Assignment 3: Pipelined Gr8BOnd

Julianne Koenig
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, USA
jmko234@uky.edu

Dillon Tate
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, USA
deta224@uky.edu

Chase Vickery
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, USA
chase.vickery@uky.edu

Abstract—**For this project, we were tasked with implementing a pipelined version of the Gr8BOnd processor in Verilog. This involved taking all the steps that were done in the multicycle version and splitting them up into different segments that can be performed in parallel.**

I. GENERAL APPROACH

To begin, we started with Julianne's previous group's implementation of the processor, which relied on the instructions encoded by Dr. Dietz's AIK assembler solution. Using this as a starting point, we modified the multicycle version of the processor into a pipelined version, taking inspiration from the Fall 2018 PinKY pipelined processor.

II. GENERAL LAYOUT

We started by looking at the figure in the Assignment 3 page outlining a possible implementation using 4 stages (instruction fetch, register read, ALU/Memory, and register write). Our group determined that starting with just 3 pipeline stages - instruction fetch, register read, ALU/Memory + register write would be the simplest course of action for an initial version of the pipelined processor. Additionally, to account for instruction dependencies that may arise, we decided to implement an interlock that would stall stages with a NOP if needed. We decided to use the "owner computes" methodology for writing to temporary pipeline registers. Our pipeline was designed so that a stage would be able to write to its corresponding information registers by number, e.g. any stage could read from register ir0, but only Stage 0 should be writing to ir0.

A. Stage 0, Instruction Fetch

Stage 0 was the instruction fetch stage. This stage handles the actual setting of the program counter and fetching the instruction at an address. In order to set the PC, a temporary register, tpc, was initially set to be either PC or a jump target depending on if a jump signal was set from a later stage. Then, if Stage 0 was not being blocked by the next stage, the PC was free to be set to either the value of `tpc` in the case of a jump or branch, or `tpc + 1` (incrementing PC) if no branch was necessary. If Stage 0 was being blocked by Stage 1, then the PC was simply not incremented.

B. Stage 1, Register Read

This stage simply takes the information encoded by the instruction it was given and reads the values from the registers specified by the instruction into the `rd1` and `rs1` registers to be used for calculations and processing in the next stage. This stage also detects read after write dependencies, and stalls the stage as well as passes a NOP along to Stage 2 if needed.

C. Stage 2, ALU/Data Memory

Stage 2 is intended to decode the needed operation of the instruction, perform the operation, and write the results into their proper location. A case statement was used to choose the proper operation. Results of the operations were then directly written into a corresponding location register. In the case that the `st` instruction was used, the register value was instead written directly into memory instead of being stored elsewhere. In this stage, there is also a special case to handle NOPs. These NOPs can only be created by the processor in the case of a data hazard and control hazards.

III. TESTING

For testing the processor, we chose to try and test for read-after-write data hazards and for branches that would change the PC. For our processor, it does place NOPs in the case of a data hazard, but the control flow for branches seems buggy.

We used ci8 and addi to test the data-hazards to make sure that NOPs were properly placed in the pipeline. Additionally, we inspected the state values from the output trace to ensure that branches were jumping to the correct address. We also checked that the branches were triggering at the correct time.

IV. ISSUES

During this project there was a large amount of confusion regarding the ALU. The difference between functions and modules made it difficult to determine how the ALU should be constructed. After deliberation and experimentation, we used a module to represent the ALU hardware.

Although the program is able to run many commands and handle hazards, the ALU does not always compute the correct values.

Another issue was how to handle jumps (control flow hazards). To make sure the program counter was correct at all times, a separate pc register was added for each stage that could be accessed to add the correct value to the immediate. While the pc was pending a change, the pc was held and NOPs were used to prevent any additional instructions.