F05825
```

Test 5: Jump Hazard

```
main:
        add $t0, $s1, $s2    # 0x02328020
        j target             # 0x08000004
        add $t1, $s3, $s4    # 0x02748820
        sub $t2, $s5, $s6    # 0x02B5A022
target:
        or $t3, $s7, $t0     # 0x02F05825
```

Part E- Complete Pipeline Codes

- Hazard Unit

```
`timescale 1ns / 1ps
module HazardUnit(
            input logic branchD,
            input logic [4:0] WriteRegW, WriteRegM, WriteRegE,
            input logic RegWriteW, RegWriteM, RegWriteE, MemtoRegE, MemtoRegM,
            input logic [4:0] rsE,rtE,
            input logic [4:0] rsD,rtD,
            output logic ForwardAD,ForwardBD,
            output logic [2:0] ForwardAE,ForwardBE,
            output logic FlushE,StallD,StallF, lwstall, branchstall
   );

   always_comb begin
      lwstall = MemtoRegE & ( rtE == rsD | rtE == rtD );
      branchstall = (branchD & RegWriteE & ( WriteRegE == rsD | WriteRegE == rtD ))
                        |
                (branchD & MemtoRegM & ( WriteRegM == rsD | WriteRegM == rtD ));
      StallF = lwstall | branchstall;
      StallD = lwstall | branchstall;
      FlushE = lwstall | branchstall;
      ForwardAD = RegWriteM & ( rsD != 0 & rsD == WriteRegM );
      ForwardBD = RegWriteM & ( rtD != 0 & rtD == WriteRegM );

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

      if ( rtE != 0 & rtE == WriteRegM & RegWriteM ) begin
         ForwardBE = 2'b10;
      end
      else if ( rtE != 0 & rtE == WriteRegW & RegWriteW ) begin
         ForwardBE = 2'b01;
      end
      else begin
         ForwardBE = 2'b00;
      end
   end
endmodule
```

- PipeDtoE

```
`timescale 1ns / 1ps

module PipeDtoE(input logic clk, clear, reset,
                input logic RegWriteD, MemtoRegD, MemWriteD,
                input logic [2:0] ALUControlD,
                input logic ALUSrcD, RegDstD,
                input logic [31:0] Read1D, Read2D,
                input logic [4:0] RsD, RtD, RdD,
                input logic [31:0] SignImmD,
                output logic RegWriteE, MemtoRegE, MemWriteE,
                output logic [2:0] ALUControlE,
                output logic ALUSrcE, RegDstE,
                output logic [31:0] Read1E, Read2E,
                output logic [4:0] RsE, RtE, RdE,
                output logic [31:0] SignImmE
                );

    always_ff @(posedge clk or posedge reset) begin
        if (reset || clear) begin
            {RegWriteE, MemtoRegE, MemWriteE} <= '0;
            ALUControlE <= '0;
            {ALUSrcE, RegDstE} <= '0;
            {Read1E, Read2E} <= '0;
            {RsE, RtE, RdE} <= '0;
            SignImmE <= '0;
        end
        else begin
            RegWriteE <= RegWriteD;
            MemtoRegE <= MemtoRegD;
            MemWriteE <= MemWriteD;
            ALUControlE <= ALUControlD;
            ALUSrcE <= ALUSrcD;
            RegDstE <= RegDstD;
            Read1E <= Read1D;
            Read2E <= Read2D;
            RsE <= RsD;
            RtE <= RtD;
            RdE <= RdD;
            SignImmE <= SignImmD;
        end
    end

endmodule
```

- PipeEtoM

```
`timescale 1ns / 1ps
module PipeEtoM(input logic clk, reset,
                input logic RegWriteE, MemtoRegE, MemWriteE,
                input logic [31:0] ALUOutE, WriteDataE,
                input logic [4:0] WriteRegE,
                output logic RegWriteM, MemtoRegM, MemWriteM,
                output logic [31:0] ALUOutM, WriteDataM,
                output logic [4:0] WriteRegM
                );

                always_ff @(posedge clk or posedge reset)
                begin
                  if ( reset )
                  begin
                    RegWriteM <= 0;
                    MemtoRegM <= 0;
                    MemWriteM <= 0;
                    ALUOutM <= 0;
                    WriteDataM <= 0;
                    WriteRegM <= 0;
                  end
                  else
                  begin
                    RegWriteM <= RegWriteE;
                    MemtoRegM <= MemtoRegE;
                    MemWriteM <= MemWriteE;
                    ALUOutM <= ALUOutE;
                    WriteDataM <= WriteDataE;
                    WriteRegM <= WriteRegE;
                  end
                end
endmodule
```