CSN221 PROJECT

IMPLEMENTING A RISC-V SIMULATOR

Under guidance of

Prof. Debiprasanna Sahoo

CREATED BY:

Shourya Goel

22114090

Contact: +91 9971 749 585

Email: shourya_g@cs.iitr.ac.in

PROJECT REPOSITORY:

https://github.com/Sh0g0-1758/RISCV-Instruction-simulator

PROJECT DETAILS:

This project consisted of four phases which included:

- Implementation of a generalized assembler
- Designing a single cycle processor
- Working implementation of a 5-stage pipeline with account for all kinds of data hazards
- Building a cache and integrating it with our core simulator

PHASE I:

This phase consisted of three parts. The first part consisted of writing RISC-V codes for factorial, GCD and LCM and prime numbers, while the second part involved finding machine encodings for these programs. This was a great exercise for me, and it helped me familiarize myself with the RISC-V ISA.

The final part involved implementing the assembler. As I was going through the binary encodings for various instructions, I observed certain patterns in them and that's when I realized that I could make a generalized assembler by creating tokens for the user input code.

To create tokens, initially I was using C++ regex library, but it was too slow. On further research, I found out that manipulation with strings makes the program lag a lot, hence I decided to use the boost C++ library which exponentially increased the execution time of my assembler. If the RISC-V ISA comes up with more instructions, one simply needs to add them to the instructions.txt file in the required format and the assembler will then create the correct machine encodings for each of them.

I tested the assembler on several test cases, and it generated the correct machine encodings for each of them.

PHASE II:

The second phase of this project involved implementing a single cycle processor which I implemented without much hiccups. I assumed an ideal memory hierarchy and created separate C++ files for memory and registers that the simulator uses. The simulator takes in the machine encodings from the assembler and iterates through them with a for loop checking for ALU Operations, Memory Read-Write, Branch Instructions etc.

PHASE III:

Implementing a 5-Stage Pipeline accounting for data hazards was a rather tough job. I initially thought of implementing it using multi-threading, but I soon realized it would be messy to synchronize the parallel execution of multiple threads. Nevertheless, I have given the half-baked implementation of it using threads in the code.

As instructed by professor, I proceeded to create the illusion of pipeline by executing the different stages of it in the reverse order. To do this I first modularized my code and created execution registers for each stage of the pipeline. After I had separate classes and functions for the register and pipeline stages, I proceeded with removing the data hazards present in the pipeline.

Firstly, to simulate operand forwarding I created a separate class for it which was updated after the third and fourth stages of pipeline. I could not remove the stall logic completely because of the store instructions which get data in the fifth stage only. Further, I used Harvard architecture thus removing memory hazards. I then proceeded with incorporating branch instructions in my code which I created by creating a program counter valid flag which if set true, flushes the first two stages of the pipeline.

After a few hours of debugging the code, I created a fully functional 5-Stage Pipeline accounting for data hazards.

PHASE IV:

In phase IV, I created a write-back allocate cache with LRU replacement policy. Further, I decided to implement a set associative cache. To give illusion of real cache simulator, I have included a stall of one second on each main memory request. I have also implemented a memory hierarchy in the simulator with the main memory being $64X64$ in size and the cache being a 4-Set Two-Way cache. The address is divided as tag index and block offset so that in the main memory it will correspond to a data location of $(16 * index + tag)^{th}$ row and $(block\ offset)^{th}$ column. Further, I have created a separate `struct` for cache blocks and implemented functions like CPU-Read request, CPU-Write request, allocation in the cache, eviction from the cache taking account for modified cache block and updating of LRU Order.

Finally, I tested my cache simulator on six batches of test cases, and it gave correct result in each of them.

Lastly, I integrated my cache simulator with my core pipelined simulator and tested it on several test cases which further gave me correct results on each of them.

FUTURE SCOPE:

It was a rather fun experience for me creating this project and I hope to add a GUI to it which is under active development. It will likely be made using IMGUI or ElectronJS. I also hope to add further instructions and make the simulator more efficient, both in terms of memory and runtime.