COL216 Assignment 8 Report

Navneel Singhal, Akash S

August 2020

Implementation

We developed a simulation of a pipelined processor in C++, implementing a subset of the MIPS instruction set (add, sub, sll, srl, sw, lw, beq, bne, blez, bgtz, j, jr, jal).
We have, in terms of implementation, stayed as close to the book as possible, with a very structural code that tries to emulate the real implementation of the processor. We stall on any hazard, as per the specifications of this assignment.

Running the code

To run the code, we only need to pipe the MIPS code into stdin while running the executable, with a file containing the initial state of the register file as the first and only command line argument.
The output consists of the set of instructions in every stage at every clock cycle, information about when the stalling happens, and we also print the final states of the registers and the memory as a check for the correctness of the code. For example, if `processor` is the executable, then running `./processor initialRegisterFile < programFile` will print the stages to stdout and the memory information to stderr (for ease of piping). Here initialRegister contains the initial state of the registers and programFile contains the programs.

Specifications of processor

1. The memory layout is as in the previous assignment (registers and memory in two separate arrays of fixed length).
2. Our model of a processor is the same as that in the book. We use 4 structs to represent the registers which are involved in inter-stage communication, and 5 functions which emulate the working of each stage in the pipeline. We also have some global variables which represent the signals relevant to hazard detection and stalling.
3. We also implement a function which represents the hazard detection unit of the processor.
4. To simulate the real implementation, we update the registers in the reverse order of stages to avoid making a copy of the data and keeping to the minimum usage of registers.
5. We use the EX stage to get information about branching instructions, and the ID stage to get information about the stalls as well as jump instructions, and use this information to proceed with the IF stage.
6. The END instruction is for the exit syscall, and whenever we come across this, we assume the execution has to be finished, and keep running the pipeline to completion.
7. The END_INSTRUCTIONS instruction is meant to signal that the whole program has been loaded into the memory.
8. The rest of the functions for parsing etc are as in the previous assignments.

Testing

Corner case testing

Corner case testing tests cases which could have caused an error in a hypothetical design of a processor using similar design decisions but these errors are either resolved or are handled like the MIPS architecture subset should have been expected to be handled (either unhandled or ignored).

1. Invalid instructions throw an error (while parsing).
2. We expect the programmer to keep the memory bounds in consideration, and thus for the sake of convenience, we do not raise exceptions for memory out of bound errors and use the garbage values instead, to make our processor faster by bypassing out of bound checks.
3. There is also a limit to the number of instructions (10^5 instructions including the END_INSTRUCTIONS and END instructions) and the behaviour in case of such cases is undefined.
4. Overwriting registers also works, for example, `add $t0 $t0 $t1` works perfectly.
5. Data hazards - to handle these, we use the data from the hazard detection unit to determine the number of cycles we need to stall the pipeline by. Based on the type of data hazard we either stall 1 or 2 cycles.
6. Control hazards - to handle these, we use the data from the hazard detection unit to determine the number of cycles we need to stall the pipeline by. We stall a single cycle for branch instructions. Due to our execution of stages in reverse order, after a single stall, the branch instruction reaches the EX stage and the comparison is done, and when the IF stage is executed the program counter is updated.
7. Cases mentioned in the descriptions of the test cases below.

We have included test cases for the these cases in the test directory.
Just to verify the code further, we use the test cases used in the previous assignments as well, like finding the sum of numbers from 1 to n recursively, and a random test case as well.
Note: The test cases for assignments 8, 9 and 10 are the same, and this was done to cross-validate the correctness across these assignments, by looking at the final states of the memory. For this sole reason, we give a Makefile in the submission for Assignment 10 that can be run optionally if you want to compare our results for all assignments.

Description of test files

For every test case, we specify an initial state of the register file, and then pipe in the MIPS program to stdin. We have made the following test cases. (Note that for every test case t.txt, the corresponding register file is t_reg.txt).

1. `acc.txt`: edge case for assignment 9, here this comes under the category of both kinds of data hazards at the same time, and we handle the case correctly by using the more recent forwarding result.
2. `branch.txt`: test case for branch hazard, which leads to a stall in each implementation as per specifications.
3. `branch_f.txt`: test case for checking whether branches work properly.
4. `example.txt`: random test case for checking data forwarding in assignment 9, 10 and stalling in assignment 8.
5. `hazard1.txt`: random test case for checking data forwarding in assignment 9, 10 and stalling in assignment 8.
6. `hazard2.txt`: test case for checking data forwarding in assignment 9, 10 and stalling in assignment 8, where we find the sum of numbers from 1 to 5 iteratively.
7. `oneton.txt`: test case for checking overall correctness in all assignments, where we find the sum of numbers from 1 to 8 recursively.
8. `bounds1.txt`: test case for illustrating garbage values for invalid memory access.
9. `bounds2.txt`: test case for illustrating garbage values for invalid memory access.
10. `invalid.txt`: test case for illustrating errors on an invalid instruction.
11. `max_instr.txt`: test case for illustrating errors due to passing more instructions than the instruction memory can handle.
12. `jump.txt`: test case for handling the special case of jr needing the value of a register that is being updated in the EX stage in the pipeline, where we stall instead of forwarding to the ID stage.