ENSC 254 Final Project Report

Richard Kim SFU ID: 301588918

Vanessa Lau SFU ID: 30584103

Milestone 1 – Questions and Implementation

1. How do you generate control logic in the ID stage?
   a. This control logic of the ID stage was generated at stage_decode function in pipeline.c. Using the helper functions gen_control() and gen_imm() on lines 79 and 80 of pipeline.c, we created the control signals and immediate values. We also converted rs1, rs2 and rd's respective register values to their respective register values.
      The gen_imm and control functions determine using the instruction's opcodes through a switch statement in stage_helpers.h what operations to execute. These operations are setting the registers values (rs1,rs2, and rd), the immediate value (imm_val) and the various control signal values (alu_src, mem_to_reg, reg_write, mem_read, mem_write, branch, aluop), based on which instruction type it is.
2. How is the mux shown in the IF stage implemented?
   a. The mux shown in the IF stage is implemented through an if else statement in stage_fetch. The pc_src is used as the control signal in the mux and the condition in the if loop, determining whether the program will branch or will just update normally to the next instruction by four bytes. If pc_src jumps (if pc_src value is 1) then the current PC address is set by the branch address and then the pc_src value is reset to 0, where it then updates normally.
3. How is the mux shown in the EX-stage implemented?
   a. The mux shown in the EX-stage is implemented through another if else statement in stage_execute. The alu_src determines whether the rs1 and immediate values are operated on or the rs1 and rs2 values are operated on. These are then put through 2 helper functions of gen_alu_control and then performs said instruction's operations, based on what value gen_alu_control provides, in execute_alu.
4. A table of the alu_op values based on the instruction

Instruction:	alu_op:
R-type	10
I-type, S-type,UJ-type	0
I-type arithmetic	1
U-type	11

5. A table of the alu_control values based on the instruction

Instruction:	alu_control:
S-type, SB-type, UJ-type, I-type	0
I-type arithmetic	0x0, 0x3, 0x5, 0x6, 0x8, 0x9, 0x10, 0x12,
R-type	0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0x10, 0x11, 0x12

6. What is the logic behind the implementation of your gen_branch function?
    a. The logic behind the implementation of our gen_branch function was, based on the instruction's opcode value, it'd choose between if it's a jump and link or a branch. For a branch, using the instruction's funct3 value, the program checks if rs1 and rs2 are or are not equal. If they fulfill the condition of their instruction (i.e. rs1 and rs2 equal for beq), the boolean of gen_branch is set to true. These are the only branch types we needed to implement however, to implement other branch types, we'd just add more cases to the switch loop. For JAL, it returns the true if the opcode is correct and, if none of these are true, then the gen_branch bool is set to false.

Milestone 2 - Questions and Implementation

How have we implemented milestone 2 with reference to our project code?

1. What is the condition in your code that determine an EX-hazard? Explain your code with how it determines the EX-hazard. Include details about how you implemented forwarding to resolve the hazard.
    a. The condition in our code that determines an EX-hazard are if the detect_hazard helper function is triggered and if it isn't in the ID stage (int i =0). The code determines the EX-hazard by checking rs1 and then rs2. The hazard is checked for in these by checking if the EX/MEM destination register matches the ID/EX rs1 and rs2 registers. If it does not match, then it sets the correct forwarding register and then passes data to the next register. In gen_forward, the function checks if

the forwarding register is valid, then adds to their ex-ex forwarding counter. It then sets the correct registers and outputs the correct forwarding register.

2. What is the condition in your code that determines a MEM hazard? Explain your code with how it determines the MEM hazard. Include details about how you implemented forwarding to resolving this hazard.
    a. The condition in our code that determines a MEM-hazard is the same as an EX-hazard, it is just determined and resolved differently. The MEM-hazard is determined differently by checking if the MEM/WB destination register is different from the forwarding register from ID/EX. It is then passed through the gen_forward function, which sets the ID/EX forwarding register as the result of the MEM/WB.
3. What is the condition in your code that determines a load-use hazard? Explain how your code determines the load-use hazard.
    a. The condition in our code that determines a load-use hazard is it first checks if it's in the ID stage (int i==1) then it checks if the current instruction in ID/EX is a load and if the IF/ID rs1 or rs2 matches ID/EX destination register.
4. Explain how you implement the pipeline flush (inserting a NOP) in your code?
    a. The pipeline flush is implemented in our code in the cycle_pipeline. If the control fix flag was applied, then the debug print for flushing is outputted, clears the destination register and then clears the source registers and then inserts the NOP by adding 0 to register 0 for IF/ID, ID/EX, and EX/MEM stages then it clears all relevant flags.
5. Explain how you insert a bubble in your code to implement a stall
    a. In our simulator code, we inserted a bubble by first, stalling the PC by decrementing by 4 to re-fetch the same instruction, then it creates bubbles to stall until it gets the correct input and output registers (i.e. if they match). Then the stall counter is incremented, and the load hazard is set to true for the next bubble cycle.
6. Document any other change you had made to make in pipeline stage functions to accommodate the above requirements. Make sure to include reasons for these changes.
    a. One of the changes that was made was in the load-use hazard detection. If the load-use hazard fix is pending, then it sets the ID/EX input destination register and output destination register to 0 and then resets the stalling flag of fixload to false. This ensures that in the next cycle, when stalling or bubbling, the ID/EX doesn't write to the wrong registers due to stale register values.