Bilkent CS224-Preliminary Work

CS224
Lab 5
Sec 04
Altay İlker Yiğitel
22203024
26.11.2024

Compute-use Hazard

Computer-use is a data type hazard affecting execute and memory stages.
It happens because the instruction before has not written back the registers current instruction is going to use. Solution is stalling or forwarding the data.

Load-use Hazard

Load-use is a data type hazard and affects memory and execute stages. It happens when the previous load word instruction has not yet changed the register that will be used. Solution is stalling the data.

Load-store Hazard

Load-store is a data type hazard affecting memory stage. It happens when store instruction needs data that is still being changed by previous instructions. Solution is stalling or forwarding the data.

J-type Jump Hazard

J-type jump is a control type hazard affecting the instructions fetched before jumping. It happens because the instructions after the jump work until the jump is complete. Solution is flushing.

Branch Hazard

Branch hazard is a control type hazard affecting the instructions fetched before branch. It happens because the instructions after the branch work until the branch is complete. Solution is flushing and branch prediction or stalling and prediction.

Logic of Hazard Unit for Forwarding

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

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

Logic of Hazard Unit for Stalling

```
StallF = MemToRegE && ((rtE == rsD) | (rtE == rtD));
StallD = StallF;
FlushE = StallF;
```

Logic of Hazard Unit for Flushing

```
FlushE = PcSrcD || JumpD;
```

Test with No Hazard

```
add $t0, $t1, $t2   // $t0 = $t1 + $t2
sub $t3, $t4, $t5   // $t3 = $t4 - $t5
and $t6, $t7, $t8   // $t6 = $t7 & $t8
or  $t9, $t1, $t2   // $t9 = $t1 | $t2
slt $s0, $t3, $t6   // $s0 = ($t3 < $t6)
```

Test with Compute-use Hazard

```
add $t0, $t1, $t2
sub $t3, $t0, $t4   # Use $t0 immediately
or  $t5, $t3, $t6   # Continue with  $t3
```

Test with Load-use Hazard

```
lw  $t0, 0($t1)
add $t2, $t0, $t3   // Use $t0 immediately
sub $t4, $t2, $t5   // Use $t2 immediately
```

Test with Load-store Hazard

```
lw  $t0, 0($t1)
sw  $t0, 4($t2)
add $t3, $t0, $t4  # Use $t0
sub $t5, $t3, $t6  # Continue using $t3
```

Test with Branch Hazard

```
label:
   beq $t0, $t1, label
   add $t2, $t3, $t4
   sub $t5, $t6, $t7
```

Test with Jump Hazard

```
label:
   j label
   add $t2, $t3, $t4
   sub $t5, $t6, $t7
```

Hazard Unit

```
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 [1:0] ForwardAE, ForwardBE,
    output logic FlushE, StallD, StallF
);

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

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

        StallF = MemToRegE && ((rtE == rsD) || (rtE == rtD));
        StallD = StallF;
        FlushE = StallF || PcSrcD || JumpD;
    end
endmodule
```

PipeDtoE

```
module PipeDtoE(
    input logic[31:0] instrD, PcPlus4D,
    input logic RegWriteD, MemToRegD, MemWriteD, RegDstD
    input logic [3:0] ALUControlD,
    input logic [4:0] rsD, rtD, rdD,
    input logic EN, clk,
    output logic[31:0] instrE, PcPlus4E,
    output logic RegWriteE, MemToRegE, MemWriteE, RegDstE
```

```
    output logic [3:0] ALUControlE,
    output logic [4:0] rsE, rtE, rdE
);

    always_ff @(posedge clk) begin
        if (EN) begin
            instrE <= instrD;
            PcPlus4E <= PcPlus4D;
            RegWriteE <= RegWriteD;
            MemToRegE <= MemToRegD;
            MemWriteE <= MemWriteD;
            ALUControlE<= ALUControlD;
            RegDstE <= RegDstD;
            rsE <= rsD;
            rtE <= rtD;
            rdE <= rdD;
        end
    end
endmodule
```

PipeEtoM

```
module PipeEtoM(
    input logic [4:0] WriteRegE, AluOutE,
    input logic RegWriteE, MemToRegE, MemWriteE,WriteDataE,
    input logic EN, clk,
    output logic RegWriteM, MemToRegM, MemWriteM,WriteDataM,
    output logic [4:0] WriteRegM,
);

    always_ff @(posedge clk) begin
        if (EN) begin
            RegWriteM <= RegWriteE;
            MemToRegM <= MemToRegE;
            MemWriteM <= MemWriteE;
            WriteRegM <= WrteRegE;
            WriteDataM <= WriteDataE;
        end
    end
endmodule
```