ECE #563 **5-Stage MIPS Integer Pipeline Processor**

Zhiping Wang | Unity ID: zwang79 | Student ID: 200265085

02.28.2019

Objective

The goal of this project is to design and implement a C++ cycle-accurate simulator of a 5-stage MIPS-like pipelined processor.

Result

Pipeline designed is able to handle three kinds of hazard successfully, testcase results are matched.

Data Structure

Data memory:

```
unsigned char *data_memory;
```

Data memory is where 32-bit data is stored with two parameters, mem_size and mem_latency, that was defined by the constructor. Pipelined processor read out or write in data by LW/SW instruction. Also, data stored in the memory is initialized to 0xFF by the constructor at the beginning of every simulation.

Instruction memory:

```
instruction_t instr_memory[PROGRAM_SIZE];
```

Instruction memory is defined as instruction_t type. The load_program function translates and decomposes the assembly code into instruction_t type variables like opcode, src1, etc. and store them into instruction memory for IF stage to fetch specific instruction.

General purpose registers:

```
unsigned gp_register[NUM_GP_REGISTERS];
```

General purpose registers are the simulation of 32 register files of microprocessor, which stores the value of operands like R1 and F1.

Special purpose registers:

```
unsigned sp_register[NUM_STAGES][NUM_SP_REGISTERS];
```

Special purpose registers are the 5 registers that link different stages of the processor, like IF_ID. Each of entry of this 2-d array has specific registers that stores the intermedia values that propagates between stages in each clock cycle, including PC, NPC, IR, A, B, IMM, COND, ALU_OUTPUT, LMD. What we basically need to do is to update these values every clock cycle.

Instruction registers:

```
instruction_t Ins[NUM_STAGES];
```

While the instruction memory is defined as instruction_t type, the sp registers cannot store the instructions directly. I simply define a new instruction_t type array to propagate the instructions, so that the special values stored in the instruction can be easily extracted: Ins[ID].opcode, Ins[ID].src1, etc..

Basic pipeline --- testcase1

With no hazard, this testcase can be simulated by basic pipeline structure. A few things that I found tricky are as follow.

Pipeline stages run backwards. This is the easiest method to avoid value overwrite.

SW instruction decoding. SW is a little bit different when it comes to decoding. According to the alu function that is given, data is written at address[a+imm], which means

sp_register[EXE][A] = gp_register[Ins[ID].src2];

bool sim_pipe::WriteBack(). I define WB stage function as a bool function, in order to indicate that the program has run to an end.

Data Hazard handle --- testcase2,3

There are two parts of data hazard handling, detection and stall insert.

bool sim_pipe::DATA_hazard_detect() is the function I use to return that there is a data hazard detected on ID stage. The basic idea of this function is to extract instruction operands from different stages and check if there is a dependency. For example, if ID.src1 is the same as EXE.dest, then there is a back-to-back data dependency that requires 2-STALL insertion. If ID.src1 is the same as MEM.dest, then we only need to insert 1 STALL. Specially, for SW instruction, operand stored in ID.dest is not actually a destination operand, so the detection condition is a little bit different.

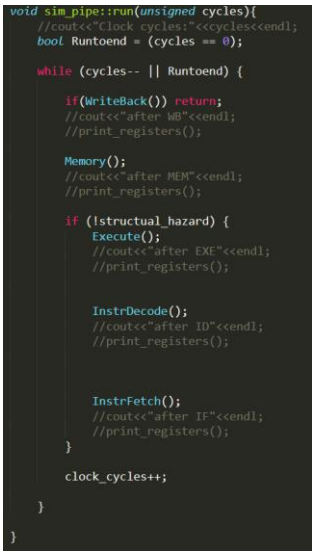
void sim_pipe::hazard_handling() is the function I use to handle hazard. As we can only insert one STALL in one clock cycle, so we need a flag signal, bool stall_stack, to indicate if there is a STALL inserted in the previous clock cycle. If there is not, we set stall_stack to true and insert a NOP into EXE stage, and proceed. If there is a NOP added in the previous cycle, we set stall_stack to false, insert a NOP and detect if there is any new hazard. NOP instructions are inserted on ID stage to EXE instruction register.

Control Hazard handle --- testcase4

Control Hazard is detected whenever ID stage decodes a BRANCH instruction by bool sim_pipe::CONTROL_hazard_detect(). The difference between dealing with data and control hazard is that NOP is inserted at IF stage to ID instruction register. In other words, when there is a data hazard, IF stage do nothing. However, for a control hazard, IF fetch a NOP instruction. Also, control hazard need stall stack method I mentioned in the data hazard section.

Structural Hazard handle --- testcase5

Structural hazard is the easiest one to handle among the three. Number of STALLs need to be inserted is fixed as memory latency. So, all I need to do is set a counter of this number when ever there is a memory operation, keep ignoring the EXE, ID, IF stages until the counter counts down to 0.



```
void sim_pipe::run(unsigned cycles){
    //cout<<"Clock cycles:"<<cycles<<endl;
    bool Runtoend = (cycles == 0);

    while (cycles-- || Runtoend) {

        if(WriteBack()) return;
        //cout<<"after WB"<<endl;
        //print_registers();

        Memory();
        //cout<<"after MEM"<<endl;
        //print_registers();

        if (!structual_hazard) {
            Execute();
            //cout<<"after EXE"<<endl;
            //print_registers();


            InstrDecode();
            //cout<<"after ID"<<endl;
            //print_registers();



            InstrFetch();
            //cout<<"after IF"<<endl;
            //print_registers();
        }

        clock_cycles++;

    }
}
```

Testcase6

Once other cases are matched, this one is not a problem.

Summary

This project gives me a deeper understanding of 5-stage MIPS pipeline. Debugging was suffering. Instead of setting break point, I found printing out intermedia values is more convenient.