Name: Shukhratbek
Subject: ADVANCED COMPUTER ARCHITECTURE (2024 Fall)
Assignment: Project1-Part2
Data: Oct 16 2024

Question 5:
Implement Data Forwarding (After all instructions are implemented, add forwarding for data hazards).
Provide instructions for compiling and running each of your tests.

Question 6:
Implement Branch Speculation (After all instructions are implemented, implement speculate and squash (if needed.)) Provide instructions for compiling and running each of your tests.

Question 7: Towards Midterm;
Implement GShare for the simulator.
Description
Implement a Gshare predictor for you in order pipeline processor.
You will implement a branch predictor that uses:
1) 16-entry Branch History Table (BHT) that is initialized to all zeros
with 4 bits of history.
2) Xor, which takes the output of the BHT and the last 4 bits of your PC.
3) This will index into a 16-entry pattern history table
containing a 2-bit state machine. The initial state will be weakly NOT-TAKEN.
You will also implement a 16-entry branch target buffer organized as a
fully associative cache with FIFO replacement.
The branch target buffer tag is the PC of the branch instruction, and the data
portion is the target address of the branch last time it was calculated.
Entries are put into the BTB only when a branch is resolved and TAKEN.
If you predict a branch to be taken but do not find an entry in the BTB,
fetch (speculatively) from PC+1.

Solutions:
Question 5:

To implement data forwarding for handling data hazards in a pipelined processor, we need to make several modifications to the existing pipeline structure. Here's a detailed explanation of how to implement data forwarding, along with instructions for compiling and running tests:

Implementing Data Forwarding

1. Modify the Datapath

First, we need to modify the datapath to include forwarding paths:

- Add multiplexers at the inputs of the ALU in the Execute stage.
- Create new data paths from the ALU output and data memory output back to these multiplexers.

2 Create a Forwarding Unit

Implement a forwarding unit that detects data hazards and controls the forwarding multiplexers:
(q5-create-forward-unit.v)

```
module ForwardingUnit (
    input [4:0] ID_EX_Rs, ID_EX_Rt,
    input [4:0] EX_MEM_Rd, MEM_WB_Rd,
    input EX_MEM_RegWrite, MEM_WB_RegWrite,
    output reg [1:0] ForwardA, ForwardB
);

always @(*) begin
    // Forward A logic
    if (EX_MEM_RegWrite && (EX_MEM_Rd != 0) && (EX_MEM_Rd == ID_EX_Rs))
        ForwardA = 2'b10;
    else if (MEM_WB_RegWrite && (MEM_WB_Rd != 0) && (MEM_WB_Rd == ID_EX_Rs))
        ForwardA = 2'b01;
    else
        ForwardA = 2'b00;

    // Forward B logic
    if (EX_MEM_RegWrite && (EX_MEM_Rd != 0) && (EX_MEM_Rd == ID_EX_Rt))
        ForwardB = 2'b10;
    else if (MEM_WB_RegWrite && (MEM_WB_Rd != 0) && (MEM_WB_Rd == ID_EX_Rt))
        ForwardB = 2'b01;
    else
        ForwardB = 2'b00;
end

endmodule
```

3. Update the Pipeline Registers

Modify the pipeline registers to include the necessary fields for forwarding:
(q5-update-pipeline-register.v)

```
// ID/EX Register
reg [4:0] ID_EX_Rs, ID_EX_Rt;

// EX/MEM Register
reg [4:0] EX_MEM_Rd;
reg EX_MEM_RegWrite;

// MEM/WB Register
reg [4:0] MEM_WB_Rd;
reg MEM_WB_RegWrite;
```

4. Integrate Forwarding into the Execute Stage

Update the Execute stage to use the forwarding multiplexers:
(q5-integrate-forwarding.v)

```
module ExecuteStage (
    input [31:0] ALU_input1, ALU_input2,
    input [31:0] EX_MEM_ALU_result, MEM_WB_result,
    input [1:0] ForwardA, ForwardB,
    output [31:0] ALU_result
);

reg [31:0] ALU_operand1, ALU_operand2;

always @(*) begin
    case (ForwardA)
        2'b00: ALU_operand1 = ALU_input1;
        2'b10: ALU_operand1 = EX_MEM_ALU_result;
        2'b01: ALU_operand1 = MEM_WB_result;
        default: ALU_operand1 = ALU_input1;
    endcase

    case (ForwardB)
        2'b00: ALU_operand2 = ALU_input2;
        2'b10: ALU_operand2 = EX_MEM_ALU_result;
        2'b01: ALU_operand2 = MEM_WB_result;
        default: ALU_operand2 = ALU_input2;
    endcase
end

// ALU operation
assign ALU_result = ALU_operand1 + ALU_operand2; // Example: ADD operation

endmodule
```

Compiling and Running Tests

To compile and run tests for the data forwarding implementation, follow these steps:

1. Create a testbench file (e.g., q5-forwarding_tb.v) that instantiates the pipeline modules and the forwarding unit.
2. Write test cases that cover various forwarding scenarios:
   (Source code attached)
3. Compile the Verilog files using a Verilog compiler (e.g., Icarus Verilog): (bash)
   iverilog -o forwarding_sim q5-integrate-forwarding_tb.v q5-update-pipeline-register.v q5-create-forwarding-unit.v
4. Run the simulation:
   vvp forwarding_sim
5. Analyze the simulation results to verify that data forwarding is working correctly.

Example Test Cases:

1. EX-EX Forwarding:
   ADD R1, R2, R3
   SUB R4, R1, R5

2. MEM-EX Forwarding:
   ADD R1, R2, R3
   NOP
   SUB R4, R1, R5

3. WB-EX Forwarding:
   ADD R1, R2, R3
   NOP
   NOP
   SUB R4, R1, R5

4. No Forwarding Required:
   ADD R1, R2, R3

SUB R4, R5, R6

5. Multiple Rorwarding:

ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R4

For each test case, set up the initial register values, run the instructions through the pipeline, and verify that the correct values are forwarded and the final results are accurate.

By implementing these test cases and running them through your simulation, you can ensure that your data forwarding implementation correctly handles various data hazard scenarios in the pipeline.

Question 6:

To implement branch speculation we'll need to modify our existing processor simulator to include speculative execution and branch prediction. Here's a step-by-step guide to implement branch speculation, along with instructions for compiling and running tests:

1. Implement Branch Predictor

Create a branch predictor class that uses a 2-bit saturating counter for each branch: (q6-branchpredictor.cpp)

```cpp
// Name: Shukhratbek
// Subject: ADVANCED COMPUTER ARCHITECTURE (2024 Fall)
// Assignment: Project1-Part2
// Data: Oct 16 2024

class BranchPredictor {
private:
    std::unordered_map<uint32_t, uint8_t> predictorTable;

public:
    bool predict(uint32_t branchAddress) {
        uint8_t counter = predictorTable[branchAddress];
        return counter >= 2;
    }

    void update(uint32_t branchAddress, bool taken) {
        uint8_t& counter = predictorTable[branchAddress];
        if (taken && counter < 3) counter++;
        else if (!taken && counter > 0) counter--;
    }
};
```

2 Modify Pipeline Class

Add speculation and squash functionality to your pipeline class: (q6-pipeline.cpp)

3 Modify Instruction Execution

Update your instruction execution logic to handle speculative execution: (q6-instruction.cpp)

```
// Name: Shukhratbek
// Subject: ADVANCED COMPUTER ARCHITECTURE (2024 Fall)
// Assignment: Project1-Part2
// Data: Oct 16 2024

CXX = g++
CXXFLAGS = -std=c++11 -Wall -Wextra -O2

SRCS = q6-main.cpp q6-pipeline.cpp q6-instruction.cpp q6-branchpredictor.cpp
OBJS = $(SRCS:.cpp=.o)
EXEC = processor_simulator

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $<

clean:
    rm -f $(OBJS) $(EXEC)
```

COMPILATION AND TESTING

1. **Create a Makefile** (q6-makefile)
2. **Compile the Simulator: (bash)**
   make
3. **Create Test Cases:** (text)

Create assembly test files to evaluate branch speculation:

```
text

# test_loop.asm
    mov r1, #0
    mov r2, #10
loop:
    add r1, r1, #1
    cmp r1, r2
    bne loop
    halt
```

```
text

# test_if_else.asm
    mov r1, #5
    cmp r1, #10
    bgt greater
    mov r2, #0
    b end
greater:
    mov r2, #1
end:
    halt
```

4. Run Tests

Execute the simulator with test files:

```
./processor_simulator test_loop.asm
./processor_simulator test_if_else.asm
```

5. Analyze Results

Add logging to our simulator to track branch prediction accuracy and performance metrics: (q6-printstatistic.cpp)

```cpp
// Name: Shukhratbek
// Subject: ADVANCED COMPUTER ARCHITECTURE (2024 Fall)
// Assignment: Project1-Part2
// Data: Oct 16 2024

void Pipeline::printStatistics() {
    std::cout << "Total instructions executed: " << totalInstructions << std::endl;
    std::cout << "Branch predictions: " << branchPredictions << std::endl;
    std::cout << "Correct predictions: " << correctPredictions << std::endl;
    std::cout << "Prediction accuracy: "
              << (float)correctPredictions / branchPredictions * 100 << "%" << std::endl;
    std::cout << "Total cycles: " << totalCycles << std::endl;
    std::cout << "IPC: " << (float)totalInstructions / totalCycles << std::endl;
}
```

Testing Scenarios

1. **Loop Behavior**: Use test_loop.asm to evaluate how well your branch predictor handles loops.
2. **If-Else Structures:** Use test_if_else.asm to test branch prediction in conditional statements.
3. **Mixed Patterns**: Create a test case with a mix of loops and conditionals to stress-test your speculation mechanism.
4. **Edge Cases**: Test with very short and very long loops, as well as nested branch structures.

By implementing these components and following this testing procedure, we'll have a functional branch speculation system in our processor simulator. This implementation allows for dynamic branch prediction and speculative execution, which are key techniques in modern processors to boost performance. The branch predictor uses past behavior to make educated guesses about branch outcomes, while speculative execution allows the processor to execute instructions ahead of time, reducing pipeline stalls and improving

efficiency.

Remember to thoroughly analyze the results, comparing the performance with and without speculation enabled. This will help us understand the benefits and potential pitfalls of branch speculation in different scenarios, which is crucial knowledge in advanced computer architecture.

Question 7: Towards Midterm;

Here's an implementation of a GShare branch predictor with a Branch Target Buffer (BTB) for an in-order pipeline processor:

GShare Branch Predictor Implementation (q7-branch-predict.cpp)

1. **Branch History Table (BHT):** Implemented as a 16-bit integer, initialized to all zeros. It stores the global history of the last 16 branches.
2. **Pattern History Table (PHT)**: An array of 16 8-bit integers, each representing a 2-bit saturating counter. Initialized to weakly NOT-TAKEN (01).
3. **Branch Target Buffer (BTB)**: Implemented as an array of 16 BTBEntry structures, each containing a tag (branch PC), target address, and valid bit.
4. **predict_branch()**: Uses the last 4 bits of the PC XORed with the last 4 bits of the BHT to index into the PHT. Returns true (taken) if the PHT entry is 2 or 3.
5. **update_predictor()**: Updates both the PHT and BHT based on the actual branch outcome.
6. **check_btb()**: Searches the BTB for a matching PC. Returns the target if found, otherwise PC+1.
7. **update_btb():** Updates the BTB with a new entry, using FIFO replacement if necessary.
8. **predict_and_update():** Main function that combines prediction, BTB lookup, and updates. It returns the predicted next PC.

To use this in your simulator, call predict_and_update() with the current PC, whether it's a branch instruction, the actual branch outcome, and the actual target address (for taken branches). This function will return the predicted next PC for fetching.

Remember to integrate this with your pipeline stages, handling speculative execution and branch misprediction recovery as needed.