INDIAN INSTITUTE OF TECHNOLOGY DHARWAD

# Cycle-Accurate Leon3 Simulator

Unnati Athwani*
December 11, 2020

**Abstract**

LEON is a radiation-tolerant, high-performance processor initially designed to be used in European space projects. Presently, it is extensively used in satellites, including ISRO. A C++ model of the open-source processor is required to be made for testing purposes. The project aims to implement the memory and cache parts of the same.

## 1 INTRODUCTION

LEON is an open-source, 32-bit processor that employs the SPARC V8 Instruction Set Architecture. It was originally designed by the European Space Research and Technology Centre (ESTEC), part of the European Space Agency (ESA). It is described in synthesizable VHDL. It is now widely utilised for spacecraft applications.

For testing purposes, a C++ simulator of the processor needs to be built. This project implements the memory and cache models of the processor. After the proper implementation of the system, different configurations of cache were tested against a set of selected benchmark programs (SPEC 2017). The performance of the models was compared and analysed, so that the best performing model can be selected.

*Email: 170010006@iitdh.ac.in

# 2 MOTIVATION

A simulator as we know is a model to represent a real system. Rather than using the real hardware itself, simulating the model fulfills many purposes. First of all, it serves the purpose of the analysis of models for research purposes. Secondly, using or building the hardware for every need will incur huge costs to the organization. Simulation turns out to be a cost-effective way to evaluate the hardware designs. More importantly, it is used for software testing and debugging, which could be cost-cutting, less time-consuming, and the organization may not have enough hardware in the first place to provide for rigorous testing of software. Hence the need for a simulator.

# 3 BACKGROUND

## 3.1 MEMORY MODEL

In computing, memory refers to a device that is used to store information for immediate use in a computer or related computer hardware device. It is the collection of locations accessed by the load/store instructions.

The simulated Memory Model is byte addressable and 4 GB in size.

### 3.1.1 LOAD/STORE

Among all the instructions, Load/store instructions are the only ones that access memory. Integer load and store instructions support byte, half-word (16-bit), word (32-bit), and double-word (64-bit) accesses.

### 3.1.2 ADDRESSING CONVENTIONS

SPARC is a “big-endian” architecture: the address of a double-word, word, or half-word is the address of its most significant byte. Increasing the address generally means decreasing the significance of the unit being accessed.

## 3.2 CACHE MODEL

A cache is a special storage space used to temporarily hold instructions and/or data that the CPU is likely to reuse.

The SPARC v8 provides separate instruction and data cache (Harvard architecture). The caches are configurable, typically 1-4 ways associative, with 1-256 KB/way. The replacement policy used is LRU (Least Recently Used), in which as clear from the name, discards the least recently used page, in case the cache is full.[1]

# 4 WORK DONE

This section discusses the work done on the project throughout the semester. The work can be roughly divided into parts which are explained in subsequent sections.

## 4.1 A SIMPLE MODEL

A simple byte-addressable model of memory was created. Initially, the size of the simulated memory was kept small ($2^{14}$ bytes instead of $2^{32}$ bytes), so that it fits in the system memory. The functions of load and store were added. The memory and register addresses were given to the program through an external file, which was in same format as the benchmark files used for the experiments. The functions for byte, half-word and double-word accesses were also added.

### 4.1.1 BENCHMARK FILES

The benchmark files are SPEC 2017, which is a standardized set of performance benchmarks for computers, and were provided by the instructor. A typical benchmark file format is shown below:
<0(load)/1(store)>,<program counter>,<number of bytes read/written>,<memory address>

### 4.1.2 MAIN PROGRAM

The pseudo-code of the main program is given below:

```
1  while true
2      if end of file is reached
3          break
4      Read line from the file
5      if instruction is load
6          PerformLoad(line.memory_address, line.register)
7      if instruction is store
8          PerformStore(line.memory_address, line.register)
```

### 4.1.3 ISSUE

The size becomes an issue here. $2^{14}$ bytes is a small size which can be fit in the system memory. But the size required was $2^{32}$ bytes (4 GB).

## 4.2 EXPANDING MEMORY SIZE

The size of the simulated memory has to be 4 GB ($2^{32}$ bytes). Such a large size is not practical due to storage constraints in the system memory. Hence, the simulated memory was represented as a binary file of size 4 GB on the disk. Any load/store request would then just require a file seek of the required byte.

#### 4.2.1 Issue

Every single file seek takes a significant amount of time. This would greatly effect the running time of the simulator.

### 4.3 A Memory Model Accelerator

To address the issue mentioned above, an in-memory data structure was needed to be created, which would essentially be a subset of the complete simulated memory. This is in same lines with the concept of a data cache.

#### 4.3.1 Implementation

A simple model of fully associative cache with configurable cache and page sizes was implemented. To efficiently search an address, a separate array *tag* was maintained, which acts as an identifier for a group of data, storing the value of *address/(page size)*. A *dirty* array is maintained to keep track of the modified values in the cache. At last, a *last_used* array keeps track of the time the addresses were last used.

Any requested address was first looked for in the cache. If present, the request results in a hit. If a miss, the index of tag to be used for bringing the required page is found. If the cache is not full, the next unused index is used, else the replacement policy (LRU) is applied to find the same. If the index returned is used and has dirty bit set to true, the page is written to (memory) file. The dirty bit for the page is then set to false, since the page has been written. Then the present page in the cache is replaced with the page that contains the required address.

Now, the corresponding action is taken, i.e., the value of the required address is returned in case of a load, and the address in cache is changed to the value given in case of a store.

#### 4.3.2 Pseudocode

The following pseudocode implements the above idea:

FULLYASSOCIATIVECACHE(memory_address, value, instruction)

```
1  cache_index = addr / page_size
2  page_index = addr % page_size
3  hit = false
4  hit = CacheSearch(cache_index) // returns true if found
5  if !hit
6       evict_index = HandleCacheMiss() // find the index where page is to be brought
7       if evict index is dirty
8            WritePageToMem(evict_index)
9       WritePageToCache (evict_index, cache_index)
10 if instruction is load
11      update last used
12 else if instruction is store
13      cache[evict_index][page_index] = value
14      update last used
15      set dirty bit to true
```

### 4.4 EXPERIMENTS

Since the cache exists to bridge the speed gap, its performance measurement and metrics are important in designing and choosing various parameters like cache size, page size, replacement policy, etc. Cache performance depends on cache hits and cache misses, which are the factors that contribute to system performance. Hence, it becomes a mandatory exercise to analyse the performance in terms of the hit and miss rates, and the time taken.

The performance is measured mainly by these metrics-

1. **Hit rate:** Cache hits are the number of accesses to the cache that actually find the required address in the cache. The hit rate therefore, is the number of cache hits divided by the total number of memory requests.

2. **Miss rate:** Cache misses are the number of accesses to the cache that don't find the required address in the cache. Since this is complimentary to the hit rate, it doesn't need to be explicitly measured or examined.

3. **Time per memory request:** Measurement of time is crucial in measuring the performance of a cache. Although for a simulator, the time also depends on the configuration of the system used. For measuring the performance of the cache, time taken per instruction/memory request is calculated for each benchmark file.

### 4.5 HARDWARE SPECIFICATIONS

All the experiments were run in the CARES servers of IIT Dharwad, which have x86 architecture and are 64-bit systems. The processor used is Intel Xeon E5-2680 v3 processor. It has 12 CPU cores and the system cache size is 30720 KB.

# 5 RESULTS

We understand that the results of the entire set-up can be obtained when a related software code runs and executes fine on the hardware support established. The simulated cache was run for a total of 15 combinations of cache and page sizes, for all the 18 benchmark files provided. The metrics discussed above were measured. The results are described below.

# 6 CONCLUSION

## 6.1 VARIATION IN HIT RATE

The configurations with smaller page sizes perform better in terms of the hit rate. Increasing cache size (or increasing number of lines) does better for the hit rate, although there may be time concerns. The best hit rate is achieved with cache size of 1 MB (1024 KB) and page size of 4 KB. These trends are illustrated in figure 6.1

## 6.2 VARIATION IN TIME PER MEMORY REQUEST

The time taken shows the same trends for page size variations. The time increases with increasing page size. This can be fairly attributed to the larger miss penalty, the time required to fetch the missing page from main memory. Larger cache sizes, however, increase the time taken, unlike the hit rates. This is due to the time taken to search the address in the cache. The best-performing configuration in terms of time is 256 KB of cache and 4 KB of page size. Figure 6.2 outlines the idea.

Table 6.2 tabulates the data used for the graphs.

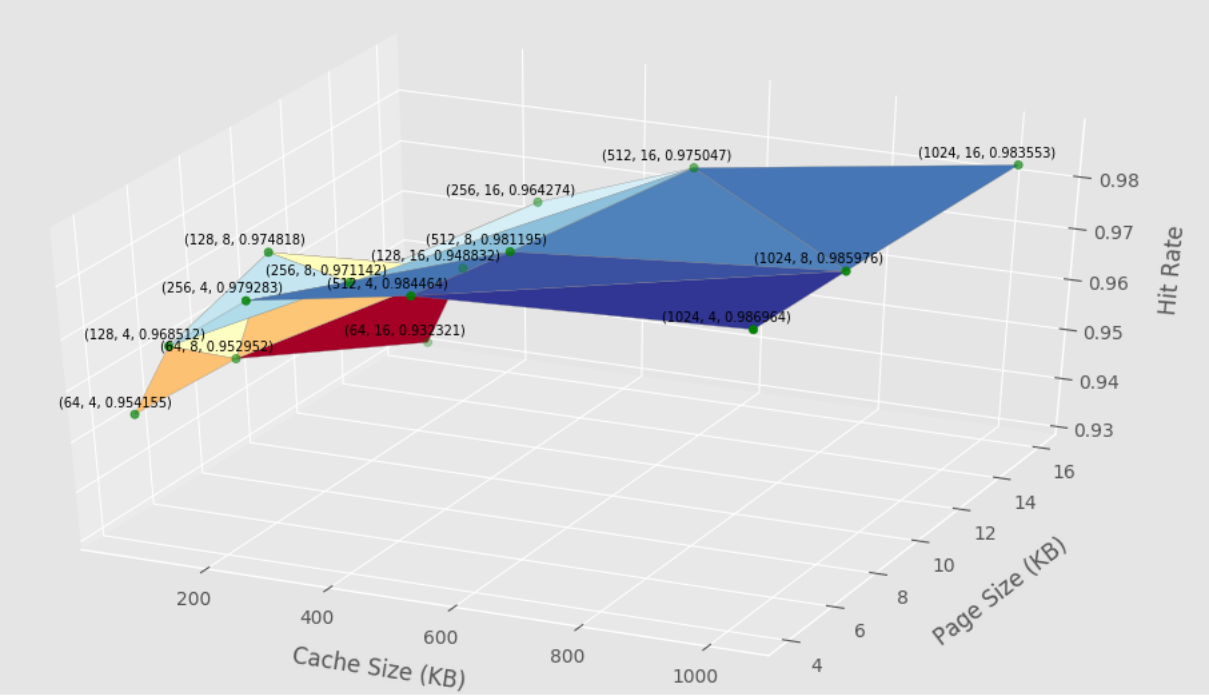


Figure 6.1: Cache size and page size vs. the hit rate

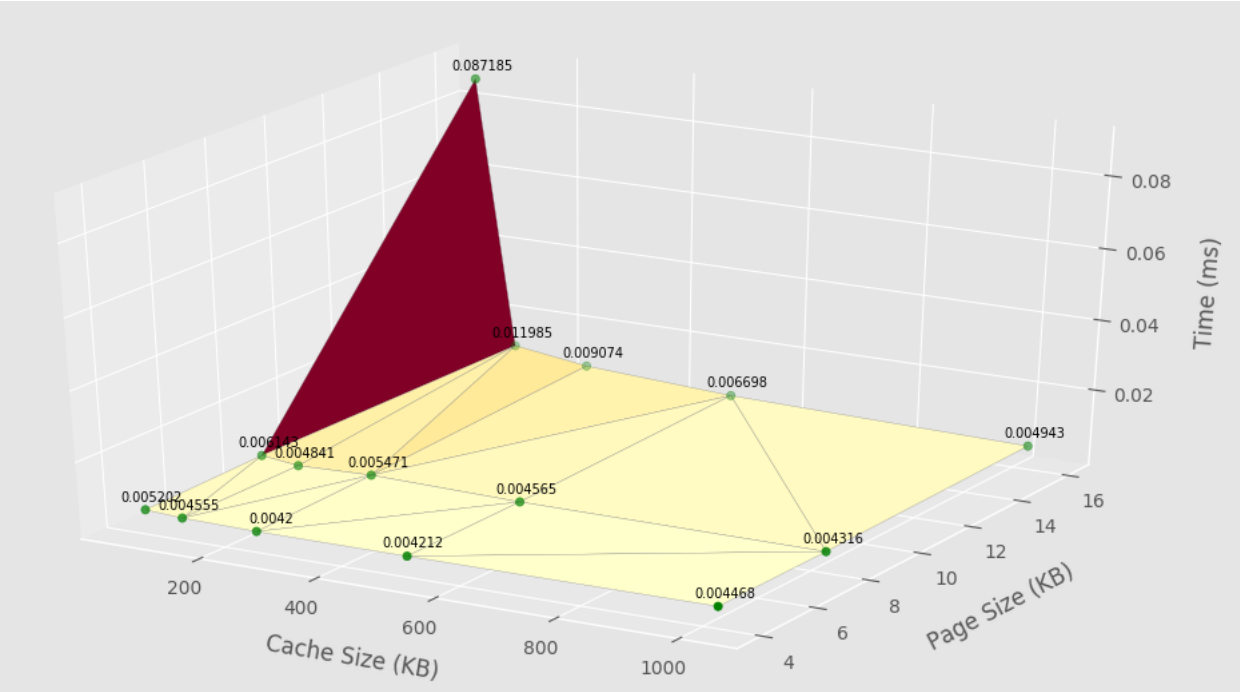


Figure 6.2: Cache size and page size vs. Average time per memory request

| Table 6.2: Variations in hit rate and time taken with cache and page sizes | | | |
|---|---|---|---|
| Cache Size (KB) | Page Size(KB) | Average Hit Rate | Average Time per Request (ms) |
| 64 | 4 | 0.954155 | 0.005202 |
| 64 | 8 | 0.952952 | 0.006143 |
| 64 | 16 | 0.932321 | 0.087185 |
| 128 | 4 | 0.968512 | 0.004555 |
| 128 | 8 | 0.974818 | 0.004841 |
| 128 | 16 | 0.948832 | 0.011985 |
| 256 | 4 | 0.979283 | 0.004200 |
| 256 | 8 | 0.971142 | 0.005471 |
| 256 | 16 | 0.964274 | 0.009074 |
| 512 | 4 | 0.984464 | 0.004212 |
| 512 | 8 | 0.981195 | 0.004565 |
| 512 | 16 | 0.975047 | 0.006698 |
| 1024 | 4 | 0.986964 | 0.004468 |
| 1024 | 8 | 0.985976 | 0.004316 |
| 1024 | 16 | 0.983553 | 0.004943 |

## 7 FUTURE SCOPE

- Set associative caches can be implemented to check for time in large cache sizes.
- Optimization of the simulated memory (which is currently a 4 GB file on disk) can be done as only one benchmark file per simulation is tested.

## REFERENCES


[1] https://www.gaisler.com/index.php/products/processors/leon3

[2] https://www.gaisler.com/products/grlib/grip.pdf

[3] https://www.gaisler.com/doc/sparcv8.pdf