2023

INF-2200 Assignment 2 MIPS

HTTPS://GITHUB.COM/UIT-INF-2200/MIPS-TIRIL_KRISTINA

KRISTINA KUFAAS (KKU020@UIT.NO, KKUFAAS) AND TIRIL LAPPEGÅRD (TIRIL.LAPPEGARD@UIT.NO, TIRLAP03)

Assignment 2 - MIPS

October 12th, 2023

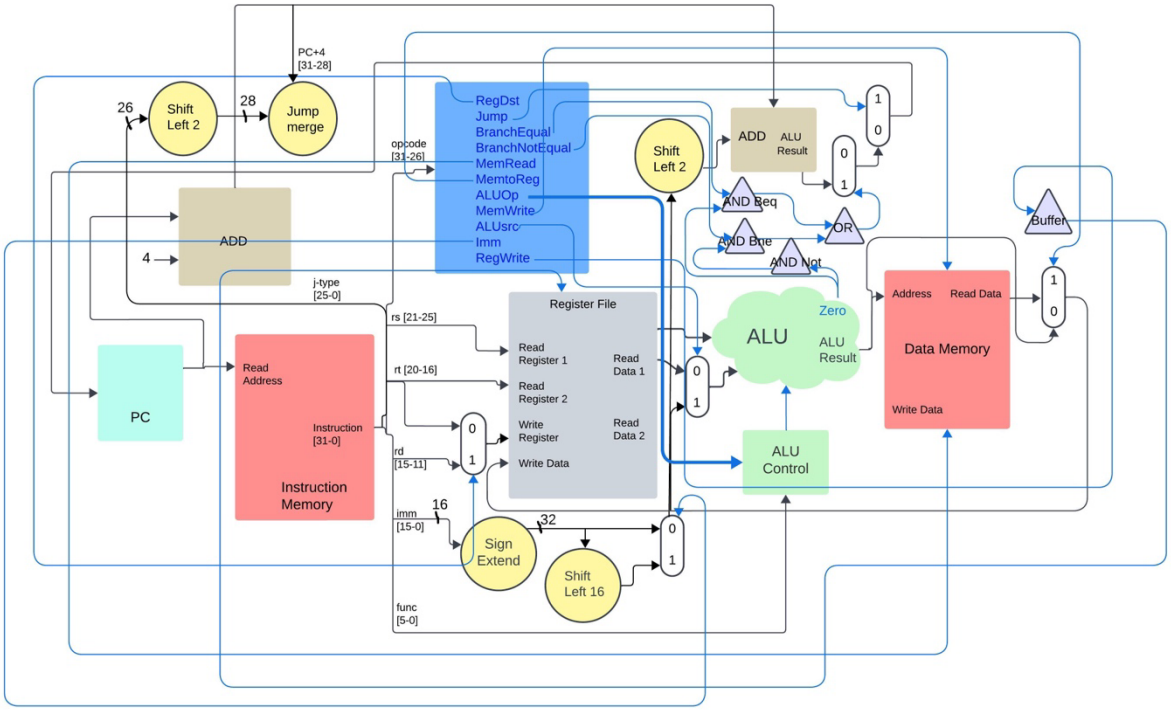
1. Introduction

The Modified Instruction Set Architecture (MIPS) is streamlined, highly scalable RISC architecture that is available for licensing, renowned for its clear design and reduced instruction set computing principles[1].
Our project in Python aimed to implement a comprehensive simulator for the MIPS architecture, able to visualize the nuances of its execution. By simulating the entire instruction lifecycle—from instruction fetch to execution—the simulator shows how data flows through registers, memory, the arithmetic logic unit (ALU), Control Unit and other components. In general, this project embodies the amalgamation of theory and practice, enabling to bridge the gap between abstract MIPS concepts and their tangible manifestations in computer operation.

2. Methods

2.1 Summary of the Datapath and Controllers

We used the following diagram for single-cycle datapath implementation, the main idea of which was borrowed from figure 4.24, Patterson's Computer organization and design, 5th edition.



Some extra CPUElements we implemented:
Buffer gate acts as a temporary storage for the RegWrite signal. Its primary purpose is to hold onto the value of this signal for one cycle, ensuring synchronization and timely data transfer, otherwise Write Data will be set to 0.

AND Beq Gate and AND bne Gate are being used in the decision-making process for the "branch if equal" (**beq**) and "branch if not equal" (**bne)** instruction. It determines whether a branch should be taken based on equality conditions. If both the condition and the beq/bne instruction signal are true, the branch is taken.

OR Gate takes the inputs from the two AND gates above and produces an control signal, depending on wether the branch is equal or not equal.

AND NOT Gate flips the input values to be the opposite value.

All these gates, when used in conjunction, allow for intricate control flows and decision-making processes within the CPU, particularly when handling branch instructions and ensuring that data flows correctly through the datapath.

2.2 Simulator implementation

The simulator implemented for the MIPS architecture, as exemplified by simulatorIFIDEX.py, is a representation of the CPU's datapath and control mechanisms. The simulator is built around a main class, MIPSSimulatorIFIDEX, which initializes and connects various CPU elements to replicate the flow of a real-world MIPS processor.
Every CPU element is instantiated, with additional functions meticulously designed to capture their precise logic and behavior, while the connectCPUElements function is a pivotal piece, establishing the interrelationships between these components. The simulator's main loop (tick function) progresses the instruction processing cycle-by-cycle, advancing the PC and simulating the execution flow of the MIPS pipeline. This intricate design, enriched with additional logic functions, captures the essence and complexity of the MIPS architecture.

2.3 Test cases

We used unit tests for each and every element, using various inputs similar to expected outputs.
During the implementation of the single cycle simulation, we used the add.mem file to check our implementation and making updates to our code. With the add.mem file it was easy to follow the instructions and check if our code acted the way we wanted it to, and if something didn’t work, we could easily see where the problem was. In the end, when the add.mem file ran as desired, we moved on to the Fibonacci.mem file and it also ran as it should.
Notification to the reader and user of this MIPS implementation; the BREAK message is for our personal gain and is not meant as an insult.

3. Discussion session

3.1 Data hazard handling

In a pipelined CPU design, data hazards arise when an instruction depends on the result of a prior instruction still in the pipeline[2]. A typical situation is when an instruction depends on the result of a previous instruction but that result is not yet available. Our approach to managing data hazards is to primarily highlight our hardware implementation through the introduction of the IF/ID pipeline register.
Our code includes the sketch of IF/ID pipeline register, exemplifying the concept. We used the final datapath and control diagram as an example [3]. This register should temporarily hold instruction details fetched in the IF stage, ensuring it's readily available for the ID stage in the subsequent cycle. By creating a buffer zone, it reduces data hazards stemming from instruction overlaps. Hypothetically extending this, registers between other stages (like ID/EX, EX/MEM, MEM/WB) can similarly manage hazards by allowing intermediate results to be stored and accessed without waiting for an instruction to entirely traverse the pipeline.
A pipelined processor divides the single-cycle processor into five stages, and each command can be executed simultaneously, one in each stage. Since each stage contains only one-fifth of the processor's total logic, the clock frequency can be almost five times higher. In an ideal case, command latency will not change, and throughput will increase five times.
Real microprocessors execute millions and billions of instructions per second, so performance is more important than latency. Pipelining requires some overhead, so in real life throughput will be lower than ideal, but in any case, pipelining has so many benefits and is so cheap that all modern high-performance microprocessors are pipelined[4].

3.2 Controller hazard handling

Control hazards occur due to the pipelining of branches, i.e., situations where the correct next instruction to fetch and execute is unknown because it's based on a decision (branch) that hasn't been made yet. When a branch instruction is encountered, the subsequent instruction's address might be uncertain, leading to a control hazard. One approach to handle branches is to use prediction, for instance to predict always that branches will be untaken [5]. It means the system assumes, by default, that every branch will continue executing the next sequential instruction, rather than jumping to a different target. If this prediction is wrong, corrective action (like flushing the pipeline) is needed once the actual branch result is determined.
In our implementation, the datapath didn't have registers to hold intermediate values and states, but we had the following idea related to IF/ID Register:
IF/ID Register would store the instruction fetched in the IF stage. The Hazard Detection Unit would monitor the pipeline and detect potential problems (hazards) that might occur due to the overlapping of instruction execution. The HDU can signal to pause (stall) the pipeline to wait for the necessary data. The HDU, upon detecting a control hazard, can use the "IF Flush" signal sent by the control unit, which effectively cancels (or ignores) the instruction currently in the IF stage, often replacing it with a "no operation" (NOP) instruction, especially if a branch instruction decides to take a different path than initially assumed.
The combination of the IF/ID register, HDU, and "IF Flush" signal ensures the pipeline operates efficiently, handling potential issues without major disruptions to the flow of instructions.

The "IF Flush" signal and Hazard Detection Unit (HDU) themselves are not branch predictors, but they are mechanisms to handle the consequences of branch decisions, especially wrong predictions.

3.3 Summary of known bugs and problems, including a description of which of the provided tests fails.

PrintAll method in Register file and DataMemory uses 2 functions in Common for printing decimal numbers which results in displaying negative numbers in Fibonacci numbers.
Due to lack of time, we couldn't enjoy all opportunities of pytest.

4. Large language model use

Throughout the development of our MIPS simulator, we extensively utilized OpenAI's large language model, ChatGPT. The model served as an invaluable consultative tool, assisting us in various stages of the project. From understanding intricate nuances of the MIPS architecture to troubleshooting and debugging our implementation, ChatGPT consistently provided insights based on its vast knowledge repository. Its capability to interpret and suggest solutions for code-related queries was particularly beneficial. Furthermore, the model played a crucial role in enhancing our documentation. It assisted in drafting sections of the report, offering succinct summaries and explanations. The real-time feedback and guidance from ChatGPT not only streamlined our development process but also elevated the quality and accuracy of our simulator. The collaborative synergy between our team and ChatGPT underscores the transformative potential of integrating advanced AI models into software development workflows.

5. Conclusion

Even though we didn't achieve a full implementation of the proper pipelined datapath, the journey provided invaluable insights and hands-on experience. This endeavor deepened our comprehension of the intricate mechanisms underpinning pipelining. Every challenge faced and solution explored enriched our understanding, proving that the process itself is as crucial to learning as the final result.

References

[1] Mips(2023). *MIPS Architectures,*
Available from: https://www.mips.com/products/architectures/[Accessed 12 October 2023]

[2] Patterson, D., Hennessy, J. (2014). *Computer organization and design* (5st edition), p. 278. Elsevier

[3] Patterson, D., Hennessy, J. (2014). *Computer organization and design* (5st edition), p. 325. Elsevier

[4] Harris, D. (2018). *Digital Design and Computer Architecture* (2nd edition), p. 479. Elsevier

[5] Patterson, D., Hennessy, J. (2014). *Computer organization and design* (5st edition), p. 283. Elsevier