EENG 467 Final Project: Pipelined Processor

Zeb Mehring

December 18, 2017

I. Overview

The purpose of this project was to design and implement a simple in-order pipelined processor similar to the E31 RISC-V Core using Verilog.

The design provided is a five-stage pipeline that accesses different memories for instructions and data. It supports the following instructions: `lw`, `sw`, `addi`, `slti`, `add`, `sub`, `slt`, `beq`, `blt`, and `bge`. The five stages are: instruction fetch, instruction decode, execute, memory access, and writeback.

II. Functionality

A. Instruction Fetch

The instruction fetch stage gets instructions from memory and propagates them to an output buffer. It increments the program counter so long as the processor isn't stalled and places the output from the instruction memory (the address requested in the previous cycle) into a buffer for the next stage. It also generates a stall in the event that the fetched instruction is a branch. While the processor is stalled, it sends NOP instructions into the buffer and does not increment the program counter. It resumes sending instructions from memory to the buffer when the branch has been computed and the program counter updated accordingly.

B. Instruction Decode

The instruction decode stage reads and processes the instruction from the buffer filled by the instruction fetch stage. If the processor is not stalled, it will output to a buffer the register operands, the immediate value, and the required ALU operation. It also detects data hazards and stalls the processor appropriately. It accomplishes this by marking the destination register of the instruction in the fetch buffer as "in use" and checks to see if the source registers are "in use." If so, it disables the updating of the instruction fetch buffer (to stall the hazard instruction), instruction memory (so that the following instruction is not lost), and the program counter (to trigger the actual stall). During the stall, it sends NOP instructions to the output buffer and maintains the signals it sent to trigger the stall (no updating of buffers, memory, or program counter). It resumes decoding instructions from the fetch buffer when the source registers of the instruction in the fetch buffer are no longer "in use."

C. Execute

The execute stage executes the operation specified in the output buffer of the decode stage. It stores to its own output buffer the result of the ALU computation and propagates the value of the second source register, which is used in the memory stage on load instructions. NOPs will cause the ALU to add the contents of register 0 to itself, producing a result of 0 (it is assumed that the value of register 0 is always kept at 0, though because NOP instructions also disable reading from and writing to data memory and the registers, this is not required).

D. Memory

The memory stage accesses memory, either storing or retrieving a value at an address specified by the ALU result in the execute buffer. Read and write enable for the memory module is activated if the instruction is a load or store, respectively. The stage outputs to a buffer the result from memory and propagates the result of the ALU, which is used in the writeback stage on register-type instructions. In the case of a branch instruction, the memory stage updates the program counter and ends the stall. If the branch was taken, it updates a register which causes the fetch stage to stall for an additional cycle while instruction memory reading is enabled to refresh the output.

E. Writeback

The writeback stage writes the value from the memory buffer back to the register file on load, immediate, or register-type instructions. It also removes the destination register from the "in use" list, effectively ending data hazards on the next cycle.

III. Timing

If the processor does not have to stall, all stages process a single instruction per cycle for a total CPI of 1. The operations done within each stage are updated on the negative clock edge (using the non-blocking assignment in Verilog) so that the subsequent stage in the pipeline can access the results in the following cycle.

In the case of a stall, the CPI is increased to accommodate. For branch hazards, the processor stalls (generating NOPs from the fetch stage) until the branch instruction reaches the writeback stage (if the branch is not taken) or until the branch exits the writeback stage (if the branch is taken, to allow for the instruction memory output to update). For data hazards, the processor stalls (generating NOPs from the decode stage) until the instruction causing the dependence exits the writeback stage and the stalled destination register has been written to. Decoding (and the rest of the pipeline) can safely proceed on the next cycle.

IV. Stalling

Stalling is essentially controlled by the register `branch_stall` and the wire `data_stall`. When a hazard is detected (fetch stage for branch hazards, decode stage for data hazards), these variables are set and are only disabled when the corresponding hazard exits the pipeline.

For branch hazards, detection and stalling is performed in the instruction fetch stage. If the opcode of the instruction coming from the instruction memory indicates a branch, then `branch_stall` is set and the program counter is not incremented. While `branch_stall` is still set, the fetch stage outputs NOPs into its output buffer and the program counter is not incremented. When the branch instruction is in the memory stage, the value of the program counter is updated and `branch_stall` is turned off. If the branch is taken, the register `branch_taken` is set, which stalls the fetch buffer for an additional cycle to allow for the instruction memory output to update to the branched instruction. Regardless, after `branch_stall` is disabled, the processor continues normal operation.

For data hazards, detection and stalling is performed in the instruction decode stage. Data hazards are tracked via a binary semaphore system. The decode stage maintains a register `in_use` whose width is the same as the number of registers. It sets the bit corresponding to the destination register of each load, immediate, or register-type instruction as it passes through the decode stage. It also checks to make sure the bits corresponding to the source registers of each instruction are are not set in `in_use`. If they are, then a stall is generated. Wires are used to disable reading from instruction memory or updating the instruction fetch buffer while the source register(s) of the instruction in the fetch buffer are set in `in_use`. When the stall is generated, `data_stall` is set and the output registers from the decode stage are filled with 0's (indicating NOP). The writeback stage unsets the destination register in `in_use` which enables reading from instruction memory, writing to the fetch buffer, and signals the decode stage to resume normally.

Note that the binary semaphore system here described has limitations: repeated writes to a single register immediately followed by a RAW hazard dependent on that register may behave incorrectly. To rectify this, a more sophisticated semaphore system should be used—one solution is to create "counters" for each register and stall the pipeline unless the source registers' counters in the fetch buffer are 0.

V. Testing

A sample test bench file (`pipeline_tb.v`) has been provided. The expected behavior is as follows:

1) Load memory word 3 (2) into register 4.
2) Branch to memory location 2061 ($PC + 2060$) if the value of register 0 is less than itself (always false). This generates a branch stall until the instruction enters the writeback stage 4 cycles later.
3) Branch to memory location 2062 ($PC + 2060$) if the value of register 1 is less than itself (always false). This generates a branch stall until the instruction enters the writeback stage 4 cycles later.
4) Load memory word 2 (1) into register 3.
5) Load memory word 1 ($2^{30} - 1$) into register 1.
6) Add the value in register 3 to the value in register 1 and store the result in register 2 ($2^{30} - 1 + 1 = 2^{30}$). This generates a data stall until register 1 has been loaded 3 cycles later.
7) Set register 5 to 1 if the value in register 3 is less than the immediate value 8, otherwise, set it to 0 ($2 < 8 \rightarrow 1$).
8) Set register 5 to 1 if the value in register 4 is less than the value in register 3, otherwise, set it to 0 ($2 \geq 1 \rightarrow 0$).
9) Add the value in register 0 to the value in register 4 and store the result in register 6 ($0 + 2 = 2$).
10) Add the immediate value 1 to the value of register 6 and store the result in register 1 ($1 + 2 = 3$). This generates a data stall until register 6 has been updated 3 cycles later.
11) Store the value in register 1 (3) to memory word 1. This generates a data stall until register 1 has been updated 3 cycles later.
12) Subtract the value in register 4 from the value in register 3 and store the result in register 6 ($2 - 1 = 1$).
13) Store the value in register 2 (2^{30}) to memory word 2.
14) Store the value in register 5 (0) to memory word 3.
15) Branch to memory location 0 ($PC - 14$) if the value of register 6 is equal to the value of register 3 (true). This generates a branch stall until the instruction enters the writeback stage, then stalls for an additional cycle while the input to the fetch stage is refreshed. Processing resumes with instruction 0.

16) Add the immediate value 2 to register 1 and store the result in register 2. Since the branch immediately preceding this instruction is taken, it should not be executed.

Note that instructions 2 and 3 test for repeated branch hazards, instruction 6 tests for a double data hazard (both source registers as conflicts), instructions 7 and 8 test for a non-stall (WAW "hazard"), instructions 9 through 11 test for repeated data hazards, and instruction 15 tests for a simultaneous branch and data hazard. The other instructions test basic functionality from those in the supported list in the section I.

VI. Challenges

The greatest challenge faced in the design was dealing with hazards. Whereas branch hazards require a delay of 1 cycle before the stall (the branch instruction must not be stalled and has to propagate through the pipeline), data hazards require an immediate cessation of all processing (the hazard instruction cannot proceed through the pipeline). This led to a divergent approach to stalling involving two separate cases: branch stalls and data stalls.

For branch hazards, the desired stalling behavior was achieved by setting an indicator register to 1 when the branch instruction is fetched and resetting it to 0 when the branch exits the memory stage. While the register is set, NOPs are generated and the program counter remains static. This was relatively straightforward to implement once the method had been deduced.

Data hazards, however, require more sophistication. Because the instruction memory has already been queried by the time the decode stage can detect a data hazard, it must use wires to communicate with and control the PC. Furthermore, it must also backpropagate the hazard to the modules in the fetch stage to ensure that the fetch buffer is not updated and that the current output of the instruction memory is preserved. Thus combinatorial comparisons between the source registers indicated by the instruction in the fetch buffer and the semaphore register were used to ensure synchronization, though this solution required lots of troubleshooting before it worked correctly.

Of particular note was the trouble of getting the two hazard types to interact appropriately—that is, enabling functionality for branch hazards that are dependent on some instruction still in the pipeline. Many bugs were encountered in this process, but they were ultimately solved by fully divorcing data and branch hazards. It is suspected that many of the problems were generated by the initial approach to detect both types of hazards in the fetch stage, which led to an interference between the two signals. Additionally, making the data hazard and branch hazard signals fully independent (neither relies on the same set of combinatorial logic) seemed to fix some issues in this vein.