Objective

Implement in C++ a cycle-accurate simulator of a processor that has dynamical scheduling using Tomasulo algorithm with the addition of a Reorder Buffer. The simulator will have 64 general purpose register where half are floating point and the other are integer, instruction and data memory units, an arbitrary number of execution units that can have different latencies based on what type of instruction is executing, a reservation station, a reorder buffer, an instruction window, and an execution log. The simulator should be able to handle RAW hazards because of register renaming. Additionally, the design must have branch speculation and can handle structural hazards as well as issue width of more than 1.

Code

The simulator will be designed and implemented in C++ using CLion. The simulator will be a simplified implementation thus memory address is performed in before entering the execution stage. Makefile was modified adding -std=c++11 in the OPT (figure 1).

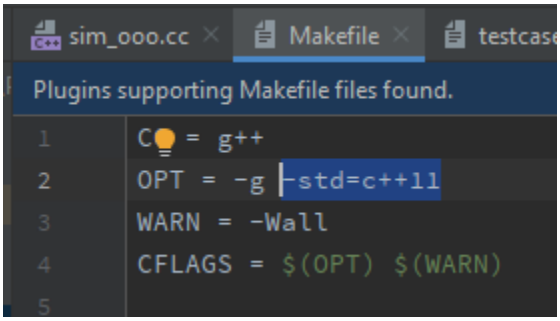

Figure 1

Data Structures

In the code template there where many different data structures that were implemented for different things such as instruction, execution unit, instruction window, reorder buffer, and reservation station. I extended some of the provided data structure including more fields in the execution unit, reorder buffer and reservation station.

In the execution unit data structure, a Boolean a data type was added that is used to handle finding instruction that is done in the write result stage. In the reorder buffer, an opcode field was added. This opcode field is added to identify what instruction is being committed in the commit stage as different instruction have different requirements. Lastly, the reservation station data structure has 4 additional data types: one opcode, two Boolean, and one unsigned. The opcode again is used to identify the instruction added into the reservation station, the two Boolean data types are used by the execution unit to handle what instruction can be executed as well as preventing the same instruction from executing more than once and the unsigned int data type is used as the immediate field for ADDI and SUBI as well as the target for branch instructions. Along with the template data structures, there are additional data members added to the simulator (see figure 2).

```
class sim_ooo{

    /* Add the data members required by your simulator's implementation here */

    unsigned int_registers[NUM_GP_REGISTERS];       // Integer Registers
    unsigned fp_registers[NUM_GP_REGISTERS];        // Floating Point Registers

    Register_Renaming regtag[2*NUM_GP_REGISTERS];   // Holds rob number for register renaming for both int and fp

    unsigned Program_Counter;                       // Keep track of what instruction to issue

    bool branchisfalse;                             // For  Branch speculation
    unsigned last_entry;                            // Keeping track of the last entry into the ROB

    unsigned eop_pc = UNDEFINED;                    // Keep track of EOP's pc used to identify the last instruction
    bool eopend = false;                            // EOP flag
    /* end added data members */
```

Figure 2

There are two arrays used to represent the two different types of general purpose registers, a data structure called Register_Renaming that is an array of twice the size of the number of general purpose register such that it can hold both renaming tag for integer and floating point, a program counter, a boolean data type for branch speculation, a rob tracker, and EOP flags.

Register_Renaming is a class that contains 3 data types, one that holds the rob entry number, two are used to identify the instruction based on the opcode and the instruction's program counter (see figure 3).

```
class Register_Renaming{
public:
    unsigned tag;   //renaming tag in ROB

    //instruction identifier to diff instruction that uses the same register for renaming
    opcode_t op;
    unsigned pc;    //identifies a new vs old instruction that uses similar destination and register when removing renaming
};
```

Figure 3

Tomasulo Algorithm with Reorder Buffer

The simulator uses 4 stages to represent the stages that a processor would contain when using Tomasulo algorithm with a Reorder Buffer. The fours stages are called Issue Stage, Execution Stage, Write Result Stage and Commit Stage. Each stage performs different tasks necessary for the processor to run instructions. In my implementation, I have the Stages performed in order.

1. Issue Stage

In this stage, the program checks for free entry in both the Reorder Buffer and the Reservation Station and if both are available, the instruction is issued to that location. If either reservation station of ROB does not have a free entry, the instruction(s) are not issued thus handles structural hazard on ROB and Reservation Station. This stage can be repeated multiple times depending on the Issue Width (see Figure 4).

```
//Issue Stage
for(int i = 0; i <issue_width; i++) {
    instr = instr_memory[(Program_Counter - instr_base_address) >> 2]; //Divide PC by 4 to get instructions
    //cout << "Program Counter: " << hex << Program_Counter << endl;
    if (instr_op == EOP)    //if EOP if fetched, note the eop's program counter such that if the
                            // last instruction is committed end RUN
        eop_pc = Program_Counter;

    instr_op = instr.opcode;
    IssueROB = get_free_ROB_entry();
    //if(IssueROB != UNDEFINED) cout << "Free ROB Found! Entry #" << IssueROB << endl;
    IssueRES = get_free_reservation_station(instr_op);
    //if(IssueRES != UNDEFINED) cout << "Free RES Found! Entry #" << IssueRES << endl;

    if (IssueRES != UNDEFINED && IssueROB != UNDEFINED) {
        set_reservation_station(instr, Program_Counter, IssueRES, IssueROB);
        set_ROB_entry(instr, Program_Counter, IssueROB);
        rob.entries[IssueROB].state = ISSUE;
        set_instr_window(IssueROB, instr_stage: ISSUE);
        Program_Counter = Program_Counter + 4;
    }
}
```

Figure 4

Additionally, the Issue stage also checks for EOP instruction and when detect takes note of the EOP's program counter. This will be used later in the Commit Stage to determine whether the program should stop running.

2. Execution Stage

In this stage, the reservation station entries are identified as ready to be executed first before it determined whether the entries are ready themselves. This is to prevent the instruction from finishing the issue stage then going straight into the execution stage in one clock cycle. As mentioned before, the reservation station entries have two additional Boolean data types, one that is set to true if both Qj and Qk fields are UNDEFINED called ready. When an instruction is executed, the execution unit sets the other bool true to prevent the same instruction executing twice. The instruction

will have latency based on which execution unit it uses thus can take multiple clock cycle before the instruction can go to the Write Result Stage. See Figure 5 for additional details.

3. Write Result Stage

In this stage, instructions that has completed the Execution Stage has its output added to the common data bus which propagates to the ROB and reservation station. Then the reservation station holding that instruction can be cleared. This stage can be repeated multiple times in a clock cycle thus can handle multiple instructions that are finish executing (see Figure 6).

```
//Execution Stage
for(int i = 0; i < reservation_stations.num_entries; i++){
    exec_op = reservation_stations.entries[i].opcode;
    if(reservation_stations.entries[i].pc != UNDEFINED){
        if(reservation_stations.entries[i].ready && !reservation_stations.entries[i].executing){
            ExecUnitIndex  = get_free_unit(exec_op);
            if(ExecUnitIndex != UNDEFINED){ // exec unit found
                exec_units[ExecUnitIndex].busy = exec_units[ExecUnitIndex].latency + 1;
                exec_units[ExecUnitIndex].pc = reservation_stations.entries[i].pc;
                exec_units[ExecUnitIndex].inuse = true;
                reservation_stations.entries[i].executing = true;   // prevent executing the same instruction
                ExecROB = find_entry(reservation_stations.entries[i].pc);
                rob.entries[ExecROB].state = EXECUTE;
                set_instr_window(ExecROB, instr_stage: EXECUTE);
                if(exec_op == LW || exec_op == LWS) reservation_stations.entries[i].address =
                        reservation_stations.entries[i].address + reservation_stations.entries[i].value1;
            }
        }
        else if(!reservation_stations.entries[i].ready){
            if(reservation_stations.entries[i].tag1 == UNDEFINED
            && reservation_stations.entries[i].tag2 == UNDEFINED)
                reservation_stations.entries[i].ready = true;
        }
    }
}

// Decrements busy for instructions being executed
for(int i = 0; i < num_units; i++){
    if(exec_units[i].busy > 0) exec_units[i].busy--;
}
```

Figure 5

```
//Write Result Stage
//Find finished instructions
for(int i = 0; i < num_units; i++){
    if(exec_units[i].busy == 0 && exec_units[i].inuse){
        WRRES = find_reservation_station_instr(exec_units[i].pc);
        WRROB = find_entry(exec_units[i].pc);
        wr_op = reservation_stations.entries[WRRES].opcode;
        //cout << "Opcode in WR: " << wr_op << endl;
        if(is_memory(wr_op)){
            rob.entries[WRROB].state = WRITE_RESULT;
            set_instr_window(WRROB, instr_stage: WRITE_RESULT);
            WRoutput = char2unsigned( buffer: data_memory + reservation_stations.entries[WRRES].address);
            //cout << "Output is: "<< hex << WRoutput << endl;
            CDB(WRROB,WRoutput);
            clean_res_station( entry: &reservation_stations.entries[WRRES]);
        }
        else{
            rob.entries[WRROB].state = WRITE_RESULT;
            set_instr_window(WRROB, instr_stage: WRITE_RESULT);
            WRoutput = alu(wr_op, reservation_stations.entries[WRRES].value1, reservation_stations.entries[WRRES].value2,
                           reservation_stations.entries[WRRES].immediate, exec_units[i].pc);
            //cout << "Output is: "<< hex << WRoutput << endl;
            CDB(WRROB,WRoutput);
            clean_res_station( entry: &reservation_stations.entries[WRRES]);
        }

        // clear execution units
        exec_units[i].inuse = false;
        exec_units[i].pc = UNDEFINED;
    }
}
```

Figure 6

4. Commit Stage

In this final stage, instructions are committed in order. If an instruction has its value ready in the ROB, it will not be able to commit until all the other instruction preceding it commits first. To make sure this occurs, a for loop is used to find the ROB entry with the lowest program counter. This entry is checked to see if it is ready to be committed. If so, the instruction's opcode determines what occurs.

For integer instructions, the integer register array is updated and that instruction is added into the execution log. Similarly, for fp instruction, that instruction is added into the execution log but the floating point register is updated instead. For branch instructions, the instruction is added into the execution log and if the target does not point to the instruction following the branch (branch instruction's pc + 4), the speculation is correct thus does not flush the ROB, reservation station, etc. Regardless of what instruction is committed, the ROB entry is freed.

If the program counter for the committed instruction indicates that it is the last instruction (EOP's pc – 4) and there are no misprediction, the program ends.

Conclusion

The processor designed for dynamic scheduling can be useful allowing instructions to execute out of order if that instruction is not dependent on the previous instructions. With Tomasulo algorithm with ROB, instructions can be dynamically scheduled using renaming. This allows out of order execution, but the instruction is still issued and committed in order. This eliminates many hazards such as read after write, write after write, and write after read hazards. In my implementation, I followed the simplified version of the simulator and can handle all testcases 1 to 6. Since I am not taking this course on the Graduate level, I do not have to implement store instructions thus not having RAW hazards on memory.

Test case	**Points**	**Comment (brief explanation of test case evaluation)**
Test case 1	6	Test case matches with the output of my code
Test case 2	6	Test case matches with the output of my code
Test case 3	6	Test case matches with the output of my code
Test case 4	6	Test case matches with the output of my code
Test case 5	6	Test case matches with the output of my code
Test case 6	10	Test case matches with the output of my code