

# Cache Design Lab Report

## Cache Design in SystemVerilog



**Ahmad Mukhtar**

**National University of Sciences and Technology (NUST)  
Chip Design Centre (NCDC), Islamabad, Pakistan**

October 13, 2024

# Objective

The objective of this lab is to design and evaluate different types of cache architectures using SystemVerilog. The three cache types designed and evaluated include Direct Mapped, Set Associative, and Fully Associative caches. These cache architectures were implemented based on specified parameters and tested using a verification framework.

# Tools Used

- Cadence Xcelium
- Python
- Linux

# Methodology

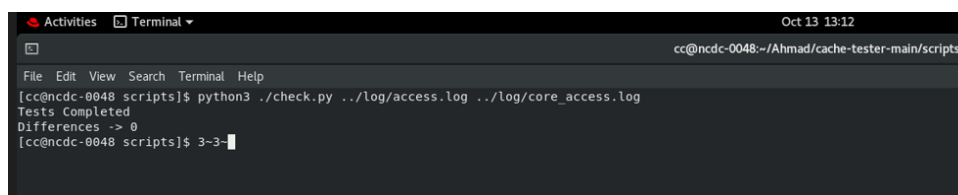
The design of cache architectures follows specific rules for data placement, access, and replacement. Each cache type was implemented with distinct strategies for these operations. Below is the detailed explanation of the methods used for designing each type of cache.

## Direct Mapped Cache

The Direct Mapped cache is the simplest type of cache design. In this architecture:

- **Block Addressing:** The memory is divided into fixed-size blocks, with each block mapped directly to a single cache line. In this design, we used a total of 256 blocks, with a 16-bit physical address divided into tag bits, block index, and word offset.
- **Indexing:** The cache uses an 8-bit block index (`address[11:4]`) to locate the specific cache line. The 4 most significant bits (`address[15:12]`) are used as the tag to identify whether the cache line contains the correct block from memory.
- **Hit/Miss Logic:** The cache checks if the tag stored in the cache line matches the incoming address tag. If the valid bit is set and the tags match, a hit occurs. Otherwise, it's a miss.
- **Data Output:** Upon a hit, the cache retrieves data from one of the four words in the block using the word offset (`address[3:2]`).

This design is efficient in terms of hardware but suffers from conflict misses, where multiple memory blocks map to the same cache line.



```
Oct 13 13:12
cc@ncdc-0048:~/Ahmad/cache-tester-main/scripts
File Edit View Search Terminal Help
[cc@ncdc-0048 scripts]$ python3 ./check.py ../log/access.log ../log/core_access.log
Tests Completed
Differences -> 0
[cc@ncdc-0048 scripts]$ 3~3
```

Figure 1: Evaluation of Direct Mapped Cache

## Set Associative Cache (2-way)

The Set Associative cache is a compromise between Direct Mapped and Fully Associative caches. For this design:

- **Set and Ways:** The cache is divided into sets, with each set containing two "ways" (blocks). A memory address can be stored in either of the two ways. In this design, we used 128 sets, each having 2 ways.
- **Indexing and Tagging:** The address is divided into a tag (`address[15:11]`), an index (`address[10:4]`), and a word offset. The index selects the set, and the tag is compared against both ways in that set.
- **Hit/Miss Logic:** A hit occurs if either of the two ways in the indexed set has a valid tag that matches the input address tag. If neither way matches, it results in a miss.
- **Replacement Policy:** While not explicitly mentioned in this design, typically a Least Recently Used (LRU) or random replacement policy is used to decide which way to evict on a miss.

This architecture reduces the conflict misses seen in the Direct Mapped cache by allowing two possible blocks per set.

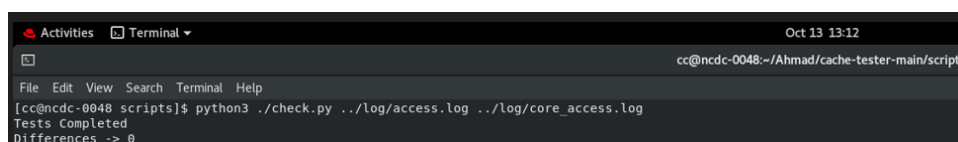


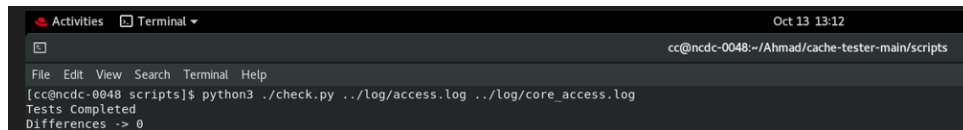
Figure 2: Evaluation of 2-way Set Associative Cache

## Fully Associative Cache

The Fully Associative cache provides the most flexibility in data placement. Here:

- **Data Placement:** Any memory block can be placed in any cache line, as there is no index restriction.
- **Tagging:** The entire cache is searched for a matching tag (`address[15:4]`). The search involves comparing the tag of each cache line with the incoming address.
- **Hit/Miss Logic:** A hit occurs when any cache line's tag matches the input tag. The valid bit must also be set. This design uses 256 lines, each of which is fully accessible to any memory block.
- **Data Output:** When a hit occurs, the cache retrieves the corresponding word using the word offset (`address[3:2]`).
- **Search Complexity:** The major drawback of this design is the complexity of searching all cache lines simultaneously, which requires significant hardware.

The Fully Associative cache minimizes conflict misses, as any block can go to any cache line. However, this comes at the cost of increased hardware complexity for searching all cache lines in parallel.

A terminal window titled 'Terminal' with a date and time of 'Oct 13 13:12'. The terminal shows the command prompt 'cc@ncdc-0048:~/Ahmad/cache-tester-main/scripts' and the execution of a Python script: 'python3 ./check.py ../log/access.log ../log/core\_access.log'. The output of the script is 'Tests Completed' and 'Differences -> 0'.

```
Oct 13 13:12
cc@ncdc-0048:~/Ahmad/cache-tester-main/scripts
File Edit View Search Terminal Help
[cc@ncdc-0048 scripts]$ python3 ./check.py ../log/access.log ../log/core_access.log
Tests Completed
Differences -> 0
```

Figure 3: Evaluation of Fully Associative Cache

# Conclusion

In this lab, we successfully designed and evaluated three cache architectures: Direct Mapped, Set Associative, and Fully Associative caches. The results showed that while Direct Mapped caches are hardware-efficient, they suffer from higher conflict misses. Set Associative caches offer a balance by reducing conflict misses, and Fully Associative caches provide the highest flexibility at the cost of increased hardware complexity. Each architecture has its trade-offs, and the best choice depends on the specific use case and system requirements.