1. Introduction

For the final project of ECE411 we designed and instantiated a 5-stage pipelined, in-order RISC-V CPU core for the rv32im variant ISA. The design consists of a processor unit with multiple extensions to boost its performance, including a branch predictor, an instruction prefetcher, and two 1-level caches for RAM data. In this process we gained a significant boost in understanding each part of a general architecture of CPU structure, while also being intensively trained with designing, implementing, verifying and optimizing a piece of digital hardware.

2. Project overview

The project was divided into four incremental checkpoints, with different parts of the design being gradually added to the structure: starting with a simple pipelined CPU, we needed to design additional parts that enable support of a range of features: control flow instructions (such as br and jump instructions, as opposed to a sequential program), data forwarding, complicated RAM interactions, and advanced ISA extensions.

The workload was roughly divided among group members according to relatively independent sub-systems. Each person would be in charge of roughly ⅓ of the total design, namely, one of the three following parts: the pipelined datapath; the CPU control logics; or, the memory interface. Collaborations and mixed contributions on many parts in the design also occurred very frequently. For each checkpoints, the above parts needed to be unit-tested to make sure they align well with the design before being integrated into a whole design and tested under a program-driven testbench.

3. Design description

(a) Overview

For each of the checkpoints, the goals were the following.

Checkpoint 1: Implementing a basic pipelined CPU structure;

Checkpoint 2: Implementing caches; extending the CPU to handle control hazards and data dependencies;

Checkpoint 3 and 4: Verifying the above designs; implementing advanced features to support multiplication instructions and optimize the performance.

(b) Milestones

i. Checkpoint 1

**The Pipeline Design.** Unlike a single-cycle design, a CPU that supports instruction pipelining can have different instructions running in each part of the architecture at the same time, boosting the clock speed and reducing the waste in hardware.

In our project, the pipeline consists of five stages: prefetching(IF), decoding(DE), execution(EXE), memory(MEM), and write-back(WB), interconnected by registers to pass down necessary signals. An instruction is first read from the memory subsystem by the IF stage, then decoded, where the control unit of the CPU would decide the operations of the remaining three stages for the instruction, and pass the control signals down the pipeline in a control word (CW) structure. The EXE stage then carries out necessary ALU operations, and the MEM stage interacts with the data memory, if required by the instruction. Finally, the WB stage handles the instruction commits and any writes to the register file.

The pipeline design for this checkpoint is a basic one, only supporting in-sequence instructions without any control-flow operations available.

ii. Checkpoint 2

In the first part of Checkpoint 2 we instantiated additional features to the CPU.

**Hazard detection.** We expanded the CPU design to support BR and Jump instructions.. A static prediction is employed: after encountering a BR/JUMP instruction, the pipeline continues to load the instructions located immediately after them in the program. On actively-taken Branches or Jumps, contents of the pipeline are flushed, and the target PC should be reloaded.

**Forwarding.** The new pipeline design also supports forwarding to save time on data dependencies. On decoding an instruction, the control unit searches all its proceeding instructions in the pipeline for any register-based data dependencies. On finding any, it will enable forwarding, taking the data in need directly from the immediate stage where it is generated, and feed directly into the EXE stage, without having to wait for it to go through the write-back stage. A large amount of CPU cycles are saved in this way.

We also worked on a new **memory subsystem** build from the cache implementations from the previous MP. Two SRAM caches, one for caching instruction, the other for data, are connected to the CPU, interacting with the main memory through an arbiter. Since the new memory subsystem, unlike in Checkpoint 2, is no longer same-cycle responding to requests, the IPC of the CPU drops below 0.5.

iii. Checkpoint 3

(c) Advanced design options

i. Option 1 (Branch Prediction and Prefetch Buffer)

A. Design

The branch prediction is based on two major components: a Branch History Table (BHT) and a Branch Target Buffer (BTB). They hold information from a range of most recent branch operations. For each cached BR instruction, the BHT holds as a bit-vector its recent history of activities, i.e. being taken vs. being not taken, along with the CPU’s prediction of it, also as a bit-vector. The BTB holds the targeted branch address for each BR instruction. Both of them are FIFO, with the PCs of instructions as keys. When a BR instruction is decoded, the control unit will search for its PC in the two buffers. If it is present in the records, then a prediction of whether the branch would be taken is generated based on the past histories of predictions and branch activities of the instruction, and either the branch target instruction or the instruction immediately next to the branch will be loaded into the pipeline, based on the prediction. Else, the BHT and the BTB would record the current BR instruction, and predict it not to be taken. When the instruction reaches EXE stage, the pipeline would flush if the prediction is false, or proceed normally if not.

A similar structure is used to save time for Jump instructions. Unlike Branches, however, the predictions of Jumps are unconditional.

We also implemented pre-fetching, which enables the instruction cache to read from the main memory when the main memory is free. In this way delays can be eliminated from the exemption of waiting for memory responses between instructions.

B. Performance analys

The performance efficiency is boosted with branch prediction and prefetching. With a 8-entry BTB/BHT, the average IPC while running M-extension Coremark grew from 0.30 to 0.338, with 21948 fewer pipeline flushes and 65998 fewer stalls, at a cost of 8848 unit areas and roughly 7% of total power usage, which is considered worthwhile.

ii. Option 2 (Advanced Multiplier and Basic Divider)

A. Design:

Most of the effort of designing execution units that would support multiplication and division went into designing the multiplier. Other than the execution units, simple(ish) logic needed to be implemented to support the multiply/divide instructions and handle edge cases(signed/unsigned, overflow, divide by zero…).

Most algorithms for increasing the speed of division involve using floating point operations. In a more robust processor floating point arithmetic would already be part of the design, but in this far simpler design that wasn’t part of the project. So for that reason a simple shift and subtract algorithm was used. You take your numerator and compare it to your denominator, subtract if the denominator is larger, and then shift your numerator over so that you can continue the process.

Multiplication was a bit trickier. There are several good algorithms for binary multiplication, although the one that popped into my head first was Karatsuba’s Algorithm. I designed it, and it worked. Then I decided I had better try to look into different algorithms as I was told that Karatsuba’s was a software solution and wouldn’t perform well in hardware. This is true the first version of my Karatsuba multiplier used recursive functions to build out the layers. So I then researched the recommended algorithm, the Wallace Tree. However, I found that the Wallace tree took up a lot of resources and wouldn’t be much more performant than Karatsuba anyway. So I redesigned my Karatsuba multiplier to run without any recursion, and achieved a 7 cycle multiplier. However, it took up a lot of space and once it came time to reduce the size of the design we felt we needed a smaller design. So upon resynthesizing my original design I found that not only was it smaller than the second version, but I could also get it to complete the multiplication in only 3 cycles. Karatsuba works by splitting the operands half over and over until you reach a base case, 1-bit for us. Then a simple AND multiplies two 1-bit operands. Then the result is sent to the fifth layer which recombines and adds several of these 1-bit operations to for the 2-bit operation. This pattern continues all the way up to the first layer, which combines four 16-bit operations into a 32-bit result.

B. Testing:

Testing was fairly straightforward: write a simple test program to test each of the variations of the multiply and divide instructions(mul, mulh, mulhu, mulhsu, div, divu, rem, remu). Testing revealed improper handling of edge cases for both division and multiplication, but those were fairly simple to fix. Another issue was making sure control could recognize stalls from execution since previously all operations were instant and execution never needed to stall the pipeline. After testing the basics of multiplication and division, the multiplication version of coremark served as the final test for the M extension. This mostly went off without a hitch, with the exception of missing and edge case when signing and unsigning results for multiplication.

C. Performance Analysis:

The divider performed as expected, 32 cycles, but no timing issues to speak of. It didn’t take up too much space either, only about 1100 units. The multiplier on the other hand took up 8500 units, but performed operations in only 3 cycles. Timing was a little hard to judge. It couldn’t hit basic timing in a single cycle(basic timing is 10000ps clock period), but split into three it did meet timing. However, it didn’t even appear in our critical path due to very poor optimization to the timing of control word generation, which we unfortunately didn’t have time to address. Power wise the multiplier was a little power hungry accounting for 4.5% of the overall power draw of our core(340uW / .762mW). However, the dead simple divider only sipped 1.1% or 87uW. All of this is well worth it when the runtime of coremark was more than halved.

4. Conclusion

In conclusion, the development of our 5-stage pipelined, in-order RISC-V CPU core for the rv32im variant ISA has been a comprehensive journey that not only deepened our understanding of CPU architecture but also provided hands-on experience in designing, implementing, verifying, and optimizing digital hardware. The project's incremental nature, divided into four checkpoints, allowed us to systematically build and enhance the CPU core's capabilities.

The initial checkpoint focused on establishing a basic pipelined CPU structure, paving the way for subsequent advancements. As we progressed through the checkpoints, additional features were integrated, including hazard detection for control flow instructions, data forwarding to optimize data dependencies, and the incorporation of a memory subsystem with two 1-level caches. These enhancements significantly contributed to the overall performance of the CPU core.

The third checkpoint introduced advanced design options, showcasing our commitment to pushing the boundaries of our CPU core's capabilities. Branch prediction and prefetch buffering were implemented, utilizing a Branch History Table (BHT) and a Branch Target Buffer (BTB) to improve instruction fetching and execution. Furthermore, the addition of an advanced multiplier and a basic divider enriched the CPU core's arithmetic capabilities, with detailed attention given to design, testing, and performance analysis.

The performance analysis of our design options revealed both successes and challenges. The branch prediction and prefetching mechanism effectively reduced instruction fetch delays, enhancing overall efficiency. The multiplier, despite its resource-intensive nature, demonstrated remarkable speed in executing multiplication operations, while the divider provided reliable performance with minimal power consumption.

Throughout the project, collaboration among team members was evident, with each member responsible for specific subsystems, yet frequent collaboration occurred for the success of each checkpoint. Despite the challenges faced, such as handling edge cases and addressing timing issues, the final implementation showcased a well-integrated and optimized CPU core.

In summary, the ECE411 final project has been a rewarding experience that not only met the goals of designing a 5-stage pipelined RISC-V CPU core but also explored advanced features and optimizations. The skills acquired through this project will undoubtedly contribute to our future endeavors in the field of digital hardware design and computer architecture.