Technical Report - ECE350L Processor

Stefani Vukajlovic

November 2018

Contents

1 Introduction **3**

2 Subcomponents **3**
2.1 Regfile . . . 3
2.2 ALU . . . 3
2.3 Mult/Div . . . 3
2.4 Memory elements . . . 3

3 Full Processor **4**
3.1 Design implementation details of each stage (F, D, X, M, W) . . . 4
3.1.1 Fetch Stage . . . 4
3.1.2 Decode Stage . . . 4
3.1.3 Execute Stage . . . 4
3.1.4 Memory Stage . . . 4
3.1.5 Writeback Stage . . . 4
3.2 Handling hazards . . . 4
3.3 How fast the processor can be clocked . . . 5
3.4 Bypassing logic . . . 5
3.5 Efficiency details . . . 5

4 Instruction Implementation **5**

5 Process **5**
5.1 Why did you choose your implementation? . . . 5
5.2 How did you test your implementation? . . . 5
5.3 What errors exist in your processor? . . . 5
5.4 Challenges . . . 6
5.5 Main learning points . . . 6

6 Conclusion **6**

1 Introduction

This report documents all the design decisions, how the individual components were implemented and how the whole design was integrated together. Additionally, the report notes the possible improvements and flaws of the processor.

2 Subcomponents

2.1 Regfile

The register file is array of 32 32-bit registers that are implemented using d flip flops. The register file has two read ports and one write port. In order to select correct register decoder has been implemented using AND and NOT gates. It decodes the 5 bit input into a correct register port number that the data will be written to. Based on the read controls the regfile outputs the data from chosen register and this has been implemented using tristate buffers. Initially the design implemented 5 to 32 multiplexer that was generated using 10 2 to 4 multiplexers, however the transition to the tristate buffers was made to ensure the proper runtime. This design is fully functional for the 20ns clock period. In the final processor register file is clocked on the negative edge of the clock.

2.2 ALU

The ALU implements the hierarchical carry lookahead adder, bitwise AND and OR, and barrel shifter. Initially, the design incorporated 32 bit lookahead adder using 4 bit lookahead adders. However, the testing results showed that this design was not fully functional for a 20ns period, thus the design was reviewed and the current design implements a 32-bit hierarchical carry lookahead adder that was build using 4 8-bit hierarchical carry lookahead adders. A hierarchical carry lookahead adders are used to further reduce the delay in computing the carries by computing the generate and propagate functions using the full bit adder and then using the second level of lookahead circuits to determine carries between the blocks, which to decrease a delay in the summation of two binary numbers. The delay in calculation the caries is now $\Theta(logN)$, which is the overal delay of the adder. The design decision to move from carry lookahead to hierarchical carry lookahead, as opposed to carry save or carry select was made partially due to the reason that the carry lookahead was already implemented and moving to hierarchical did not include too much additional work and partially due to the functionality of carry save and carry select. Carry save is mostly used for adding multiple numbers and our processor only adds two numbers at the time. Carry save has much better performance then hierarchical CLA when it comes to adding big numbers like 1000 bit numbers but since the ALU only performs the operations on 32 bit numbers this was also not a concern. Carry select would introduce additional hardware complexity, which was not necessarily needed since hierarchical carry lookahead adder is also fully functional for 20ns period. There is additional hardware cost for the hierarchical CLA compared to ripple carry or regular CLA, however the trade of the performance was worth the hardware investment. Additionally, the adder also outputs the lessThan, notEqual adn overflow signals. Bitwise AND and OR uses generate loops to AND and OR the individual bits and outputs the result. Barrel Shiter implements shift left logical and shift right logical. The 5-bit control input determines the shift amount. The barell shifter uses the input bits as select bits to the muxes that either shift 16, 8, 4, 2 or 1 bit or do nothing.

2.3 Mult/Div

Multiplier implements Booth's algorithm to compute the product of the two 32 bit numbers, which means that the multiplication takes 32 cycles. The divider implements the design similar to the better divide circuit mentioned in lectures with one deviation, it has a separate register for the quotient as opposed to writing the quotient into the Remainder Register, like in the simple divide circuit, however all the logic alligns with the better divide circuit design. Divider also takes 33 cycles to complete. Divider hadles the division by negative number similar to the Naive algorithm for multiplication, where it converts the divider to its 2's complement and then it also takes the 2's complement of the quotient. Both multiplier and divider output exceptions and result data ready signal which signals that the result can be read.

2.4 Memory elements

The processor implements the Verilog megafunctions for imem and dmem, which are clocked on a negative edge. The pipeline registers are implemented using the array of d flip flops, and they were clocked on the positive edge of the clock to ensure that the data has enough time to propagate from the outputs of the latches to the functional units, memory elements and register file.

3 Full Processor

The processor is one-wide, five stage pipelined MIPS based processor. The processor implements full bypassing, stall and flush logic to achieve better performance. It also handles all hazards. For branches, the processor always assumes that the branch is not taken and flushes the pipeline if the prediction is false.

3.1 Design implementation details of each stage (F, D, X, M, W)

3.1.1 Fetch Stage

The Fetch stage write into the PC register (current PC) it computes the nextPC value and wires it back into the input mux to the PC register. The mux chooses whether the nextPC is PC+1, or a target or PC+1+N, depending on the instruction. Furthermore, the address is sent to the imem and the instruction is obtained. The obtained instruction and the next PC values, i.e. PC+1 are stored into the FD latch.

3.1.2 Decode Stage

Decode stage takes the instruction output of the FD latch and decodes it into series of controls that guide the execution of the following stages in the pipeline. For example, the instructions were first interpreted in terms of opcodes, source and destination registers, target values, ALU opcode etc. Based on these the other cotrols such as MemoryRead, MemWrite, regWrite and similar were set and used to control when and how the instructions are executed. Control signals are being saved in the latches. In this stage the data from source registers is retrieved and also saved into the DX latch. After the decode stage is done all the controls are saved into the DX latch, some are used in execute stage some are propagated to the XM latch, etc.
The processor also implements branch and jump calculation in the decode stage to optimize performance, since those are simple enough instructions that do not necessarily require the ALU functional unit, thus the output of next PC calculation based on jump or target goes into the PC mux input. The calculation of branches and jumps in decode reduces the penalty for miss-predicting the branch, however it introduced additional hardware complexity as the adder, lessThan and notEqual circuits were added to the decode stage. Additionally, the execute stage contains sign extension for immediate.

3.1.3 Execute Stage

Execute Stage contains the ALU, that now includes the mult\div unit. The inputs to the ALU are chosen based on the hazard bypass control outputs. And if the inputs is to come from the DX latch then input B into the ALU is determined based on the instruction controls, it could be data from register file or an immediate. Execute and memory controls, along with ALU result, ALU input B, rd and rs register numbers are saved into the XM latch.

3.1.4 Memory Stage

In the memory stage the data from memory is retrieved or written into the dmem, depending on the instruction. SOme instructions are not accessing dmem at all, which is why the ALU result is propagated to the nextlatch, MW latch, along with data read from the memory.

3.1.5 Writeback Stage

In this stage, the data which is written to the register file is chosen and if the data is not to be written to the regfile then the enable for reg file is set to zero in this stage.

3.2 Handling hazards

Since the branch, jump calculation happens in the decode stage, it is possible that data hazard will occur with ALU instruction immediately preceding the branch and the processor will have to stall. The worst case stall is one cycle if the immediately preceding instruction is not a load, if it as load then the processor will have to stall for two cycles. The stall is a result of decode stage happening before the execute stage of the preceding instruction. Furthemore, the only time that we need to stall if the instruction is not branch is if we had a load followed with an instruction that uses the output of a load as source, then the processor stalls for one cycle. In the case of the jumps and branches, the processor also flushes the FD latch, i.e. flushes the instruction from the fetch stage when it needs to jumo to certain address that is not $PC + 1$. There is additional hazard when it comes to the mult\div. The processor needs to stall until the data is ready, which could be either until the exception is raised or data calculated which is 33 cycles.

3.3 How fast the processor can be clocked

All the units were design to be able to run on a 20ns clock. However, there are certain computations that take a little bit more time which the time constraints did not allow for improvement and those include the calculation of the quotient in the worst case scenario. Other components like the ALU and regfile were tested and proved to be fully functional on the 20ns clock period. Propagation of the signals in the pipeline was also designed to align with 20ns clock cycle.

3.4 Bypassing logic

Bypass logic allows ALU inputs to come from the latch deeper in the pipeline, as opposed to acquiring it from the register file. This reduces the number of overall stalls for a significant amount and thus increases IPC significantly. This processor implements bypassing from the beginning of the writeback and memory stage to the inputs of the ALU as well as from the from the writeback to the memory.

3.5 Efficiency details

Having branch and jump calculation in the decode stage required additional hardware, however it trades off with significantly better performance as only one instruction is being flushed, the one currently in the fetch. The bypass and hazard detection logic introduces more hardware complexity however as explained earlier results in significant performance improvement. Furthermore, the utilization of hierarchical carry look ahead adder required additional hardware complexity, but insured that the calculations are fast enough and since adder is one of the most used units (the only functional unit) it certainly affects performance significantly and is worth the hardware investment. Moreover, full bypassing allowed for significant reduction in number of stalls which further contributed to better performance and efficiency.

4 Instruction Implementation

R instructions were among the easiest to implement as their the ALU opcodes matched the ALU design and they for the most part followed the standard path through the pipeline. On the other hand, I, JI and JII instructions included additional logic into their implementation. I instruction required the logic to choose immediate as an input to the ALU, JI and JII instructions included logic to calculate next PC and to check branch conditions.

5 Process

5.1 Why did you choose your implementation?

The implementation depended on several factors and those include efficiency, hardware complexity, time complexity and time to finish the implementation. Due to the limit time for the processor was 1 wide and without a branch predictor, while having more time would probably lead to implementing a branch predictor to further optimize performance and then the next level would be implementing the two wide pipeline. Some of the designs, like Carry Save adder introduced additional hardware complexity but not enough performance trade offs to be implemented. On the contrary, early branch calculation and jump PC propagation allowed for better efficiency, as it reduced the number of flushed instructions.

5.2 How did you test your implementation?

Each module implemented was tested as it has been implemented using the Verilog Waveforms. Additionally the ALU and mult\div were tested using the testbenches to further ensure the correctness of the modules and to to understand which inputs were incorrect and what unit needed to be reviewed. The final processor was test using the Waveforms and different mif files that varied the instructions and most of the possible hazard and bypassing situations.

5.3 What errors exist in your processor?

There are few errost that currently exist in the final processor design. First error is the calculation of next PC for jr instruction. The problem is that the source control for PC evaluates one cycle to early or in other words the register value of jr is late to the input mux into the PC register, this causes the next PC to be wrongly evaluated. The solution to this problems was found by testing the jr instruction in the waveform

where the cycle of miss-evaluation was detected. The problem was attempted to be solved by implementing d flip flop which did not work, toggle flip flop was also a failed attempt too. There were multiple attempts to synchronize these two signals but days of effort rendered useless. Another issue that was not resolved was full implementation of mult\div into the ALU. More precisely the stalling of other stages was based on the $data_resultRDY$ signal, which seems to never be triggered to a one ones the result is ready, which interesting since the mult\div $data_resultRDY$ signal worked as when it was a separate unit, which suggest that there is delay from mult\div to ALU outputing it. However the way this was implemented is that the mult\div output is directly the output of ALU, which makes this phenomenon harder to understand. The lack of time inhibited the resolution of this problem. Furthermore, very few exceptions in the ALU and mult\div were not properly assessed which suggest that some exceptions in the processor will not be properly assessed too. TO improve these more testbeches should be written to exploit the error and to determine the cause.

5.4 Challenges

- Register File had initialization errors which led to the assumption that there existed a problem with run time. The challenge was solved by implementing tristate buffers instead of multiplexors.
- Run time problem of the adder. The solution was implementation of faster adder that was built upon the current one (carry lookahead adder -> hierarchical carry lookahead adder).
- it was challenging to implement jump and branch next PC bypassing into the PC source mux. This was solved by implementing memory elements that delayed certain signals in order to synchronize the processes.
- Overall timing of the processor was challenging because the signals needed to be ready when they were read to ensure the correctness. This was solved by triggering different parts of processor at different clock edge.

5.5 Main learning points

This processor served as a great learning experience as all the theory was implemented and the challenges that are not mentioned in the books arouse and lead to better understanding of the concepts. Primarily I would accentuate the importance of timing, which was not something I thought about a lot in the past while learning these concepts and how these timing concepts lead to issues and different ways to overcome them.

6 Conclusion

In conclusion, this project was a successful learning experience, as all the concepts learned are much better understood now that they were implemented in the Verilog HDL. Debugging the project was much harder then it is usually for software programming which led to huge time miss predictions in terms of planning. As a result, few components were not completely functional, which is also a valuable experience that will help plan the future projects better.