High-level RISCV Simulator

1st Anmol Goyal
Electronics and Communication Department
Indraprastha Institute of Information Technology
Delhi, India
anmol19147@iiitd.ac.in

2nd Anurag Yadav
Electronics and Communication Department
Indraprastha Institute of Information Technology
Delhi, India
anurag19150@iiitd.ac.in

3rd Yash Aggarwal
Electronics and Communication Department
Indraprastha Institute of Information Technology
Delhi, India
yash19219@iiitd.ac.in

***Abstract*—We have designed a simulator that simulates a subset of RISCV(RV32) ISA in C++. The main components of this simulator are Main Memory, registers, instruction register, program counter(PC). We took assembly code as input in the form of string and converted it into binary code Instruction of 32-bit and We have designed a simulator that simulates a subset of RISCV(RV32) ISA in C++.**

This simulator has 5 stages:

- **Instruction Fetch: New instructions to be executed are loaded from the memory into the instruction register**
- **Decoding: The fetched instructions are decoded i.e. the simulator identifies the type of the instruction and the values of the required registers are also read.**
- **Execution: Different operations are performed as defined in the instruction**
- **Memory: If there is a need of accessing the memory, then this stage helps in loading and storing the values in the main memory.**
- **Writeback: In this stage, the values are written in the registers**

The workflow of our simulator is as follows:
First, it loads the input binary(compiled by our assembler) in the main memory, and then it starts executing the code at the address 0. Each instruction is then passed from the above mentioned 5 stages of our simulator.

After this we have added cache to our program . Our cache has two different writing policies like write back and write through , three different replacement policies Least recently used , First in First out , Randomly and we are checking for hit and miss and performing all the cache operation.

***Index Terms*—LRU, FIFO, Write-back, RISCV, Assembler, Cache, Write Through**

I. Introduction

We have created a RISC V Simulator in High-Level CPP language. We have created an assembler and a Simulator along with Main memory and Cache Memory. The Assembler converts the Assembly language code into its corresponding binary. The simulator then reads the generated Binary file and then executes the instructions and stores the result in the output file. Output consists of Time taken, Program Counter value, Cache Memory, Values Stored in all registers and Memory location. We have different functions for all the Instructions by categorizing them in different instruction formats and then sub categorizing them.

II. Assembler

We are given a set of instructions as input but as the computer understands binary we must convert these input into binary commands, The job of the assembler is to convert instructions in assembly language to binary, so the following are the implementation details of our assembler. A binary instruction has three major parts:

1) Tag
2) Index
3) offset

And for every instruction, we have some different combinations used to convert them to binary we can categorize them into several categories:

1) **R format :** these are the instructions which have 3 register inputs some of its examples would be add, xor, mul basically all logical and arithmetic operations fall under this.
2) **I format:** These are instructions with immediates and loads example would be addi, lw, jalr
3) **S format:** These are store instructions like sw.
4) **SB format :** All the branch instructions comes under this like beq , bge.
5) **U format:** Instructions with upper immediates like lui.
6) **UJ format :** Jump instructions like jal.

So in this assembler first we identify which type of instruction is this from that instruction we take out different details like which registers are being used, what is data and all required information after identifying this we need to follow certains rules according to Risc v standard notations and for this we have followed the rules from the link: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html. After running this to all instruction every thing would have been converted to binary and we save all the binary instruction in binary.txt file.

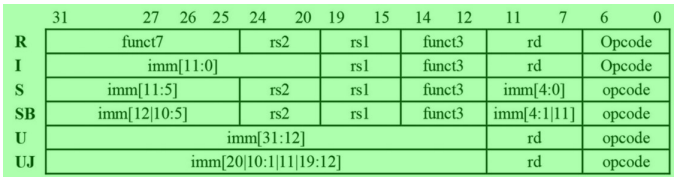

Fig. 1. RISC-V Instruction Formats ,Instructor: Steven Ho

III. Cache Memory

Cache Memory is a special very high-speed memory. It is used to speed up and synchronize with high-speed CPUs. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.

We have implemented various types of L1 cache divide on the basis of Associativity, Write policy and Replacement policy.

1) **On the basis of Associativity:**
   - Direct Mapped
   - Fully Associative
   - K -way set associative
2) **Index On the basis of Write policy :**
   - Write Back
   - Write Through
3) **On the basis of Replacement policy :**
   - Least Recently Used (LRU)
   - First in First Out (FIFO)
   - Random

The cache parameter are :

1) Size
2) Miss penalty
3) Hit time
4) Block Size
5) Associativity (Direct, Fully Associativity, K -way set associativity)
6) Write policy (Write back(WB),write-through(WT)).
7) Replacement Policy (LRU, RANDOM, FIFO)

We have made different arrays to store data and tag for each entry in cache. We have made various functions to achieve the functionality of cache. The functions that we made are: //

A. Hit-Miss

In this function we check whether a particular address is present in the cache memory or not. If the address is present in the cache then this function will return the value stored in cache, along with the location of the cache line in which data is present. If data is not present then it will return the value to be stored in cache, along with the location of the cache line in which data is to be stored.

B. isFull

This function will check whether the cache is full or not(i.e. the location whether we have to write the data is available to write or not). If there is no valid location where we can write the data then it will return -1, else it will return the index of cache where we can write the data.

C. Replace

It will check which replacement policy we are using and will call the appropriate function. We have made 3 more functions for FIFO, LRU and Random which we call from this function.

D. Read

It will call the Hit-Miss function and check whether a particular data is present in cache, if data is not present then it will check if cache is full or not. If the cache is not full along with a miss then it will simply write from main memory otherwise it will call replace function. If hit-miss returns a hit then it will simply return the data present.

E. Write

It will call the Hit-Miss function and check whether a particular data is present in cache, if data is not present then it will check if cache is full or not. If the cache is not full along with a miss then it will simply write, otherwise it will call replace function. If hit-miss returns a hit then it will simply write the data. We have also handled both replacement policies i.e. Write Back and Write Through. For write back we have maintained a dirty bit which becomes high when the data is updated in cache at that particular location and when we call replace policy we check this dirty bit and if the dirty bit is high then we write at memory location . In write through policy we simply write at both cache and main memory at the same time.

F. Dump

It will print the cache with tag array and data array.

G. FIFO

This function is for FIFO replacement policy. In this we have chosen a set and accordingly mainted a counter and replaced at location equal to counter mod set size.

H. LRU

This function is for LRU replacement policy. In this we have chosen a set and accordingly mainted a priority queue with the least recently used location having maximum priority and most recently used having least priority. When this function is called we replace at the cache line with most priority and update the priority queue accordingly.

I. Random

This function is for Random replacement policy. In this we have chosen a set and accordingly write at a random cache line in the set.

IV. Main Memory

We have created a string array to simulate memory. We have made a function to set the size of memory and to initialize it to zero. We have also made a function to set the memory access time and a dump function to print the memory.

V. Simulator

The binary file returned by assembler is then used by the simulator to run the assembly code that we have written. The simulator reads the binary coded instructions and interprets them. We have five stages in simulator :

- **Instruction Fetch:** New instructions to be executed are loaded from the memory into the instruction register
- **Decoding:** The fetched instructions are decoded i.e. the simulator identifies the type of the instruction and the values of the required registers are also read.
- **Execution:** Different operations are performed as defined in the instruction
- **Memory:** If there is a need of accessing the memory, then this stage helps in loading and storing the values in the main memory.
- **Writeback:** In this stage, the values are written in the registers.

We have made various functions to achieve the functionality of the Simulator. The functions that we made are:

A. Fetch:

It reads the instruction from memory and updates the Program Counter.

B. Decode:

It first characterizes all the instructions as **R, I, S, SB, U or UJ** and then finds the values of all the registers, address, immediates, offset etc. according to the category of the instruction. It then calls the respective function according to the respective instruction format.

C. Execute:

This function takes parameters from the decode function and then executes the instructions by calling different functions(such as AND, ADDI, BEQ, LW, SW etc.) according to the instruction by passing the required parameters such as register addresses, immedates, offsets etc.

D. Memory:

This function reads and/or writes in the memory (Cache and/or Main Memory) by taking the memory address and/or data as parameters.

E. Write Back:

This function updates the registers according to the results obtained by performing respective operations according to instructions.

F. Reg-Dump:

This function basically prints all the registers along with data stored in them.

VI. Result

In the Fig. 2 we can can clearly see that by increasing cache size and associativity the miss rate decreases.

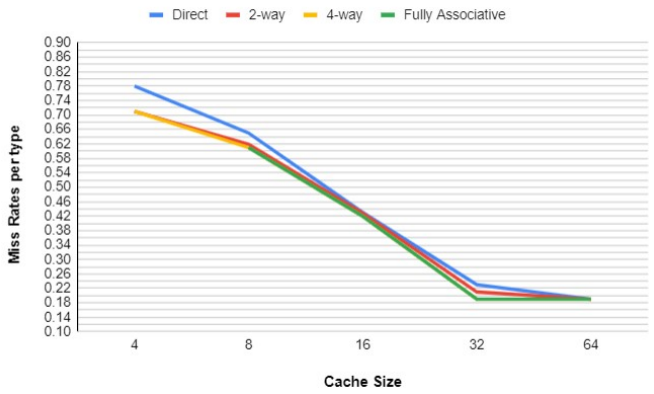

Fig. 2. Cache Type vs Miss Rate plot

We have executed a particular set of instructions with cache and without cache , in without Cache implementation total number of cycles we found to **2178 cycles** and in implementation with cache the number of cycles reduced to **520 cycles**, which is about **four times better performance**.

VII. Conclusion

From the above graph and observations, we can clearly see that the cache miss rates can be improved by changing the Cache sizes and the associativity of the Caches. We can see that the miss rates are improved if we take larger caches. This can be explained as by taking larger caches the capacity misses reduces. Capacity misses occur when the cache cannot contain all the blocks needed during the execution of a program. Also, by looking at the graph we can clearly see that the miss rates are improved if we increase the associativity of the caches. This can be explained as by increasing the associativity, the number of conflict misses decreases. Conflict misses are caused when several addresses map to the same set and evict blocks that are still needed.
By implementing a system with both cache and main memory we can clearly see multi-fold improvement in performance of our simulator as compare to a simulator implemented using only main memory.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our advisor, Dr. Sujay Deb, for providing us with this opportunity and for all his guidance throughout this project. We would like to thank all the TA's who were very supportive, helped in clearing the concept and was a source of learning.

REFERENCES

[1] Course material of ECE-511 (Computer Architecture) conducted at IIIT Delhi during Monsoon 2021.

[2] "RV32I, RV64I Instructions — riscv-isa-pages documentation", Msyksphinz-self.github.io. [Online]. Available: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html. [Accessed: 12- Dec- 2021].

[3] "Specifications - RISC-V International", RISC-V International. [Online]. Available: https://riscv.org/technical/specifications/. [Accessed: 12- Dec- 2021].

[4] S. Ho, "RISC-V Instruction Formats", Inst.eecs.berkeley.edu. [Online]. Available: https://inst.eecs.berkeley.edu/ cs61c/resources/su18$_lec/Lecture7.pdf$.[$Accessed$: $12 - Dec - 2021$].