Parth Sehgal

Felipe Monteiro

CS 3220 Assignment 7 Part II Report

In order to implement the branch instruction properly, we have to handle stalls and push bubbles through the pipeline correctly. To handle branch stalls in the decode stage we essentially use the logic:

case brn/brz/brp/brnz/brzp/brnp/brnzp:

br\_stall = 1;

if (I\_EDCCWEn == 1 || I\_MDCCWEn == 1)

dep\_stall = 1;

else

dep\_stall = 0;

More specifically, the dependency stalls are conditional code dependencies, which need to be resolved in order to determine whether a branch should be taken or not.

Anytime we encounter any branch instruction, we assert the branch stall signal to the fetch stage. In addition to this, if we receive a condition code write enable signal from any later stages, we also set dep\_stall to 1 and assert the dependency stall signal to the fetch stage.

Fetch changes its O\_FE\_Valid and latch\_keep values based on branch stall and dependency stall signals:

|  |  |  |
| --- | --- | --- |
| case | latch\_keep | O\_FE\_Valid |
| Branch stall == 1 && dependency stall == 1 | 1 | 1 |
| Branch stall == 1 && dependency stall == 0 | 0 | 0 |
| Branch stall == 0 && dependency stall == 1 | 1 | 1 |
| Branch stall == 0 && dependency stall == 0 | 0 | 1 |

In addition to this, we have to insert NOPs manually in the case where the branch stall signal was asserted in one clock cycle and then released in the next. The problem here is that the branch is still in the pipeline, but the stall signal is low.

if (I\_BranchStallSignal == 1) begin

branch\_detected = 1;

branch\_bubble\_count <= 0;

end else if (I\_BranchStallSignal == 0 && branch\_detected == 1) begin

branch\_bubble\_count <= branch\_bubble\_count + 1;

if (branch\_bubble\_count == 2) begin

branch\_detected = 0;

end

end

When the branch stall signal is 0 but a branch has been detected (branch\_detected == 1), we can keep the O\_PC the same and push a NOP though the O\_IR. A NOP is denoted by an instruction for which a valid opcode does not exist, leading to nothing happening in the pipeline for the NOP.

To ensure that the hexadecimal output 0x37 is shown clearly in the FPGA board's hexadecimal display, we had to implement the halt instruction. This was accomplished in the execute stage of the pipeline through the following logic:

`OP\_HALT: begin

My\_O\_BranchPC\_Signal = I\_PC - 4; // Re-execute the halt instruction ad infinitum

My\_O\_BranchAddrSelect\_Signal = 1;

end

The branch address select signal is asserted, and the branch PC is decremented by one instruction, causing the PC to jump back to the previous instruction, which was a halt instruction. Effectively, this means that halt is executed in an infinite loop. Thus, the PC does not circle back around to 0. Rather than causing the code to be re-executed over and over, which makes it hard to discern the valid output of 0x37 for sum.hex, the halt instruction we implemented means that the code is executed one time only, leading to 0x37 being outputted unobtrusively and clearly.

The provided files my\_sum.s and sum.s both produce the correct output of 0x37; my\_sum.s was written by us whereas sum.s was provided.