CS224 Lab - 05
Design Report

Mert Fidan - 22101734 - Section 3

06.05.2023

a) The list of all hazards

Name	Type	Affected Stages
Compute-use (RAW)	Data	Execute,Memory, Writeback
Load-use	Control	Decode, Execute
Branch	Control	Decode,Execute, Memory

b) Solutions to the hazards

- For compute-use hazards, which are RAW(read after write) hazards, data forwarding is needed. Thus, two ALU sources(SrcAE, SrcAB) are chosen from the three inputs of the multiplexers by the ForwardAE(for SrcAE) and ForwardBE(for SrcAB) signals sent from the hazard unit. The multiplexers' inputs take the source data from memory stage, writeback stage and DtoE pipeline register.
- For load-use hazards, stalling the processor is needed since forwarding is not possible for this hazard. By flushing the Execute stage, and stalling

Fetch and Decode stages the flushed instruction will be repeated in the next clock cycle, though with correct data. To implement this, FlushE, StallF and StallD signals are sent by the hazard unit to clear the pipeline registers or enable them to continue their execution.

- For branch hazards, calculation of BTA is done in Decode stage instead of Memory stage, reducing the branch misprediction penalty from 3 to 1. Branch hazards, similar to load-use hazards, are solved by stalling the pipeline according to the used registers and instruction type of the predeceasing instruction. If the previous instruction is a load instruction and branch instruction uses the related registers, stalling occurs due to Memory stage. If the previous instruction is R-type and branch instruction uses the related registers, stalling occurs due to Execute stage (OR relation between stalling due to Execute and stalling due to Memory stages).
- Therefore, for the stall operation, the hazard unit outputs(hazard control signals for load-use and branch hazards) are chosen as FlushE = StallD = StallF = lwstall(load-use stall control signal) OR branchstall(branch stall control signal).

c) Logic equations for each signal output by the hazard unit

- **lwstall** = ((rsD == rtE) || (rtD == rtE)) & MemtoRegE

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

- **FlushE** = **StallF** = **StallD** = (**lwstall** || **branchstall**)

- **ForwardAD** = (rsD != 0) & (rsD == WriteRegM) & RegWriteM

- **ForwardBD** = (rsD != 0) & (rsD == WriteRegM) & RegWriteM

- if ((rsE != 0) & (rsE == WriteRegM) & RegWriteM){
  ForwardAE = 2'b10
  }
  else if ((rsE != 0) & (rsE == WriteRegW) & RegWriteW){
  ForwardAE = 2'b01
  else{
  ForwardAE = 2'b00
  }

- if ((rtE != 0) & (rtE == WriteRegM) & RegWriteM){
  ForwardBE = 2'b10
  }
  else if ((rtE != 0) & (rtE == WriteRegW) & RegWriteW){

ForwardBE = 2'b01
}
else{
ForwardBE = 2'b00
}

d) Extended pipeline with “rol” instruction

- **ALU operation for rol:**
  (a << shamt) | (a >> (32 - shamt))

- **ALUControl for rol:** 100

- **Extended ALU module:**

```
module alu(input  logic [31:0] a, b,
           input  logic [4:0] shamt,
           input  logic [2:0]  alucont,
           output logic [31:0] result);

    always_comb
        case(alucont)
            3'b010: result = a + b;
            3'b110: result = a - b;
            3'b000: result = a & b;
            3'b001: result = a | b;
            3'b100: result = (a << shamt) | (a >> (32 - shamt)); // for ROL
            3'b111: result = (a < b) ? 1 : 0;
            default: result = {32{1'bx}};
        endcase
```

endmodule

- Shift amount(***shamt***) input for the ***rol*** instruction is added to ALU module.
- Since ***rol*** is an R-type instruction, all of the calculations are done inside the ALU. Thus, except from the ALU, no change is made to the pipelined processor.
- The hazards ***rol*** can cause are ***compute-use hazard***(RAW) which is due to reading a register directly after writing to it in the previous instruction, and ***load-use hazard*** if ***rol*** instruction is the successor of a load instruction. Compute-use hazard is solved by using ALU source multiplexers connected to Memory, Writeback stages and DtoE pipeline register; the control signals are sent by hazard unit to enable ***data forwarding*** in the processor. Load-use hazards are solved by ***stalling*** the processor and waiting(flushing) until correct data can be read from the register file.

➔**Test programs for the pipelined processor**

- **With Hazards:**

######### initial instructions

```
8'h00: instr = 32'h20080005; //ADDI $t0 $zero 0x0005
8'h04: instr = 32'hac080060; //SW $t0 0x0060 $zero
8'h08: instr = 32'h8c090060; //LW $t1 0x0060 $zero
8'h0c: instr = 32'h212a0004; //ADDI $t2 $t1 0x0004
8'h10: instr = 32'h212b0003; //ADDI $t3 $t1 0x0003
8'h14: instr = 32'h8d6b0058; //LW $t3 0x0058 $t3
```

```
    8'h18: instr = 32'h014b5022; //SUB $t2 $t2 $t3
    8'h1c: instr = 32'hac0a0070; //SW $t2 0x0070 $zero
    8'h20: instr = 32'h2008a0a0; //addi t0, zero, 0xa0a0
    8'h24: instr = 32'h01004880; //rol t1, t0, 2
    8'h28: instr = 32'h8c080070; //LW $t0 0x0070 $zero
    8'h2c: instr = 32'h8d09006c; //LW $t1 0x006C $t0
    8'h30: instr = 32'h01094820; //ADD $t1 $t0 $t1

######### only beq hazards
    8'h00: instr = 32'h2008a0a0; //addi t0, zero, 0xa0a0
    8'h04: instr = 32'hac090060; //SW $t1 0x0060 $zero
    8'h08: instr = 32'h8c0a0060; //LW $t2 0x0060 $zero
    8'h0c: instr = 32'h22b40003; //addi $s4 $s5 0x0003
    8'h10: instr = 32'h114d0000; //beq $t2 $t5 0x0000
    8'h14: instr = 32'h212c0004; //ADDI $t4 $t1 0x0004
    8'h18: instr = 32'h21300003; //ADDI $s0 $t1 0x0003
    8'h1c: instr = 32'h8d6b0058; //LW $t3 0x0058 $t3
    8'h20: instr = 32'h01495022; //SUB $t2 $t2 $t1
    8'h24: instr = 32'h114d0000; //beq $t2 $t5 0x0000
    8'h28: instr = 32'h01004880; //rol t1, t0, 2
    8'h2c: instr = 32'h8c100070; //LW $s0 0x0070 $zero
    8'h30: instr = 32'h8e29006c; //LW $t1 0x006C $s1

######### only load-use hazards
    8'h00: instr = 32'h2008a0a0; //addi t0, zero, 0xa0a0
    8'h04: instr = 32'hac090060; //SW $t1 0x0060 $zero
    8'h08: instr = 32'h8c0a0060; //LW $t2 0x0060 $zero
    8'h0c: instr = 32'h21540003; //addi $s4 $t2 0x0003
    8'h10: instr = 32'h212c0004; //ADDI $t4 $t1 0x0004
    8'h14: instr = 32'h21300003; //ADDI $s0 $t1 0x0003
    8'h18: instr = 32'h8e2b0058; //LW $t3 0x0058 $s1
    8'h1c: instr = 32'h01695022; //SUB $t2 $t3 $t1
    8'h20: instr = 32'h01004880; //rol t1, t0, 2
```

8'h24: instr = 32'h8c100070; //LW $s0 0x0070 $zero
8'h28: instr = 32'h8e29006c; //LW $t1 0x006C $s1

######### compute use hazards
8'h00: instr = 32'h2008a0a0; //addi t0, zero, 0xa0a0
8'h04: instr = 32'hac090060; //SW $t1 0x0060 $zero
8'h08: instr = 32'h0151a020; //add $s4 $t2 $s1
8'h0c: instr = 32'h02956020; //add $t4 $s4 $s5
8'h10: instr = 32'h21300003; //ADDI $s0 $t1 0x0003
8'h14: instr = 32'h01695022; //SUB $t2 $t3 $t1
8'h18: instr = 32'h01004880; //rol t1, t0, 2
8'h1c: instr = 32'h8c100070; //LW $s0 0x0070 $zero
8'h20: instr = 32'h8e29006c; //LW $t1 0x006C $s1

- **Without Hazards:**

8'h00: instr = 32'h2008a0a0; //addi t0, zero, 0xa0a0
8'h04: instr = 32'hac090060; //SW $t1 0x0060 $zero
8'h08: instr = 32'h8c0a0060; //LW $t2 0x0060 $zero
8'h0c: instr = 32'h212c0004; //ADDI $t4 $t1 0x0004
8'h10: instr = 32'h21300003; //ADDI $s0 $t1 0x0003
8'h14: instr = 32'h8d6b0058; //LW $t3 0x0058 $t3
8'h18: instr = 32'h01495022; //SUB $t2 $t2 $t1
8'h1c: instr = 32'h118d0000; //beq $t4 $t5 0x0000
8'h20: instr = 32'h01004880; //rol t1, t0, 2
8'h24: instr = 32'h8c100070; //LW $s0 0x0070 $zero
8'h28: instr = 32'h8e29006c; //LW $t1 0x006C $s1