Computer Organization and Architecture

Marcel Morin
Computer Engineering, U3
260605670
marcel.morin@mail.mcgill.ca

Uday Sahni
Computer Engineering, U3
260614324
uday.sahni@mail.mcgill.ca

Stone Yun
Electrical Engineering, U3
260616314
stone.yun@mail.mcgill.ca

Richie Piyasirisilp
Computer Engineering, U3
260624968
master.piyasirisilp@mail.mcgill.ca

Abstract – In this document we will go over our design process and implementation of a MIPS pipeline processor in VHDL. This design was split into the following stages: Instruction Fetch, Instruction decode, Execute, and writeback. Our testing and integration of these stages will be described in depth in the following sections as will our optimization that we added on to enhance our pipeline.

I. OVERVIEW

The design problem was to implement a standard five-stage pipelined 32-bit MIPS processor. The proposed processor is able to implement a 27-instruction subset of the MIPS instruction set architecture. In addition, the processor implements early branching meaning that branch instructions are resolved at the decoding stage rather than the execution stage. This optimization reduces delays imposed by each branch instruction and thereby increases the overall performance. A brief overview of each stage will be provided in the following sections.

The instruction fetch stage consists of the program counter, instruction memory, and an adder. At the beginning of the program, machine code instructions are loaded into the instruction memory module. The adder and program counter work in tandem to reference the next instruction address. Since each instruction is 32-bits and MIPS uses byte addressing, the adder adds four to the program counter value to fetch the next instruction. There is also a 2:1 mux that selects between the adder output and the decoded branch address. This allows the processor to move to another instruction address if the program branches or jumps.

The core components of the decode stage are the instruction decoder and the registers module. The decoder interprets the fetched instructions and outputs the register addresses that the register module will use for referencing and outputting data. Since early branching was implemented, branch instructions are resolved in this stage. The decoder outputs the target address of branch instructions and sends it to the comparator, which decides if the program must branch/jump. If the program is required to branch, the target address is added to the current instruction address and sent back to the IF stage to update the program counter. Finally, there is a sign/zero extender for sign extending immediate values from 16 to 32 bits.

The execute stage consists of two comparators, two 4:1 mux's, a 2:1 mux, and the ALU. The ALU is the most important component of this stage since it performs the arithmetic operations necessary to execute the instructions. The ALU is capable of completing both I and R type instructions as determined by the inputted ALU op code. The 2:1 mux is used to toggle between feeding the ALU register contents, or an immediate value based on the decoded instruction. It is controlled by the decoder, which outputs a signal to indicate whether the mux should select the register data or the immediate value. Finally, the comparator and 4:1 muxes implements the data forwarding mechanism. The 4:1 muxes are connected to the EX, MEM, and WB stage. The comparator checks the register addresses to see if the current instructions makes reference to the same register address that is in the EX, MEM, or WB stages. It then sends a control signal to the 4:1 mux's so the appropriate data can be used during computation.

The MEM stage houses the main memory block and can either retrieve data on a load instruction or re-write the referenced memory address on store instructions (in which case, a write enable request has been submitted by the decoder and sent to the memory block). Furthermore, there is a 2:1 mux at the output controlled by the load signal that controls the output of the MEM stage. If a load instruction is expected, the data from the desired memory location is selected and sent to the register file for write-back. Otherwise, the output of the ALU is selected for writing back to the register file since that data will be the result of executing the instruction.

The last stage of the processor is the WB stage which forwards the resulting data of an instruction to the required location such as the register file and the EX stage (for data forwarding). *Figure 1* shows the high-level organization of our processor [1]. It details the connections and signals within and between each stage.

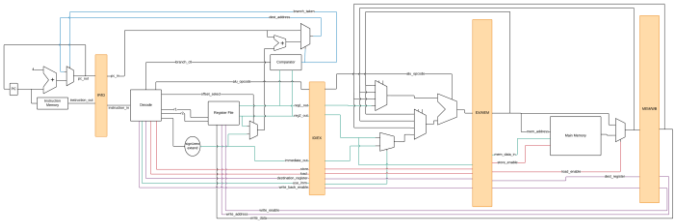

Fig. 1. Block diagram of the pipelined processor. Components between the latches (in orange) were organized into module blocks. The top level module manages signals and data that are used by multiple stages.

II. PROCEDURE

A. Design Approach

As this is a large project with five stages and several components per stage, a multi-step design approach was implemented. First, a complete block diagram was needed to detail all the components needed for each stage, as seen in *Figure 1* [1]. During the review of this block diagram, course material and online resources were reviewed. This allowed for a better understanding of which components would be needed and the type of control logic required for implementing features such as data forwarding, branching, and memory addressing (for load and store). After verifying with the course material and online sources that all the necessary components and control signals were present, a document was created to detail each component, stage by stage.

The descriptions of all components seen in the block diagram were written from scratch and implemented using VHDL. Each component within a stage was implemented asynchronously to allow each stage to be completed within one clock cycle. Upon completion of the components of a stage, they were integrated together as a single block in a higher-level module representing the stage. This module contains all the input and output signals of its respective stage and connects the ports of each component to the necessary signals. It should be noted that the latches dividing each of the five stages were written as modules of their own and were clocked. They acted like "gates" that controlled the data between stages, storing data values from one stage and presenting them to the next stage on the following clock cycle. These latches are essential for the implementation of a pipeline. Storing the results of each stage in a latch frees up that stage to perform subsequent instructions.

Finally, after successfully integrating the components within each stage, a top-level module was designed for connecting the entire pipeline together. In this top-level module, inter-stage signals such as forwarding signals, branching signals, write enabling, and write back data and signals were implemented and connected to their required ports.

B. Testing and Evaluation

The pipelined processor was tested in several stages in ModelSim. The first testing stage was to evaluate each individual component separately to ensure that they behaved as expected. For example, the adder component was tested by forcing its two inputs and its enable signal and verifying that the correct value was outputted. It was important to test each component individually before testing them together because problems could be easily determined and resolved.

The next testing stage was to evaluate each of the five higher-level pipeline stage modules with a test bench. This was done to verify that given the correct inputs and signals, each pipeline stage would be able to produce the correct outputs. For example, the test bench written for the instruction fetch stage set an enabling variable to force which instruction would be performed next. The program counter and destination address inputs were tracked to check that they only changed when they were expected to, depending on the test case. The stage's outputs were also tracked to ensure that the fetching stage produced the appropriate program count and instruction to be fed into the decoding stage.

Lastly, all five stages were tested together to evaluate the pipelined processor as a whole. Similar to the individual pipeline stages, the final product was tested using a test bench and a MIPS test program. A set of MIPS instructions composed of arithmetic, logical, transfer, shift, memory, and control-flow operations was inputted and processed in the pipeline processor. Variable delay cycles were also added between the instructions to provide a more detailed test bench. At the end of the program, the contents of the register and the contents of the memory block were both written to respective .txt files where starting with line 0, each line contained the data of its respective register or memory address. This was used to verify that all the operations and procedures performed as expected and were written back correctly. Thus, at the final stages of our design, the contents of the register file and memory file were used as an indicator of the processor's correct operation.

Even though the entire pipelined processor was tested from each individual component to full stages, there were still multiple problems when testing all five stages together. The two main issues that persisted were in the port mapping and latency between stages. To fix the issues in the port mapping, every associated component was checked and remapped so that all the wiring connected correctly. As for the latency problems, the components in question were reviewed and were found to have design problems that lead to timing errors. Upon successful debugging, the contents of the register and memory .txt files turned out as expected, and the resulting speedup of early branch detection was observed in the waveforms produced by the ModelSim simulations. Details of the optimization method are discussed in the following section.

III. OPTIMIZATIONS

There are several ways to optimize a pipelined processor including caching, branch prediction, and early branch detection. For this project, early branch detection was chosen. This method allows branching instructions to be detected and resolved at the ID stage rather than at the EX stage of the pipeline. This is possible because as soon as the decoder has finished interpreting a branch instruction, it has extracted the target address from the machine code. Therefore, instead of waiting until the EX stage to update the program counter, it can

be updated immediately following the instruction decode. The implementation of the early branch prediction is outlined in the *Overview* section. Early branch detection speeds up branching instructions by two clock cycles (as learned from the course material). According to the SPEC92 benchmark mentioned in class, branching instructions constitute 20% of all instructions [2]. Assuming the standard of five clock cycles for a regular instruction to be executed, early branch detection speeds up branching by a factor of 2.5. Using Amdahl's Law (see Equation 1 below), an overall speedup of 1.136 is observed [3]. To test this, a test bench was written for the pipeline without early branch detection and a test program was simulated. After that, the same test program was run again but with the early branch detection implemented. The program counter was then observed and noted. After extensive debugging of the branching mechanism it was verified that early branch detection speeds up branch predictions by two clock cycles.

$$Speedup_{overall} = 1/[(1-F)+(F/S)] \quad (1)$$

Where F is the fraction of branching instructions and S is the speedup factor.

IV. Conclusions

We designed and optimized a standard five-stage pipelined 32-bit MIPS processor that is capable of performing a 27-instruction subset of the MIPS instruction set architecture. Early branch detection was the chosen optimization method, resulting in an overall speedup of 1.136.

If given the opportunity, more thorough test benches could have been implemented to test each component more exhaustively. This would have reduced the amount of problems faced when running the pipelined processor as a whole. But

References

[1] W. Chang, D. Lavoie-Boutin, M. Lashari, and S. Sheriff, "MIPS 5 stage pipeline", GitHub, 2016. [Online]. Available: https://github.com/dlavoieb/ecse-425. [Accessed: 15- Apr- 2018].

[2] Hayward *et al.*, "L02- trends", myCourses, 2018. [Online]. Available: https://mycourses2.mcgill.ca/d2l/le/content/298520/viewContent/3646554/View. [Accessed: 15- Apr- 2018].

[3] Hayward *et al.*, "L04- caching", myCourses, 2018. [Online]. Available: https://mycourses2.mcgill.ca/d2l/le/content/298520/viewContent/3679107/View. [Accessed: 15- Apr- 2018].