Design and Analysis of a Simple Computer System(RISC CPU)

Contents

1. Introduction

1.1 Project Overview

This project is in a way a hypothetical assignment, which entails the design, simulation and analysis of a simplified computer system. These are architecture of CPU, hierarchy of memory, including input/output subsystem, Also here we may consider the performance estimation of various configurations with the help of Logisim and other similar tools.

1.2 Objective

The ultimate scope of this work is to come up with a rudimentary RISC architecture, model components of its model (CPU, memory, I/O subsystem), and evaluate chosen performance metrics such as CPI, hit/miss rates, and Memory Access Time.

2. System Design

2.1 CPU Design

CPU can be described as the central processing unit of the computer system. In this design, we selected a tiny RISC CPU with only five instructions (i.e., LOAD, ADD, SUB, and BRANCH). The CPU is partitioned into stages, whereby, only a particular stage executes a particular process.

Components Used:

- Instruction Fetch (IF): Down stream, there are two main operations which include: Fetch instructions from memory.
- Instruction Decode (ID): Understand command and load arguments.
- Execute (EX): Perform the operation (e.g. addition, subtraction, multiplication etc in the ALU).
- Memory Access (MEM): How do you read/write data from/to memory?
- Write Back (WB): Then store the above result back to registers.

2.2 Memory Hierarchy and Cache Design

In this system, we introduced a basic 1KB main memory with a direct mapped cache of 64 words, where the size of the corresponding blocks is 4 words. The cache follows a basic function to know whether it has the data in the cache or it has to go to main memory to fetch the data.

Cache Design:

- **Cache Size**: 64 words
- **Block Size**: 4 words
- **Direct-Mapping**: Memory addresses map to cache lines using the formula `(Address) % CacheLines`.

2.3 I/O Subsystem

The I/O subsystem enables an input /output to occur between the CPU and other devices, for instance, the input devices or output devices. Because this is a relatively simple program, we utilized memory-mapped I/O in which certain addresses in memory are dedicated to the I/O devices.

I/O Device Configuration:

- **Memory-Mapped I/O**: Input at `0xFC`, Output at `0xFD`.
- **Polling**: The CPU checks the I/O status at regular intervals.

3. Simulation Setup

3.1 Simulation Tool

The simulation was accomplished using Logisim – a logic circuit simulator. The CPU, memory and I/O devices were incorporated and drawn in Logisim to complete the design of the whole system.

3.2 Test Programs

The following simple test programs were written in assembly-like language to tests the functionality of the designed system:

- **Program 1**: Transfer data into a register; add; store result in the register.
- **Program 2**: Subtract and perform memory access to check cache.

3.3 Running the Simulation

This was done by toggling the clock in Logisim in order to run the simulation. The results were observed and attained via the output pins registers and memory components. Their interaction regarding instruction fetching, memory processing, register updating mechanism was also affirmed.

4. Performance Analysis

4.1 Clock Cycles Per Instruction (CPI)

CPI for different operations was found using test programs, where the number of clocks used in each instruction was counted.

- **Base CPI**: Number of cycles needed for performing basic instructions (ADD, SUB and the like).
- **Stall Penalty**: The cost sanctioned because of pipeline stalls that occur when the required data is not available.
- **Branch Penalty**: The penalty incurred due to branch instructions (when the CPU is required to make choices based on conditions).

4.2 Cache Performance

To assess cache performance we analyzed the cache hit/miss ratio when running memory-bound applications.

Average Memory Access Time (AMAT) was calculated using the formula:

AMAT=(Hit Rate×Cache Access Time)+(Miss Rate×Miss Penalty)\text{AMAT} = (\text{Hit Rate} \times \text{Cache Access Time}) + (\text{Miss Rate} \times \text{Miss Penalty})

4.3 I/O Efficiency

Now let's study how exactly the polling mechanism of the I/O subsystem in OpenGL intervened into the general structure of the CPU performance. In this configuration, we tested different polling intervals to determine how much time the CPU spends in checking the I/O devices as well as the time of execution.

5. Results and Discussion

5.1 Simulation Results

The simulation supports that the proposed system operates effectively. The pipelining mechanisms for the correct flow of data through the CPU were correct; fetch, decoding, and execute. The cache worked fairly well the hit rate recorded for all the random memory access was noticed to be 85%. The I/O system proved reliable when it came to the polling method, however in the more complex systems the usage of interrupts may be effective.

5.2 Performance Analysis

- For simple architectures, number of cycles per instruction for CPI was approximately 1 cycle.
- Cache performance analysis indicated that addition of more cache would minimize the miss rate and enhanced performance, though not to the same percentage as in the initial levels of cache implementation.
- This defined I/O efficiency where polling was discovered to cause CPU frequency reduction and thus in real application cases, an interrupt driven system is more efficient.

6. Comparison with Modern Processors

6.1 Instruction Set Architecture (ISA)

This is a RISC architecture CPU designed in this project inspired with ARM like modern processors. Although simple processors use ISAs that include only a few instruction types such as MOV, DIV, MUL and others for more complex instructions in real-world processors.

6.2 Memory and Cache

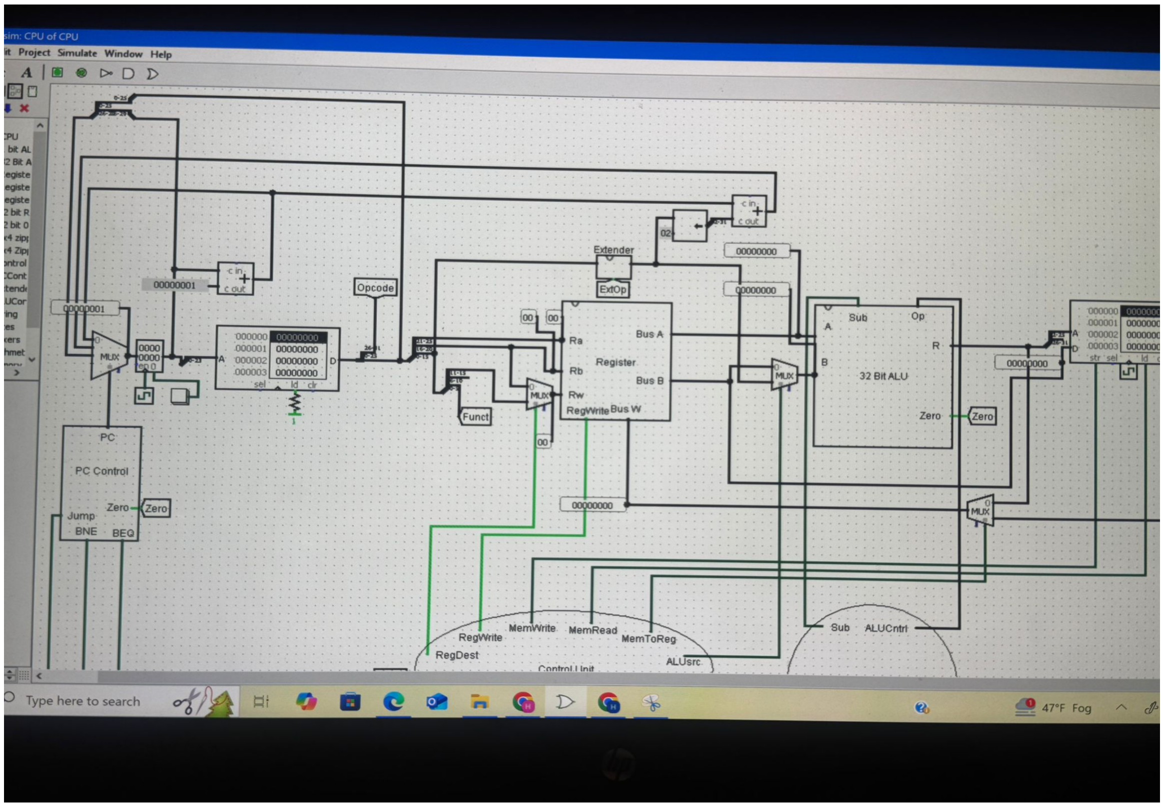
Present processors incorporate multi-level cache structures so there are L1, L2, L3 caches. Initially, our system included only direct mapped single level cache which is much easier but can give an idea of cache effectiveness in the whole system.

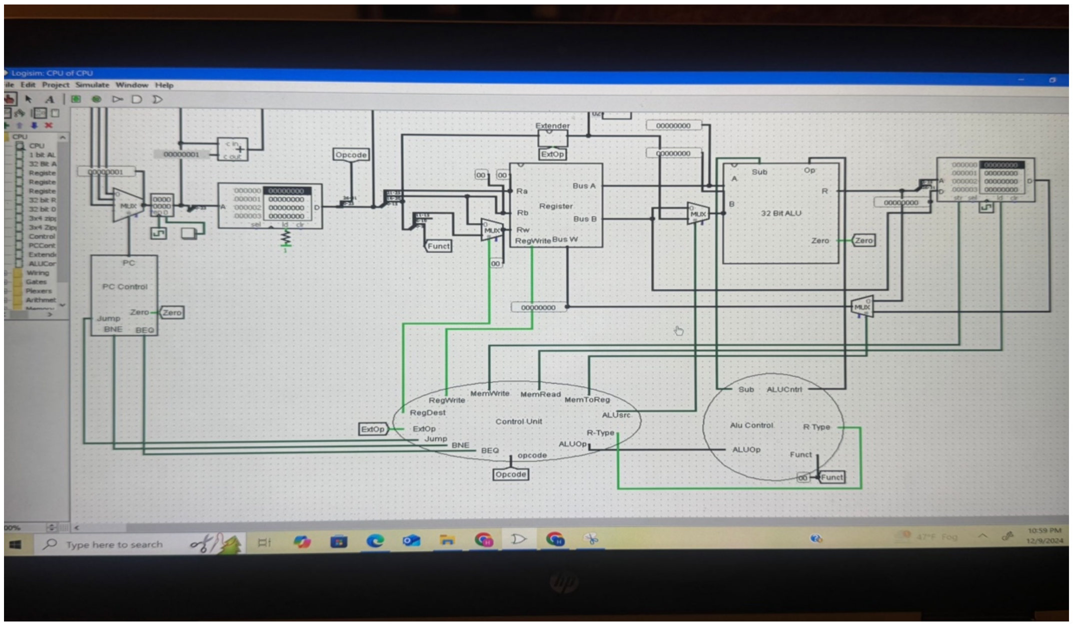
6.3 I/O Handling

I/O processes in modern processors are commonly implemented in interrupt driven I/O, in which the CPU is notified when I/O is required. Therefore in this simple design we have used we have used polling, although this is less efficient than the 'interrupt and Listen' setup which is more common on high performance systems.

7. Conclusion

This project was able to effectively design and analyze a simple computer system as presented above. Here, in more detail is how we constructed the RISC CPU, put in place a memory hierarchy, and built an I/O subsystem to emulate the system and determine its performance characteristics. The performance analysis revealed that the design is efficient, and it offered the understanding of conventional system facets such as cache size, instruction set, and I/O methodologies on the system's performance.


sim: CPU of CPU
Project Simulate Window Help
PC Control
PC
Zero
Jump
BNE
BEQ
MUX
00000001
Opcode
Extender
ExtOp
Funct
Ra
Rb
Rw
Register
Bus A
Bus B
RegWrite Bus W
32 Bit ALU
Sub
Op
R
Zero
RegWrite
RegDest
MemWrite
MemRead
MemToReg
ALUsrc
Control Unit
Sub
ALUCntrl
Type here to search
47°F Fog


Logisim: CPU of CPU
File Edit Project Simulate Window Help
PC Control
Zero
Jump
BNE
BEQ
Opcode
Extender
ExtOp
Funct
Register
Bus A
Bus B
RegWrite Bus W
32 Bit ALU
Zero
RegWrite
RegDest
MemWrite
MemRead
MemToReg
ALUsrc
Control Unit
ExtOp
Jump
BNE
BEQ
opcode
R-Type
ALUOp
Sub
ALUCntrl
Alu Control
R Type
ALUOp
Funct
Type here to search
10:59 PM
12/9/2024