CISC 340 – Pipelined Simulator – Overview

James Allender
Vincent Li

November 21, 2018

1 Description:

The pipelined simulator is a machine code simulator that is based off of the single cycle behavioral simulator from project 1. The program consists of a single procedural C program to read in a given file containing machine code and simulate the machine code with the correct behavior. The program prints out the state of the machine at each cycle and then the statistics on the run to the screen at the end of execution.

2 Overview:

Our program works by reading in a machine code file generated using the professors machine code assembler. The number of lines of code is used to set the memory space for the instruction and data memory. The instruction and data memory are two separate arrays, initially containing the same data at the beginning of execution, to prevent simulated structural hazards resulting from memory being accessed by the fetch and write back stages at the same time.

Our project can be built by running the included make file:
$make
And our prioject is run using the syntax:
$pipelineSimulator -i [input file path]

As a pipelined simulator our program has a five stage pipeline used to process instructions. These stages are Fetch, Decode, Execution, Memory, and Write back. Fetch gets instructions from memory, decode interprets an instruction and gets the required values for execution, execution executed the instruction, memory stores or fetches data from/to memory, and write back writes the data back to the register file.

The **main()** function reads in the input file specified by the user and used that file to set the size of memory and load in the instructions. The initial state of the pipeline buffers is then set and initial NOOPs are loaded into the sages. The cycles of the machine are executed in a while loop in **main()** that runs until a halt is detected in the memory stage of the simulator leading to the loop ending and final statistics of the run of the machine being printed. Each of the stages of the pipeline are executed using functions that are called inside this while loop.

Between each of these five stages, and one after the final stage, are is what is effectively a set of pipeline registers that acts as pipeline buffers (and will be refereed to as such). These buffers are used to hold the output of the previous stage between cycles. At the beginning of a cycle a stage will read in data from the preceding pipeline buffer and then store it's output data in the following pipeline buffer. These buffers are IF/ID, ID/EX, EX/MEM, MEM/WB, and WB/END. The buffer following the last stage is used solely for data hazard handling and will be discussed shortly. The buffers used as input are in a structure **state** that represents the state of the machine during the current cycle execution and does not change during the cycle because the entire cycle theoretically happens at the same time instead of procedurally, and the output of the stages are stored in a structure that represents the following state **newstate**.

To handle data hazard we wrote two functions: **hasDataHazard** and **forwarding**. The basic idea for these two function is to check if there are any data hazard that will affect the execution in the current cycle. The **hasDataHazard** function takes a register index and a pipeline buffer index, specified 0-4 for the possible buffers, to check if the instruction in the given buffer writes to a register that will cause a data hazard. In the **forwarding** function the program calls

hasDataHazard six times to check if register A (field 0 in the instruction) and register B (field 1 in the instruction) are in pipeline buffer EX/MEM, MEM/WB or WB/END. To correct a found data hazard, the program takes the most updated data, the data in the closest following buffer, and forwards it to the execution stage by returning a structure including the forwarded data and flags specifying the nature of the forwarding.

If the flag for data in one of the register variables of the struct is 1, it indicates that we should update the data before the execution; otherwise, the program should ignore the data. However, If there is a LW instruction which would cause a data hazard the following instruction, the result from LW instruction will not be loaded from the data memory before the instruction pass the memory stage, so the **forwarding** function's returned instruction is changed to a NOOP and the flags for forwarding data are 0. In this case we stall by maintaining the contents of IF/ID and ID/EXE buffers and running this returned NOOP to be processed in execution and do not forward any data. The MEM/WB and WB/END buffers will execute normally moving the LW instruction forward. After we have the result from the LW instruction, we forward it to the execution stage. In the case of any required instruction following a LW instructions, we stall as described until the two instructions are in MEM/WB and WB/END and can both be forwarded to the instruction in execution. This is why we have the WB/END buffer following the final stage.

The handling of control hazards resulting from branch on equals (BEQ) instructions was far easier to handle than the data hazards. Our program uses branch not taken prediction meaning that when we fetch a BEQ instruction we assume it will not be taken. When the BEQ instruction makes it to the memory stage it's branch is taken if the result of the ALU was 0. When a branch is taken the instruction is in the memory stage and will be loaded into the MEM/WB buffer to continue execution and be retired. Taking the branch means the contents of IF/ID, ID/EX, and EX/MEM are now invalid. We overwrite each of these buffers with a NOOP instruction and modify the program counter to the branch target specified by the BEQ. The next instruction fetched will be at the branch target and the mispredicted instructions, replaced with NOOPs, will continue through the pipeline but do not count as fetched or retired instructions. This misprediction and instruction replacement leads to more instructions being fetched than are retired because the instructions that were fetched but replaced with NOOPs are not retired.

3 Challenges:

Easily the biggest challenge in this project was handling hazards, specifically data hazards with forwarding and NOOP insertion. A big part of this was needing to make sure we understood in detail how a pipelined approach works in minute detail. We spent a lot of time discussing how the pipeline works and trying out different methods for forwarding data to our execution stage and doing this forwarding at different parts of execution. Part of the problem we ran into was figuring out where we should be sending our data and where we should be doing the checking for hazards. This lead to a number of different versions of our forwarding mechanism. After we figured out to do our forwarding in execution and forward to a variable in execution we had to think of how to do this in a way that would allow us to forward appropriately and stall when necessary.

Initially we tried searching each of the following buffers to see if a single register that could cause a data hazard was in there, and then went through all the buffers again for the other possible hazard causing register. This wasn't working for us because in s certain ordering of consecutive load words that were required for a following instruction we would stall for one but the other would get written back to the register file without ever being forwarded. Part of the problem here was that we weren't forwarding from both the MEM/WB buffer and the WB/END buffer, we were only forwarding from WB/END.

We changed our approach to forward from both the MEM/WB buffer and the WB/END buffer and only stall if a LW is in the EX/MEM buffer. We also changed to checking each of the pipeline buffers for both of the possible data hazard registers at the same time instead of all of the buffers for one register at a time. We did this initially to ensure we knew whether or not a register was going to need to be forwarded regardless of if a NOOP bubble was inserted. This had been necessary because we were only forwarding from WB/END. After we re-wrote our forwarding/NOOP bubble insertion functions from the four very complex ones down to the two relatively straight forward ones we have now this checking became unnecessary, but we kept the method of checking anyway because it still seemed like a logical way to check for the hazards.