

ECE 463/563 – Microprocessor Architecture Project 1 Report

Objective

To design and implement in C++; a cycle-accurate simulator like a 5-stage MIPS pipelined processor. The simulated pipeline will have general purpose registers (R0 to R31), special purpose registers, data memory, and 5 pipelined stages called Fetch (IF), Decode (ID), Execute (EXE), Memory (MEM), and a Writeback (WB) stage. The design must also be able to handle hazards common in a 5-stage MIPS pipelined processor; data memory hazards caused by reading a register that need to be written first, control flow hazards caused by conditional and unconditional branches, and structural hazards with memory latency.

Data Structures

The simulated pipeline needs both general purpose registers and pipeline registers. The data structure used in implementing them are also important. In my implementation of general-purpose registers, I used an array that can hold up to 32 registers. In figure 1, both get and set function need to access the data structure with only the register number. Since arrays holds data starting from index 0, like how registers are labeled, it was apparent to use an array. For example, if I want register R3, which is the 4th register, I can simple use that 3 to find the right index in the array; `gp_register[3*]` will be the 4th element in the array.

```
//returns value of general purpose register
int sim_pipe::get_gp_register(unsigned reg){
    return gp_register[reg]; //please modify
}

//sets gp register to a value
void sim_pipe::set_gp_register(unsigned reg, int value){
    gp_register[reg] = value;
}
```

Figure 1

In my implementation of pipeline registers, I used a struct to create my own data structure called `pipeline_register`. In `pipeline_register`, there are the special purpose registers that are needed to move information around in the pipeline. Since there are 4 pipeline registers inbetween each of the 5 stages, I created 4 `pipeline_register` objects (see figure 2).

```
pipeline_register IF_ID;
pipeline_register ID_EXE;
pipeline_register EXE_MEM;
pipeline_register MEM_WB;
```

Figure 2

Hazards

There are 3 hazards that are necessary to handle data hazards, control hazards and memory latency hazards. The hazards must be detected and resolved at specific stages with both data and control hazards are detected in the instruction decode stage and the structural hazard is detected in the memory stage. The hazards resolve in the writeback stage for data hazards, end of

execution stage (memory stage) for control hazards and memory latency resolves when the instruction reaches the writeback stage.

1. Data Hazards

In the simulated pipeline, the data dependencies between instructions needs to be implemented. The hazards are detected in the decode stage and removed when the instruction leaves the writeback stage. My implementation to handle this hazard is to check the register(s) that the instruction will read on (figure 3). Afterwards, save the registers that will be written on and will not be removed until they have been written. If the hazard is detected, stalls will be implemented to hold the read instruction until the write instruction is completed.

```
switch (instruction.opcode) {
    case SW:
    case ADD:
    case SUB:
    case XOR:
        if (checkRAW(instruction.src2)) {
            NOP_flag = true;
            break;
        }
    case LW:
    case ADDI:
    case SUBI:
    case BEQZ:
    case BNEZ:
    case BLTZ:
    case BGTZ:
    case BLEZ:
    case BGEZ:
        if (checkRAW(instruction.src1)) {
            NOP_flag = true;
            break;
        }
        NOP_flag = false;
        break;
    default:
        break;
}
```

Figure 3

2. Control Hazards

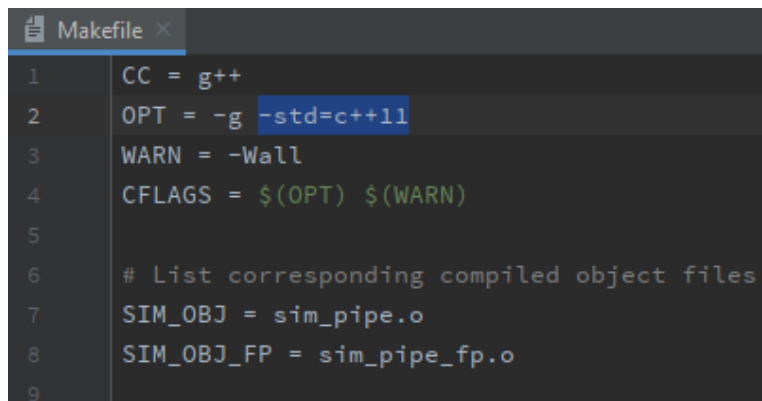
Control hazards are caused by conditional and unconditional branches. The detection happens in the decode stage and resolves once the pipeline knows what the next instructions will be. My implementation will stall the fetch stage until the branch has been executed. This will change what instruction is being fetched wither the branch condition is true or false.

3. Structural Hazards

The memory latency issue is detected whenever a load or store instruction reaches the memory stage. When detected, the memory stage will be stalled until the latency is accommodated for. In my implementation, stalls are added to the writeback stage and a counter in the memory stage is set. Then the counter resolves, load and store will send their special purpose registers to the MEM/WB pipeline register. While the stalls are occurring the EXE, ID and IF stages are halt as it waits for the MEM stage. The counter resets when the load and store instruction reaches the writeback stage.

Conclusion

I used CLion to design and implement the simulated pipeline processor and because of that, Makefile has been modified with the addition of `-std=c++11` in the OPT (see figure 4).



```
1 CC = g++
2 OPT = -g -std=c++11
3 WARN = -Wall
4 CFLAGS = $(OPT) $(WARN)
5
6 # List corresponding compiled object files
7 SIM_OBJ = sim_pipe.o
8 SIM_OBJ_FP = sim_pipe_fp.o
9
```

Figure 4

My implementation of each hazards works well. The data hazard is handled in both write instruction with a sequential read instruction and one instruction in between the write and read instruction. The control flow hazard works for the branch in testcase 4 and 5, but it is unknown how it will deal with jumps. The memory latency works when there is a RAW hazard and when there are none but, the structural hazard cannot handle a branch hazard along with it as proven in testcase 6. It is unknown if the hazard can handle all three at the same time.