Chuck Richardson

ECE 563

Dr. Michela Becchi

2/14/2023

Project 1 Report

1. Objective

The objective of this project was to implement a 5-stage MIPS pipelined processor in c++. This pipelined processor contains five stages including Fetch(IF), Decode(ID), Execute(EXE), Memory(MEM), and Writeback(WB). This design also includes 32 general-purpose registers(R0-R31). This MIPS processor needed to handle basic register arithmetic instructions such as ADD, SUB, XOR, as well as memory operations such as LOAD and STORE, and branching operations such as BEQZ, BNEZ, BLTZ, etc. This pipelined processor also had to handle hazards such as structural hazards pertaining to resource conflicts, data hazards pertaining to dependencies between instructions, and control hazards pertaining to branches and other control flow instructions. During this project the following metrics were kept track of such as the IPC(Instructions per Clock Cycle), Total Clock Cycles, Total Instructions, and Number of Stalls.

2. Data Structures

The MIPS pipelined processor needed special purpose registers, general purpose registers, and pipelined registers. For the general-purpose registers, I used an array with the size of the number of general-purpose registers. These general-purpose registers have a struct of type int_file_t, in which it contains two integer values to hold the value and to check if it is busy being written to. For the special purpose registers, I had to use a two-dimensional array in which the first index was the number of stages and 2nd index was the number of special purpose registers. For the pipelined registers, I used an array with the size of the number of stages to hold the instructions for each collective stage. This instruction register array has struct of type instruction_t to hold the opcode, src1, src2, etc. This is shown in Figure 1.

```
instruction_t     instruction_register[NUM_STAGES];
int_file_t        int_file[NUM_GP_REGISTERS];
unsigned          sp_registers[NUM_STAGES][NUM_SP_REGISTERS];
```

Figure 1: Data Structures Implementation

3. Hazard Handling

There are three types of hazards to be handled in the MIPS pipeline including structural hazards, data hazards(due to memory latency), and control hazards. Data Hazards and Control Hazards are detected in the Instruction Decode Stage. Data hazards are resolved in the writeback stage, control hazards are resolved in the execution stage, and structural hazards are resolved in memory stage once memory operations have completed.

3.2. Data Hazards

In the MIPS Pipeline there exists data dependencies between instructions. For RAW(Read-After-Write) Hazards we must stall the decoding of the instruction until the value for a specified register that is

needed to be written back has completed in the writeback stage. We check if the general-purpose registers are busy being written to and if they are then we stall. Once the value is written back, the instruction from decode stage can then be propagated to the execute stage. This is shown in Figure 2.

```
// Handling of RAW Data Hazards
if(( instruction.src1_op && int_file[instruction.src1].busy ) || (instruction.src2_op && int_file[instruction.src2].busy)) {
   stall_count++;
   instruction_register[EX].set_stall();
   clear_sp_register(EX);
   return true;
}

//Increment busy to say that the destination register is busy being written to
if(instruction.dest_op)
   int_file[instruction.dest].busy++;
```

Figure 2: Data Hazard Checking

3.3. Control Hazards

Control Hazards are caused by conditional and unconditional branching such as BEQZ,BLTZ, JUMP, etc. This detection happens in the decode stage. This will also change what instruction is being fetched, whether the branch condition is true or false. At the decode stage we will check if the instruction being executed at the execute stage is a branch if it is then we stall the decode stage as well as the fetch stage. Then once the target of the branch is calculated we will then execute the next instruction whether it be the instruction at the target of the branch or the next instruction in program order.

```
//Checking if we have a Control Hazard
if(instruction.branch_op) {
   stall_count++;
   instruction_register[ID].set_stall();
   clear_sp_register(ID);
   instruction_register[EX]              = instruction;
   return true;
}
else if (instruction_register[EX].branch_op) {
   stall_count++;
   instruction_register[ID].set_stall();
   clear_sp_register(ID);
   instruction_register[EX]              = instruction;
   return true;
}
else {
   instruction_register[EX]              = instruction;
   return (instruction.opcode == EOP);
}
}
```

Figure 3: Control Hazard Checking

3.4. Structural Hazards

One source of a Structural Hazards is memory latency. This issue is caused whenever a load or store instruction reaches the memory stage. If the instruction at the memory stage is a load or a store the other stages will be stalled such as IF, ID, and EXE until the memory stage has been completed its operation. Note we also have to stall Writeback in this case due to the ordering of our stages in the run function. The implementation of data memory latency and stalls are shown in Figure 4.

```
  //--------------------------------------------------------------------
  // Case of Load or Store
  //--------------------------------------------------------------------
   switch(instruction.opcode) {
      case LW:
         while(data_memory_latency_count--){ // Introduce Data Memory Latency
            stall_count++;
            instruction_register[WB].set_stall();
            clear_sp_register(WB);
            return true;
         }
         sp_registers[WB][LMD]               = read_memory( sp_registers[MEM][ALU_OUTPUT] );
         break;

      case SW:
         while(data_memory_latency_count--){ // Introduce Data Memory Latency
            stall_count++;
            instruction_register[WB].set_stall();
            clear_sp_register(WB);
            return true;
         }
         write_memory(sp_registers[MEM][ALU_OUTPUT], get_gp_register(instruction.src2));
         break;

      default: break;
   }
   instruction_register[WB]               = instruction;
   sp_registers[WB][ALU_OUTPUT]           = sp_registers[MEM][ALU_OUTPUT];
   return false;
}
```

Figure 4: Structural Hazard Implementation

4. Conclusion

In conclusion, my MIPS Pipeline works for all testcases. I used Clion as well as valgrind to debug possible issues such as incorrect values for each respective stage and segmentation faults. My implementation of data hazards works extremely well as well and passes all testcases. Shown below is a sample of my output from testcase6.

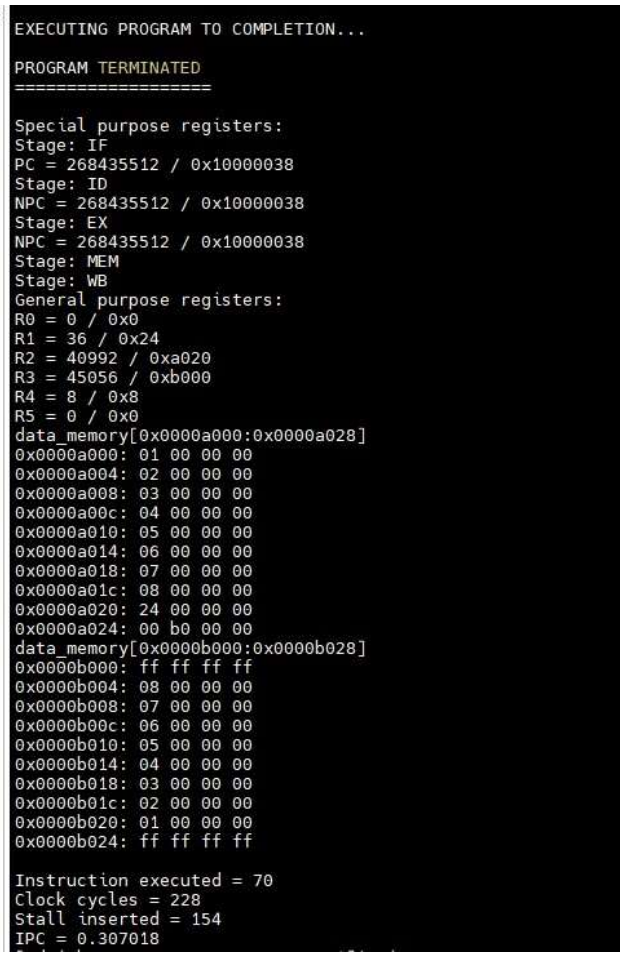

Figure 5. Testcase 6 Output