Table of Contents

Background Study

A **victim cache** is a small, fully associative cache that sits between the primary cache—typically the Level-1 (L1) cache—and the next level in the memory hierarchy. It was introduced to address a specific performance problem in conventional cache designs, particularly **conflict misses** that occur frequently in direct-mapped and low-associativity caches. In early cache architectures, designers favored direct-mapped caches because of their simplicity, low latency, and minimal hardware cost. However, this design choice often led to situations where multiple memory blocks compete for the same cache line, causing repeated evictions even when sufficient total cache capacity exists. The victim cache concept emerged as a practical solution to mitigate this inefficiency without significantly increasing the complexity or access time of the L1 cache.

The fundamental idea behind a victim cache is to temporarily store cache blocks that are evicted from the primary cache. Instead of discarding an evicted block immediately to the lower-level cache or main memory, the system places it into the victim cache. Because the victim cache is **fully associative**, it can hold any evicted block regardless of its original index in the primary cache. When a cache miss occurs in the L1 cache, the system checks the victim cache before accessing the next memory level. If the requested block is found there—a situation known as a **victim cache hit**—the block is swapped back into the L1 cache, significantly reducing the miss penalty and improving overall performance.

Victim caches are particularly effective in workloads that exhibit **cache thrashing**, where two or more frequently accessed memory blocks map to the same cache line in a direct-mapped cache. In such scenarios, the L1 cache repeatedly evicts one block in favor of another, even though both are actively used. By capturing these evicted blocks, the victim cache acts as a buffer that absorbs this conflict behavior. This makes a small direct-mapped cache behave more like a higher-associativity cache, often achieving comparable performance gains with far less hardware overhead than increasing associativity directly in the L1 cache.

From an architectural perspective, the victim cache is usually small—often only 4 to 16 entries—to ensure that lookup latency remains low. Its fully associative nature requires comparators for all entries, but because of its limited size, the additional hardware cost and power consumption remain manageable. Importantly, victim cache access is typically performed **in parallel or in sequence with the L1 miss handling logic**, so it does not significantly increase the critical path for L1 hits. This design allows processors to preserve the fast access time of the L1 cache while still benefiting from reduced miss rates.

Historically, victim caches gained prominence through academic research and were later adopted in commercial microprocessors. Early studies demonstrated that even a very small victim cache could eliminate a large fraction of conflict misses in direct-mapped caches. As a result, victim caches became an attractive design choice in high-performance and embedded processors, where balancing speed, area, and power is crucial. They have also influenced later cache innovations, such as adaptive cache associativity and hybrid cache hierarchies, reinforcing their importance in the evolution of modern memory systems.

In summary, the background study of victim caches highlights their role as a targeted optimization in cache memory design. Rather than replacing traditional cache structures, victim caches complement them by addressing a specific weakness—conflict misses—using a relatively simple and efficient mechanism. This makes them a valuable architectural technique, especially in systems where low-latency access and efficient hardware utilization are primary design goals.

Problem Statement and Motivation

- Modern processors rely heavily on L1 caches for low-latency memory access. There are 2 extreme cache architectures; Direct Mapped Cache and Fully Associative Cache, each with its own pros and cons.
- **Direct-mapped caches** have **simple hardware** and **low hit time** but suffer from **high miss rates** mainly due to **conflict misses**. On the contrary the **Fully Associative caches** have **highest hit rates** but their **hardware is complex** and their **latency is high** due to the **large number of comparators** equal to the number of lines(Increasing associativity increases hardware complexity and access time).
- Victim cache provides a low-cost and latency effective solution to reduce conflict misses.
- This project aims to design a victim cache to improve cache efficiency.

Milestones

1. Design of a small parametrized Victim Cache architecture.
2. Implementation of fully-associative tag matching and replacement in the Victim.
3. Functionality Validation of the Victim Cache through Simulation and Test cases.
4. Design of the parametrized traditional L1 Direct Mapped Cache and functionality validation.
5. Modification of the Direct Mapped Cache (Extra Ports and Eviction to Victim Policy)
6. Integration of swap mechanism between L1 cache and Victim Cache on Victim Cache hit.
7. Functionality Validation of the Modified Direct Mapped Cache.
8. Interconnection of the two caches for a complete L1 block.
9. Final Validation of the complete design (Presynthesis Evaluation)
10. Performance Improvement Evaluation compared to a cache without a victim cache.
11. RTL Synthesis and Gate-Level Netlist Generation.
12. Physical Design and layout to GDSII Generation
    a. Floorplanning and Power Planning
    b. Standard Cells Placement
    c. Pre-CTS Optimizations and Clock Tree Synthesis (CTS)
    d. Post- CTS Optimizations for setup and hold constraints.
    e. Detailed Routing
    f. Filler Cells Placement
    g. Physical Verification (DRC and LVS check)
    h. GDSII Generation

RTL Implementation

VICTIM CACHE

The Victim Cache design consists of **two primary submodules**:

1. **Tag Store Module** – Responsible for storing and managing the tags of the cache blocks. It performs tag comparisons to identify hits and helps in determining which block should be replaced during cache evictions.
2. **Data Store Module** – Stores the actual data corresponding to each cache block. It handles read and write operations to ensure that the correct data is returned on cache hits and updated on write-backs.

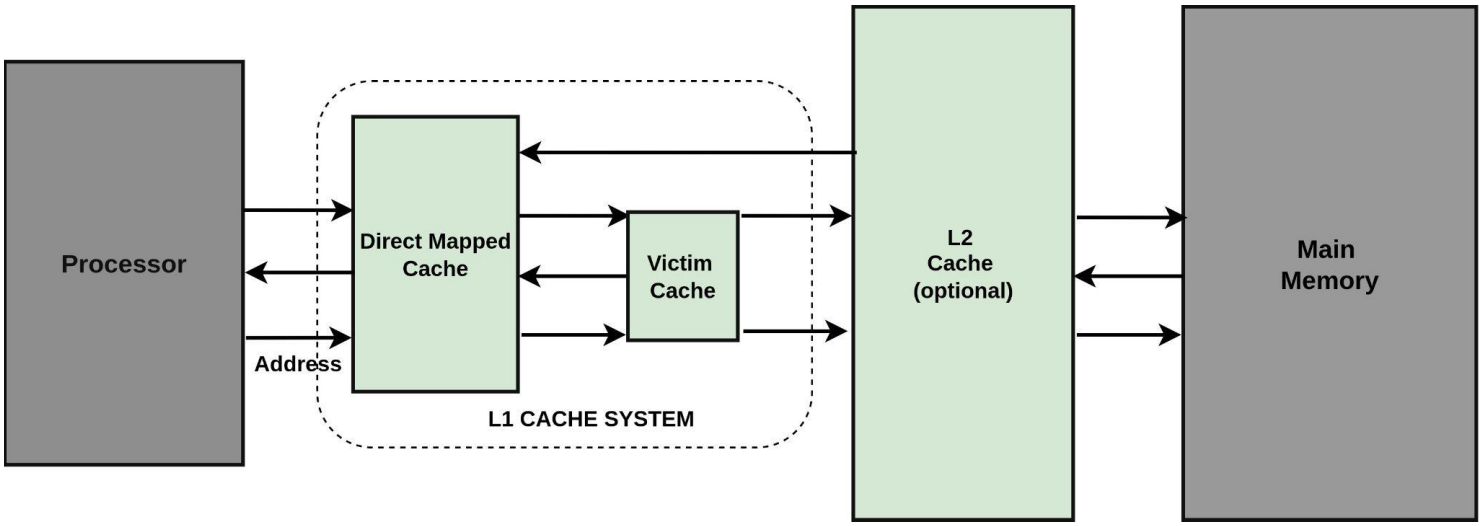
These submodules are **instantiated and coordinated by a FSM/Controller module**, which implements the control logic of the Victim Cache. The controller handles cache operations such as read, write, eviction, and swap operations with the L1 cache. It monitors signals from both the Tag Store and Data Store modules to manage cache hits, misses, and victim cache replacements effectively.

This modular approach ensures a clear separation between data storage, tag storage and control logic, making the design scalable, easier to verify, and reusable for different cache configurations.

The Victim Cache **does not read from higher level memory/cache (main memory/ l2 cache)** directly. It only writes to the higher level memory/cache and the **write policy** of the Victim cache is **"Write-Back".**

Position of the Victim Cache in the Architecture

The Victim Cache is positioned **between the L1 Direct Mapped cache and the next level in the memory hierarchy**, such as the L2 cache or main memory. All memory requests are first handled by the L1 cache to ensure low-latency access. When an L1 cache miss occurs, the Victim Cache is accessed before forwarding the request to higher-level memory. The Victim Cache temporarily stores blocks evicted from L1 and provides a mechanism to swap recently evicted data back into L1 on a victim cache hit. This placement allows the system to reduce conflict misses in the L1 cache without increasing its hit-time or critical access path.



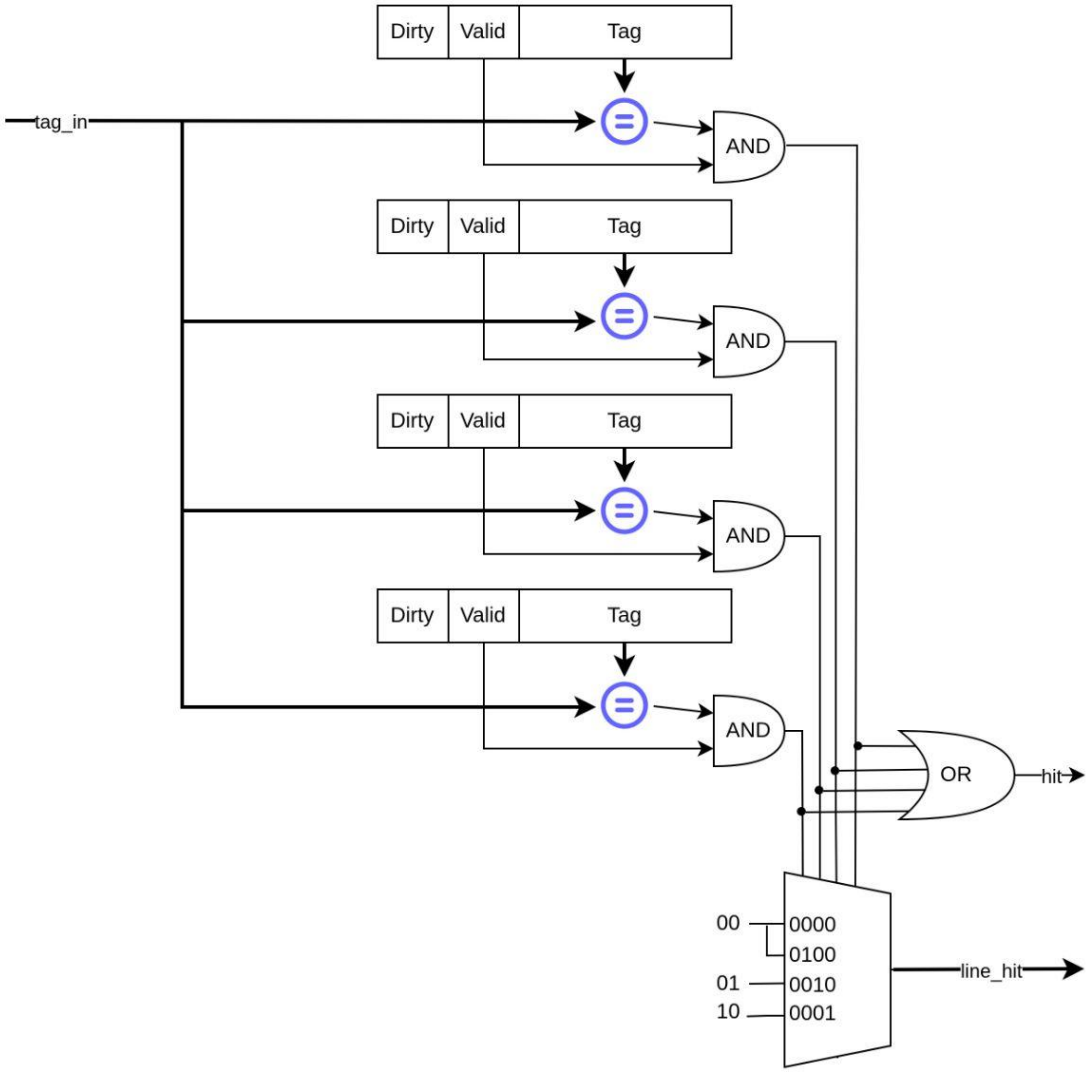
VICTIM CACHE'S POSITION

Tag Store Module

The **Tag Store Module** of the Victim Cache is responsible for storing the tags and associated metadata of cache lines evicted from the L1 cache. This module maintains essential information such as **valid bits** and **dirty bits**, which are directly taken from L1 along with the evicted cache lines and are **not modified inside the Victim Cache**.

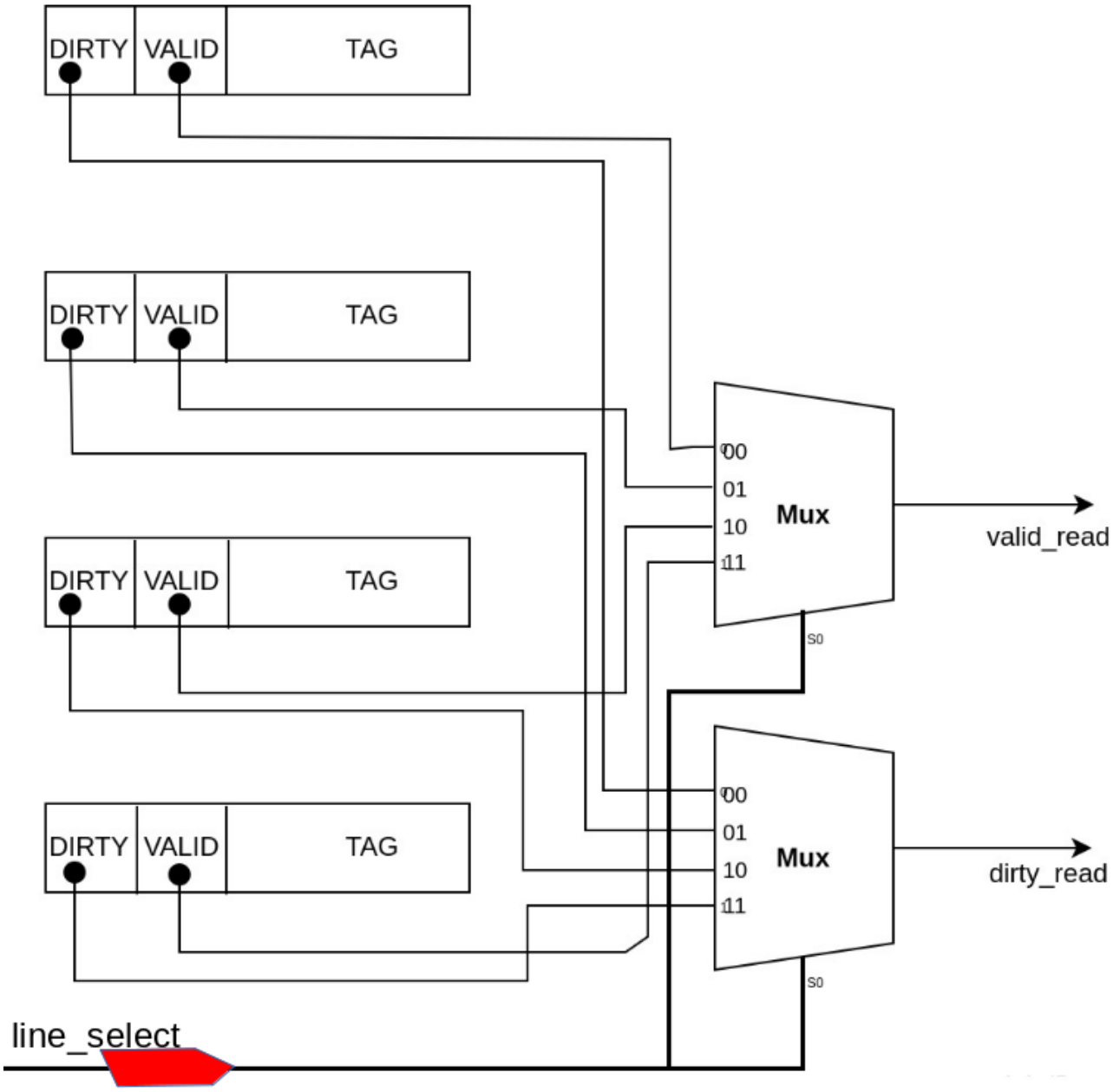
Operations

1. **Tag Lookup:**
   - Performed on every cache access to determine if the requested address resides in the Victim Cache.
   - The lookup is **combinational** and outputs a **hit/miss signal** along with the **index of the matching line** if a hit occurs.
   - This operation does **not output the metadata or tag values themselves**; it only indicates the presence or absence of the block.



TAG MODULE'S LOOKUP OPERATION

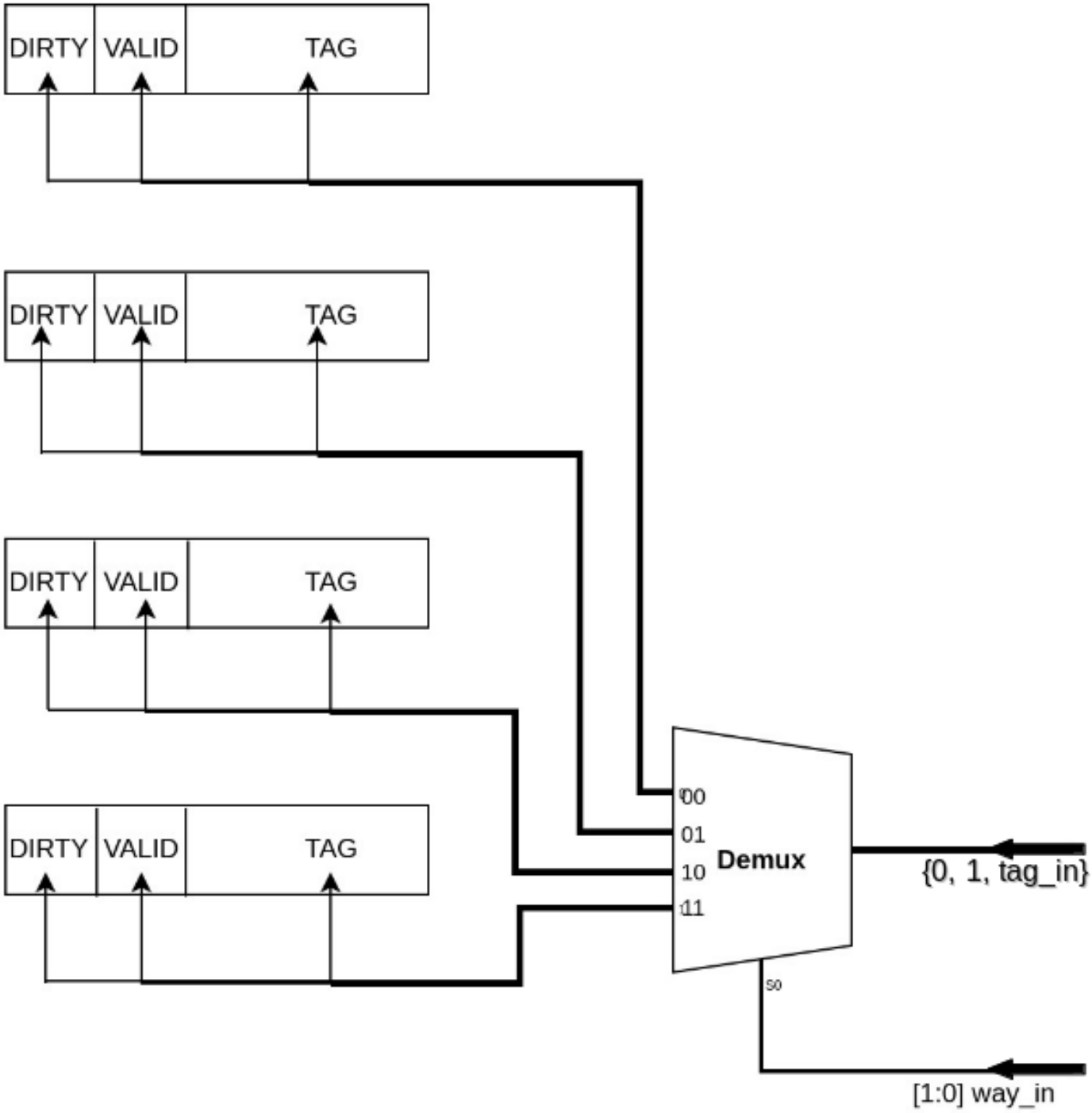
2. **Read Operation:**
    - Once the FSM/Controller identifies the line to access, the **read operation** is used to fetch the actual metadata of a specific line.
    - The read is **combinational** and outputs the **tag, dirty bit, and valid bit** of the specified line number.
    - This ensures that the Controller has all necessary information to manage swaps, replacements, or write-backs.



TAG MODULE'S READ OPERATION

3. **Tag Write:**

- Sequential operation synchronized with the system clock.
- Updates tag, valid bit, and dirty bit after **eviction from L1**, when a line is **replaced** in the Victim Cache, or when a victim line is accessed by the L1.



TAG MODULE'S WRITE OPERATION

Data Store Module

The **Data Store Module** of the Victim Cache is responsible for storing the actual data of cache lines evicted from the L1 cache. This module holds the complete content of each cache line and works in tandem with the Tag Store to ensure data consistency. On a cache access, when the Tag Store indicates a hit, the Data Store provides the corresponding cache line data to the Victim Cache controller for swapping back with the L1 Direct Mapped cache.
Data read operation in the Data Store is **combinational**, allowing the data to be accessed immediately on a hit, while **write operations** (on evictions or swaps) are **sequential**, synchronized with the system clock. The Data Store does not modify the data autonomously; all updates occur under the control of the FSM/Controller module, ensuring proper coordination with L1 cache operations and maintaining data integrity during replacements.

Victim Cache Controller (FSM)

The **Controller (FSM) Module** is the central unit that coordinates the operations of the Victim Cache. It orchestrates all interactions between the Tag Store and Data Store modules, as well as the interface with the L1 Direct Mapped cache. The FSM manages cache operations such as **reads, writes, evictions, and swaps**, ensuring that the correct data and tag information is transferred between the caches efficiently.

Operation on Cache Access

- The controller first monitors the **Tag Store** to determine if the requested line is present in the Victim Cache.
- If a **victim cache hit** occurs, the FSM initiates a **swap operation** between the Victim Cache and the L1 cache, coordinating sequential writes to the Data Store and Tag Store.
- If a **miss occurs**, the Victim Cache simply informs the L1 cache that the requested line is **not present** in the Victim Cache as well.

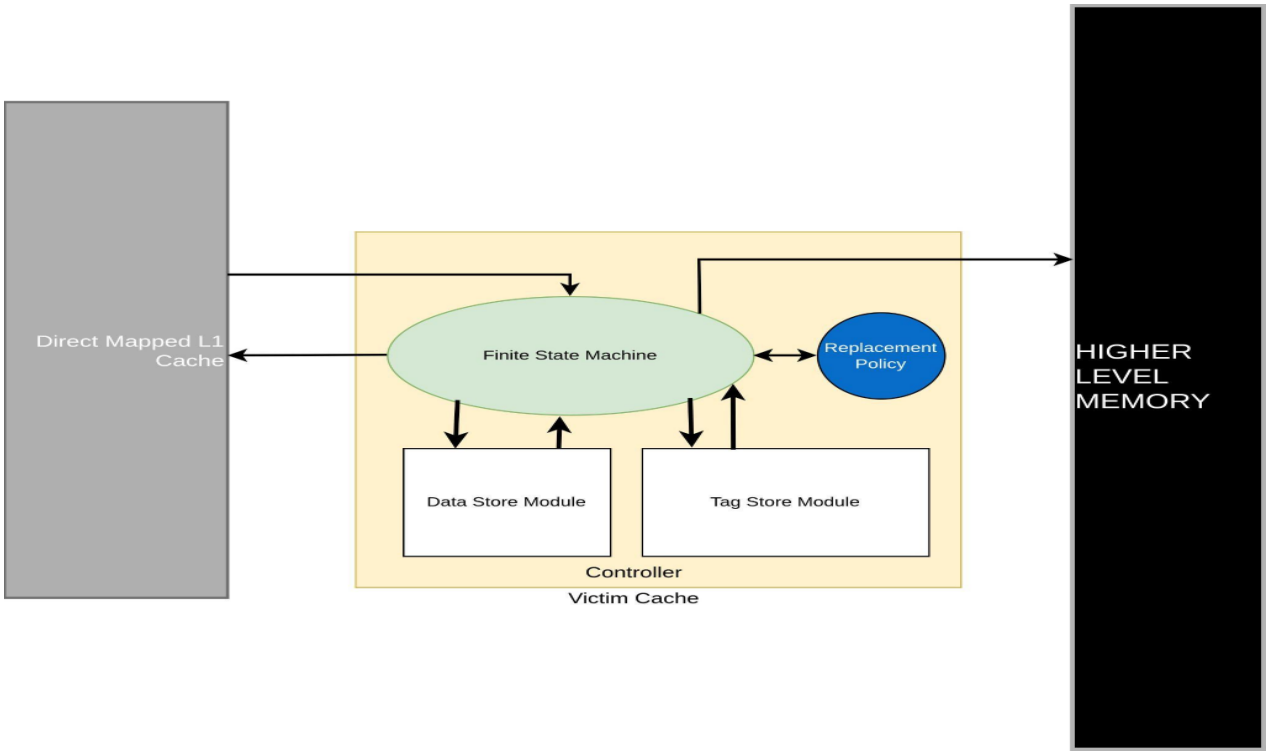
Handling Evicted Lines from L1

When a cache line is evicted from L1 and sent to the Victim Cache:

1. The controller checks the **replacement policy** to find an empty slot (**valid == 0**).
2. If an empty slot is available, the evicted line is stored there.
3. If no free slot exists, the FSM selects a line to replace based on the **FIFO replacement policy**, which is used in the Victim Cache to minimize conflict misses in L1.
4. If the line to be replaced has its **dirty bit high**, the FSM ensures it is **written back to main memory** before replacement. If the line is not dirty, it is simply replaced without writing back.

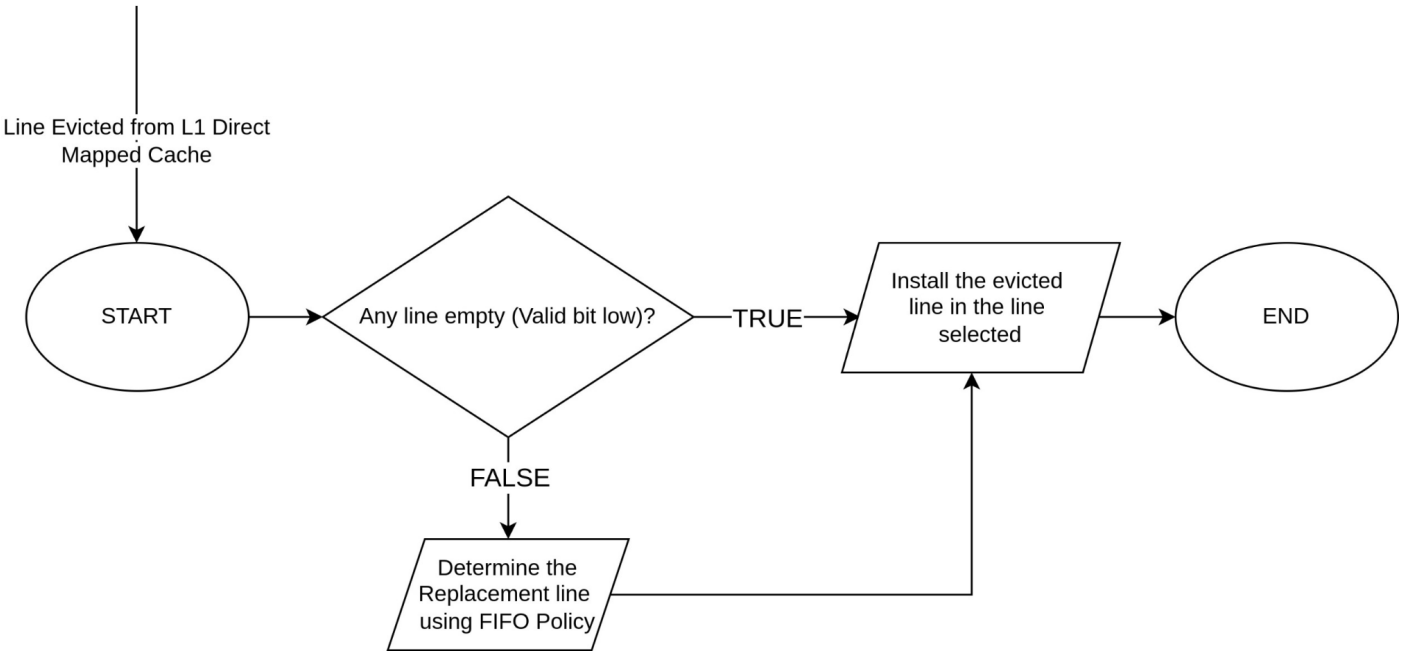
The FSM ensures that **all operations are properly timed and synchronized with the system clock**, preventing conflicts during simultaneous reads, writes, or swaps.

Design of Victim Cache

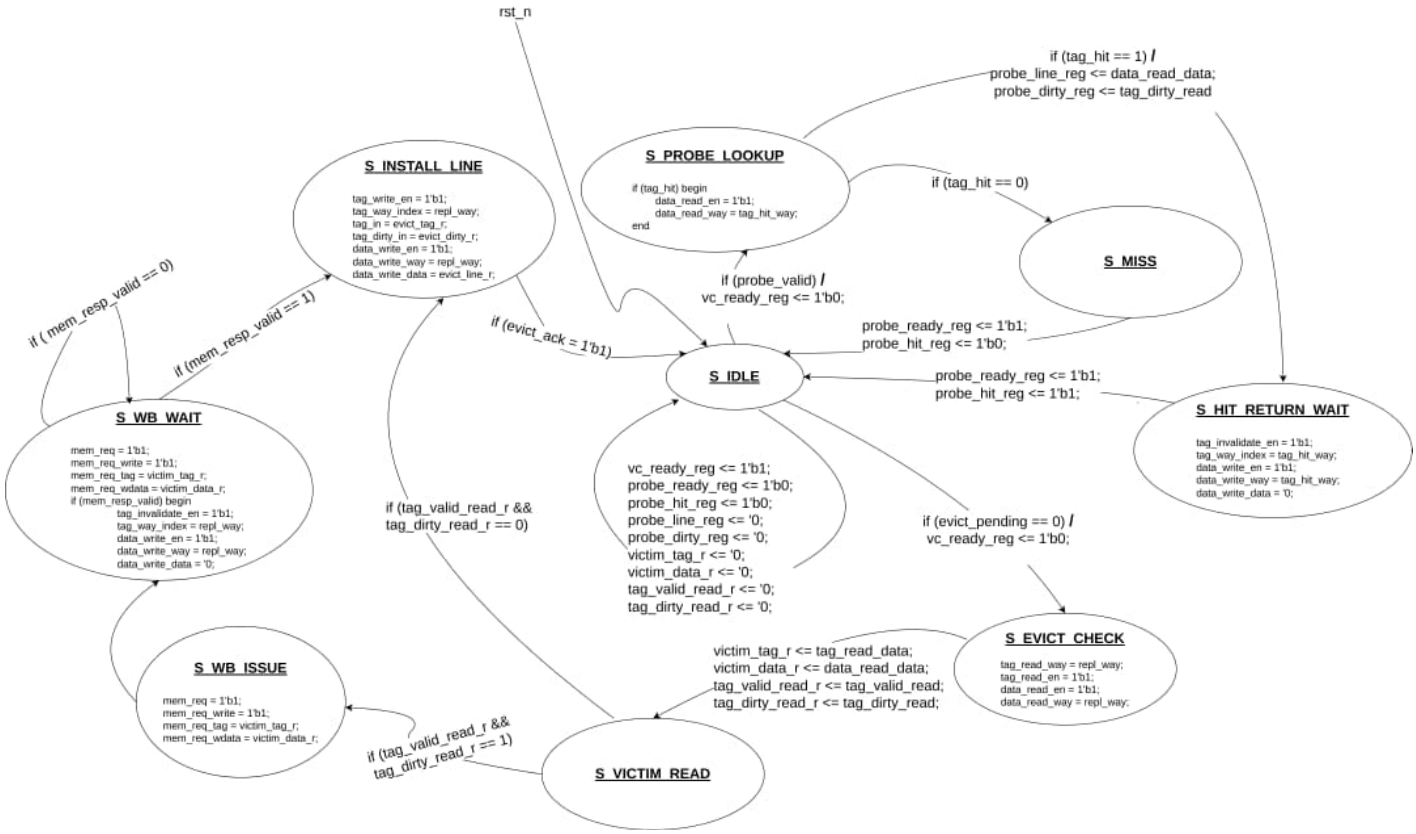


VICTIM CACHE CONTROLLER WITH FSM AND REPLACEMENT POLICY (FIFO) POINTER

The above figure also shows that the victim cache has both input and output ports for L1 Direct Mapped Cache interface but has only outputs port for Higher level Memory



FLOW CHART FOR REPLACEMENT POLICY



FINITE STATE MACHINE OF THE CONTROLLER

Testbench Design and Verification Methodology for Victim Cache

A comprehensive **SystemVerilog testbench** was developed to validate the functional correctness of the Victim Cache and its controller under a wide range of corner cases. The testbench instantiates the Victim Cache Controller along with the Tag Store and Data Store modules and applies directed test scenarios to verify correctness, timing behavior, and protocol handling.

Testbench Structure

The testbench generates a **clock and active-low reset** and models three main interfaces:

- **Probe Interface (from L1 cache):** Used to test victim cache lookup and data return behavior.
- **Eviction Interface (from L1 cache):** Used to insert evicted L1 cache lines into the Victim Cache.
- **Memory Interface:** A simplified lower-level memory model that responds to write-back requests from the Victim Cache.

All inputs are driven synchronously with the clock, and outputs are monitored using handshake signals and assertions.

Input Stimulus Methodology

- **Eviction inputs** (`evict_valid`, `evict_tag`, `evict_line`, `evict_dirty`) are asserted for a single cycle to model an L1 cache eviction.
- The testbench waits for `evict_ack` to confirm that the Victim Cache has successfully accepted the

evicted line.
- **Probe inputs** (`probe_valid`, `probe_tag`) are asserted to simulate L1 cache misses and to check whether the requested line exists in the Victim Cache.
- A simple **memory response model** is used to acknowledge write-back requests (`mem_req`) after a fixed delay by asserting `mem_resp_valid`.

Observed Outputs and Checking Mechanism

- **Probe results** are verified using `probe_hit`, `probe_ready`, and `probe_line` signals.
- **Write-back behavior** is validated by monitoring `mem_req`, `mem_req_write`, `mem_req_tag`, and `mem_req_wdata`.
- **Replacement and serialization behavior** is checked using `evict_ack`.
- Assertions and explicit checks are used to flag incorrect behavior, and the simulation terminates on any failure.

Test Scenarios Covered

The following directed test cases were implemented to ensure full coverage of the Victim Cache functionality:

1. **Insert and Probe Clean Line**
   Verifies that a clean line evicted from L1 is correctly stored in the Victim Cache and results in a hit on a subsequent probe, and that the entry is invalidated after being returned to L1.
2. **Write-Back on Dirty Victim Only**
   Confirms that a write-back to memory occurs **only when a valid and dirty victim line is evicted**, and that clean lines are overwritten without triggering a memory request.
3. **Clean Victim Replacement Without Write-Back**
   Ensures that eviction of a clean victim cache line does not generate a write-back request to memory.
4. **Probe Priority Over Pending Evictions**
   Validates that probe requests are serviced before pending eviction installations, ensuring low-latency response to L1 misses.
5. **Back-to-Back Probe Hits**
   Tests the Victim Cache's ability to handle consecutive probe requests without loss of correctness.
6. **Back-to-Back Evictions**
   Verifies correct serialization of multiple eviction requests using internal pending logic.
7. **FIFO Replacement and Pointer Wraparound**
   Confirms correct FIFO replacement behavior, including proper pointer wraparound and invalidation of old entries.

```
========Starting victim cache Corner Testing==========

TEST 1: insert & probe clean line
Probe returned correct data and hit
  PASS: returned entry invalidated in VC after probe return


TEST 2: VC write-back only for valid&&dirty entries (FIFO order)
  Installed 4 entries; way0 was dirty
mem_req_wdata: 0000000000000000000000000000000a, should be: 'hA
mem_req_wdata: 0000000000000000000000000000000a, should be: 'hA
  DUT issued mem write-back for expected tag 10
  PASS: write-back occurred and install completed (evict_ack seen)


TEST 3: clean victim eviction does not cause mem_req
NOW WAITING FOR EVICT_ACK DURING WAYS FILL
NOW WAITING FOR EVICT_ACK DURING WAYS FILL
NOW WAITING FOR EVICT_ACK DURING WAYS FILL
NOW WAITING FOR EVICT_ACK DURING WAYS FILL
NOW WAITING FOR EVICT_ACK FOR TEST PASS
  PASS: clean victim was overwritten without write-back


TEST 4: probe prioritized over pending evict
  Probe serviced while evict_pending was set (as expected)
  Install completed after probe handled


TEST 5: Back-to-back probe hits
  PASS: VC handled back-to-back probe hits


TEST 6: back-to-back evictions (pending handling)
  PASS: two back-to-back evicts correctly serialized


TEST 7: FIFO wraparound correctness
  PASS: FIFO wraparound behaves correctly
ALL TESTS PASSED


========victim cache Corner Testing Ended==========
```

TESTBENCH RESULTS SHOWING ALL THE TESTS PASSED

L1 DIRECT MAPPED CACHE

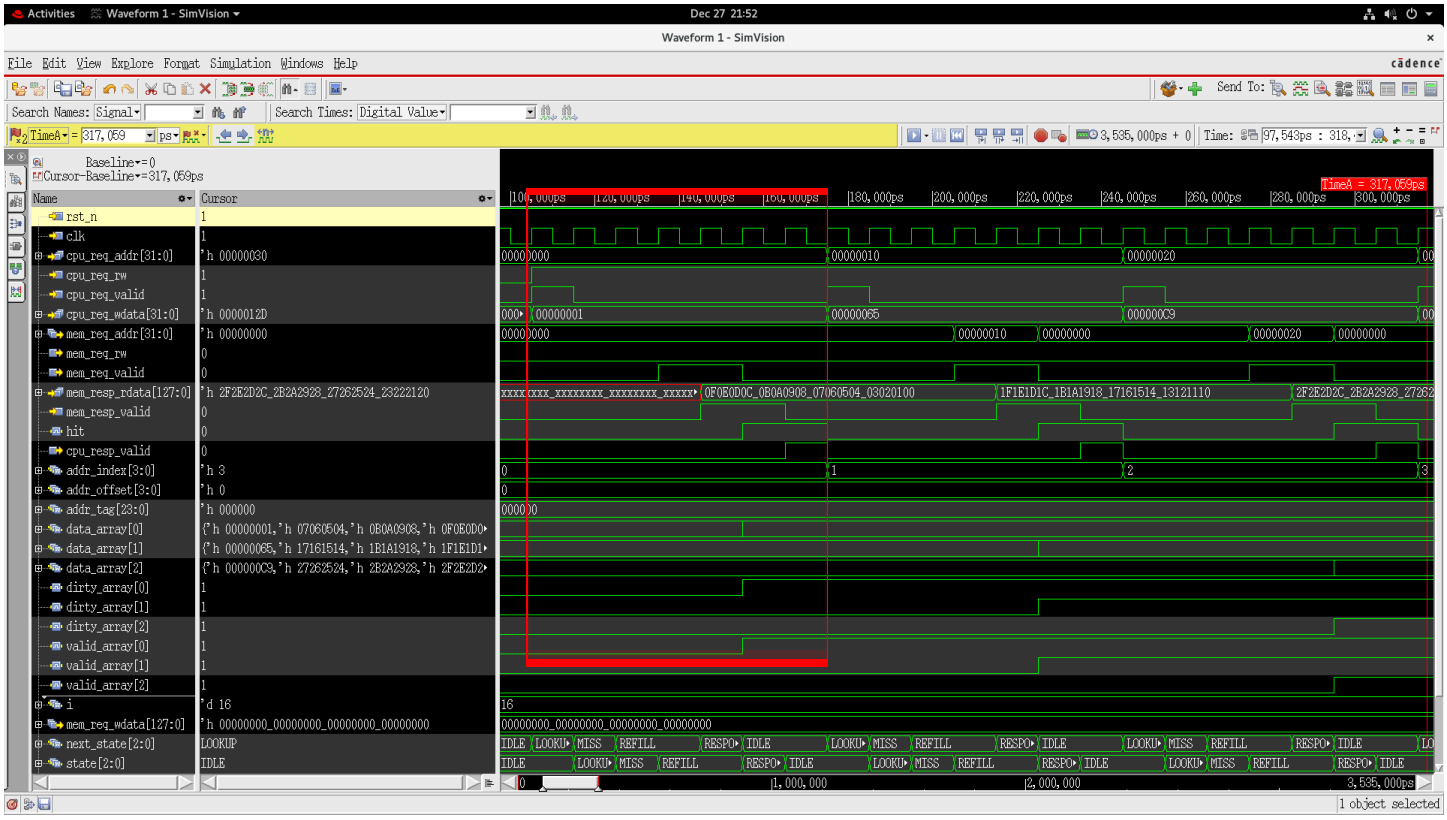
Design and Architecture Explanation

Before integrating the functionally Verified Victim Cache, the L1 Direct Mapped Cache is **first designed, implemented, and validated independently at the RTL level**. The cache is implemented using an FSM-based control structure and follows a **write-back, write-allocate policy**, allowing realistic cache behavior to be modeled. During this phase, the L1 cache is verified through simulation to ensure correct operation for all fundamental scenarios, including read and write hits, cache misses, dirty line write-backs, and block refills from lower-level memory.

This independent verification step ensures that the tag comparison logic, valid and dirty bit handling, data storage, and memory interface operate as expected.

Simulation

CPU Write Operation Waveform



CPU WRITE OPERATION (WRITE MISS)

The above simulation shows the write operation done by the cpu. The CPU initiates the write operation by asserting `cpu_req_valid` along with `cpu_req_rw = 1`, indicating a write request. The target address `cpu_req_addr` corresponds to the first cache line, and the write data is driven on `cpu_req_wdata`.

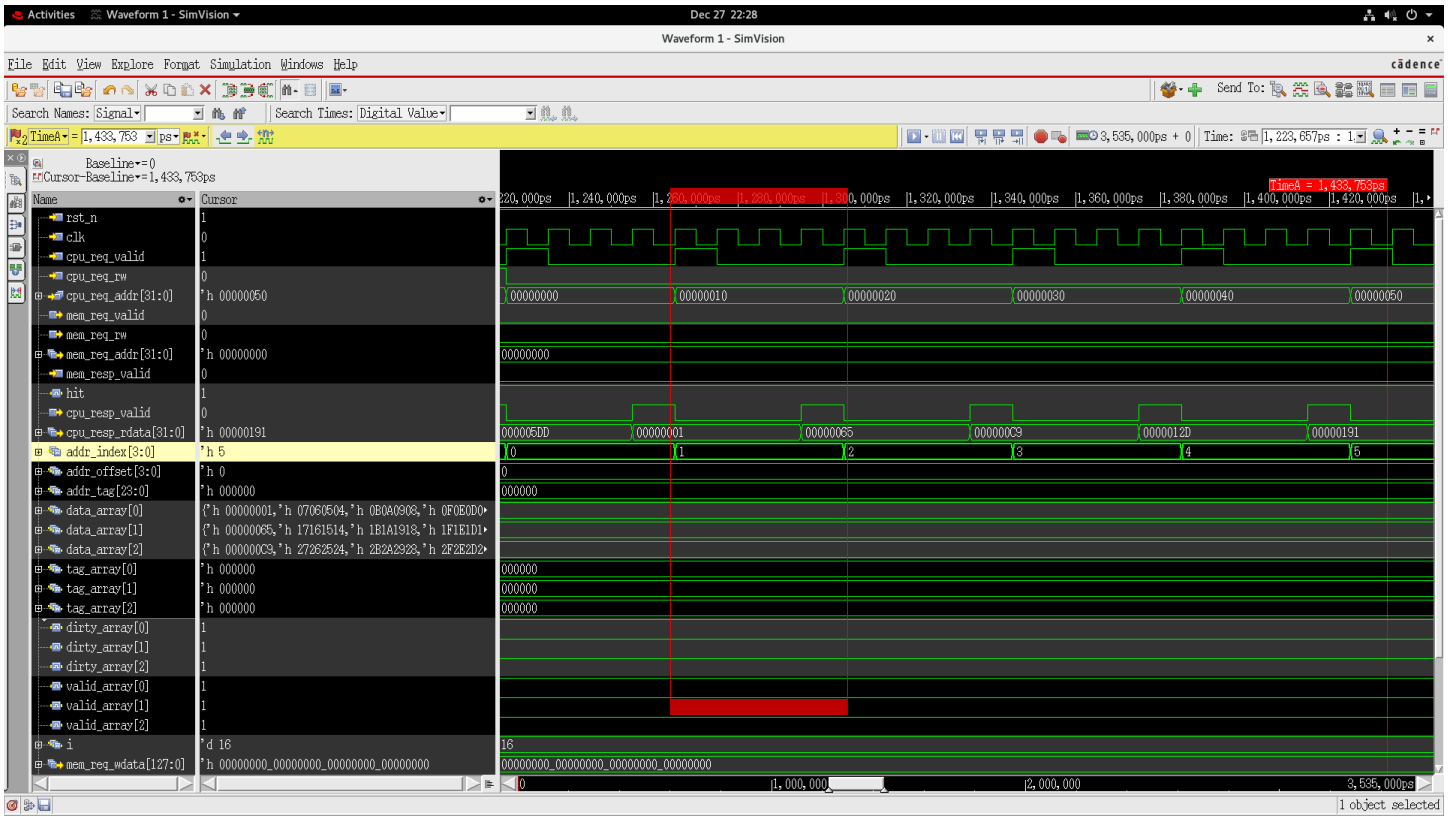
Upon receiving the request, the L1 cache enters the lookup stage and decodes the address into tag, index, and offset fields. Since the cache has just been reset, the valid bit for the indexed cache line is low. As a result, the tag comparison fails and the access is identified as a cache miss.

Following the miss detection, the controller transitions to the miss-handling state. Because the selected cache line is invalid and not dirty, no write-back operation is required. The cache then issues a read request to main memory by asserting `mem_req_valid` with `mem_req_rw = 0`, requesting the full cache line corresponding to the requested address indicated by `mem_req_addr`, which would be same as `cpu_req_addr`.

When the memory responds by asserting `mem_resp_valid`, the fetched cache line is written into the cache data array. At the same time, the tag array is updated with the new tag, the valid bit is set, and the dirty bit is asserted because the original CPU request was a write. The CPU's write data is then written into the appropriate word within the cache line based on the offset. As can be seen in the above simulation screenshot that the first word of the data block fetched from memory is changed after storing in the cache line.

Finally, the cache asserts `cpu_resp_valid` to signal completion of the write operation, after which the controller returns to the idle state, completing the first write transaction. Similarly the subsequent write operations are performed.

CPU Read Operation Waveform



CPU READ OPERATION (READ HIT)

The CPU initiates the read operation by asserting `cpu_req_valid` with `cpu_req_rw = 0`, indicating a read request. The read address `cpu_req_addr` is set to the second cache-line address (`1 * LINE_BYTES`).

The L1 cache decodes the address into its tag, index, and offset fields and enters the lookup stage. Since this cache line was already brought into the cache during the previous write operation, the valid bit for the indexed line is set and the stored tag matches the requested tag. As a result, the access is identified as a cache hit.

Because the request is a read hit, no memory request is generated and `mem_req_valid` remains deasserted.

The cache directly selects the requested word from the data array using the offset bits.

The read data is then driven onto `cpu_resp_rdata`, and the cache asserts `cpu_resp_valid` to indicate that the data is available to the CPU. After completing the response, the controller returns to the idle state, concluding the first read-hit operation. Similarly the subsequent read operations are performed.

MODIFICATION OF THE L1 DIRECT MAPPED CACHE AND INTERFACING WITH VICTIM CACHE

The modified L1 direct-mapped cache introduces a Victim Cache to reduce conflict misses and improve overall cache efficiency. Compared to the earlier standalone L1 implementation, the replacement and miss-handling behavior is significantly enhanced through two dedicated interaction paths with the Victim Cache: a **Victim Cache probe path** and an **eviction path**.

Comparison With The Previous Traditional Direct Mapped Cache

In the previous L1 cache design, when a cache miss occurred and the indexed cache line was already occupied, the replacement policy depended on the dirty bit. Dirty lines were written back, while valid but clean lines were overwritten and discarded. In the enhanced design, this behavior is changed: **any valid L1 cache line—regardless of whether it is clean or dirty—is evicted to the VictimCACHE Cache before replacement**. This ensures that recently evicted clean lines are preserved and remain accessible.

Victim Cache Probe Path

On an L1 cache miss, the cache does not immediately access main memory. Instead, it first probes the Victim Cache to determine whether the requested block was recently evicted. During this operation, the L1 cache asserts `vc_probe_valid` along with `vc_probe_tag`, which consists of the concatenated tag and index of the requested block.

The Victim Cache responds using a handshake mechanism signaled by `vc_probe_ready`. If a hit occurs, the Victim Cache asserts `vc_probe_hit` and returns the corresponding cache line through `vc_probe_line`, along with its dirty status via `vc_probe_dirty`. These values are synchronously captured by the L1 cache and used during the subsequent installation phase.

Eviction Path to Victim Cache

If the L1 cache line selected for replacement is valid, it is evicted to the Victim Cache instead of being overwritten or written back directly to memory. The evicted line's metadata and data are transferred using `vc_evict_valid`, `vc_evict_tag`, `vc_evict_line`, and `vc_evict_dirty`.

The L1 cache waits for the Victim Cache to acknowledge acceptance of the evicted line through `vc_evict_ack` before invalidating its local cache entry. This handshake-based eviction ensures correct synchronization and preserves data integrity between the two cache structures.

By redirecting all valid evictions—both clean and dirty—to the Victim Cache and probing it on every L1 miss, the enhanced architecture keeps recently displaced blocks close to the processor. This significantly reduces conflict misses and minimizes unnecessary main memory accesses, improving effective cache performance.

Top-Level Integration Module

The `top` module serves as the system-level integration point that connects the CPU interface, the modified L1 Direct Mapped Cache, the Victim Cache, and the main memory interface into a cohesive cache hierarchy. Its primary purpose is to coordinate communication between these components while maintaining a clean separation of responsibilities between cache control logic and system interconnect logic.

At the highest level, the CPU interfaces only with the L1 cache through the request and response signals (`cpu_req_valid`, `cpu_req_rw`, `cpu_req_addr`, `cpu_req_wdata`, `cpu_resp_valid`, and `cpu_resp_rdata`). This abstraction ensures that neither the CPU nor the testbench is aware of the presence of the Victim Cache, preserving modularity and allowing the cache hierarchy to be extended transparently.

L1 Cache and Victim Cache Connectivity

The modified L1 cache (`l1_cache_dm_with_vc`) is instantiated within the top module and is directly connected to the Victim Cache through two distinct interfaces: the **probe interface** and the **eviction interface**. These interfaces carry all necessary control, data, and handshake signals required for Victim Cache interaction, such as `vc_probe_valid`, `vc_probe_hit`, `vc_evict_valid`, and `vc_evict_ack`.

The Victim Cache controller operates as an independent module with its own internal storage and control logic. It responds to probe requests from the L1 cache, supplies evicted lines when a Victim Cache hit occurs, and accepts evicted L1 cache lines regardless of whether they are clean or dirty. This design choice aligns with the enhanced replacement policy, where all valid L1 lines are preserved in the Victim Cache instead of being discarded.

The `vc_ready` signal is used to indicate Victim Cache availability, ensuring that the L1 cache only initiates probe or eviction transactions when the Victim Cache is ready to accept them. This handshake-based communication improves robustness and avoids data loss during concurrent operations.

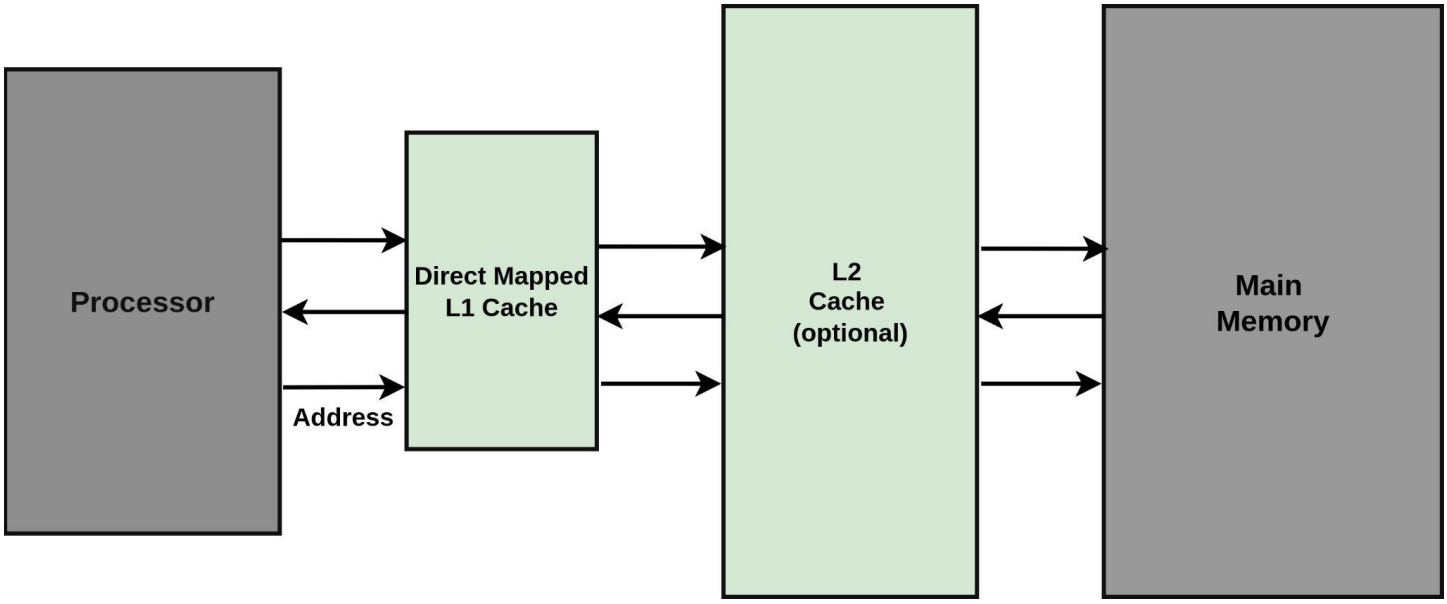
Shared Memory Interface Arbitration

The top module also manages access to the shared main memory interface. Both the L1 cache and the Victim Cache may generate memory requests: the L1 cache issues read requests on **complete cache misses** (miss in both direct-mapped and the victim), while the Victim Cache may initiate memory accesses for write-back.

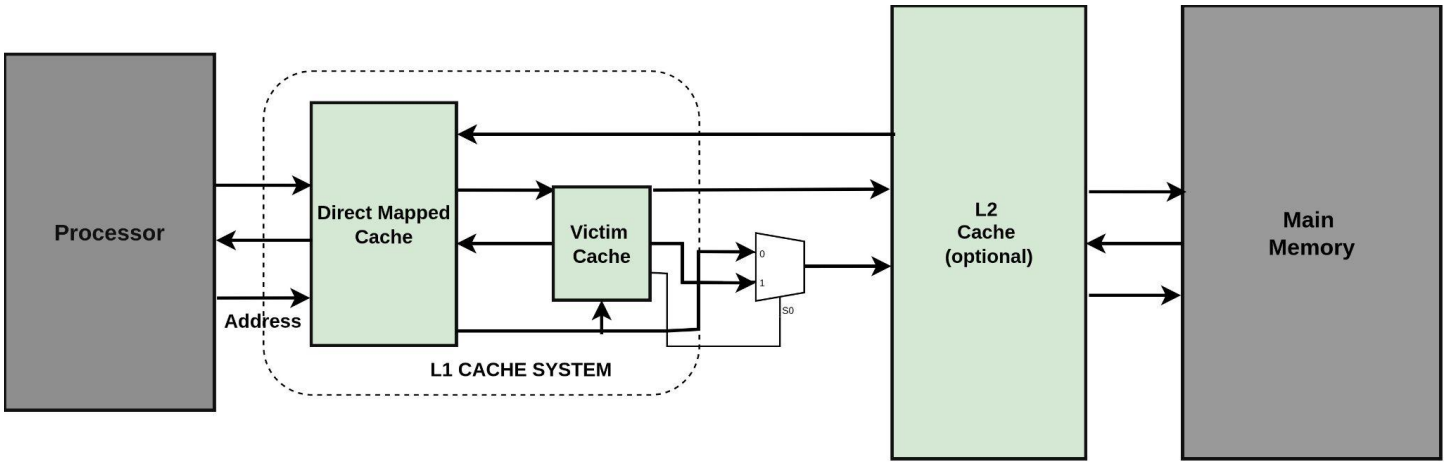
To handle this, the top module implements a simple priority-based multiplexer on the memory interface. Victim Cache memory requests are given priority over L1 cache requests. When the Victim Cache asserts its memory request signal, its request parameters are forwarded to the external memory interface. Otherwise, the L1 cache's memory request signals are passed through unchanged.

This centralized arbitration logic ensures that memory access remains consistent and conflict-free while keeping the individual cache modules simpler and more focused on their core functionality.

Design of Victim Cache



L1 WITHOUT VICTIM CACHE



L1 WITH VICTIM CACHE

The Final Pre-Synthesis Simulation

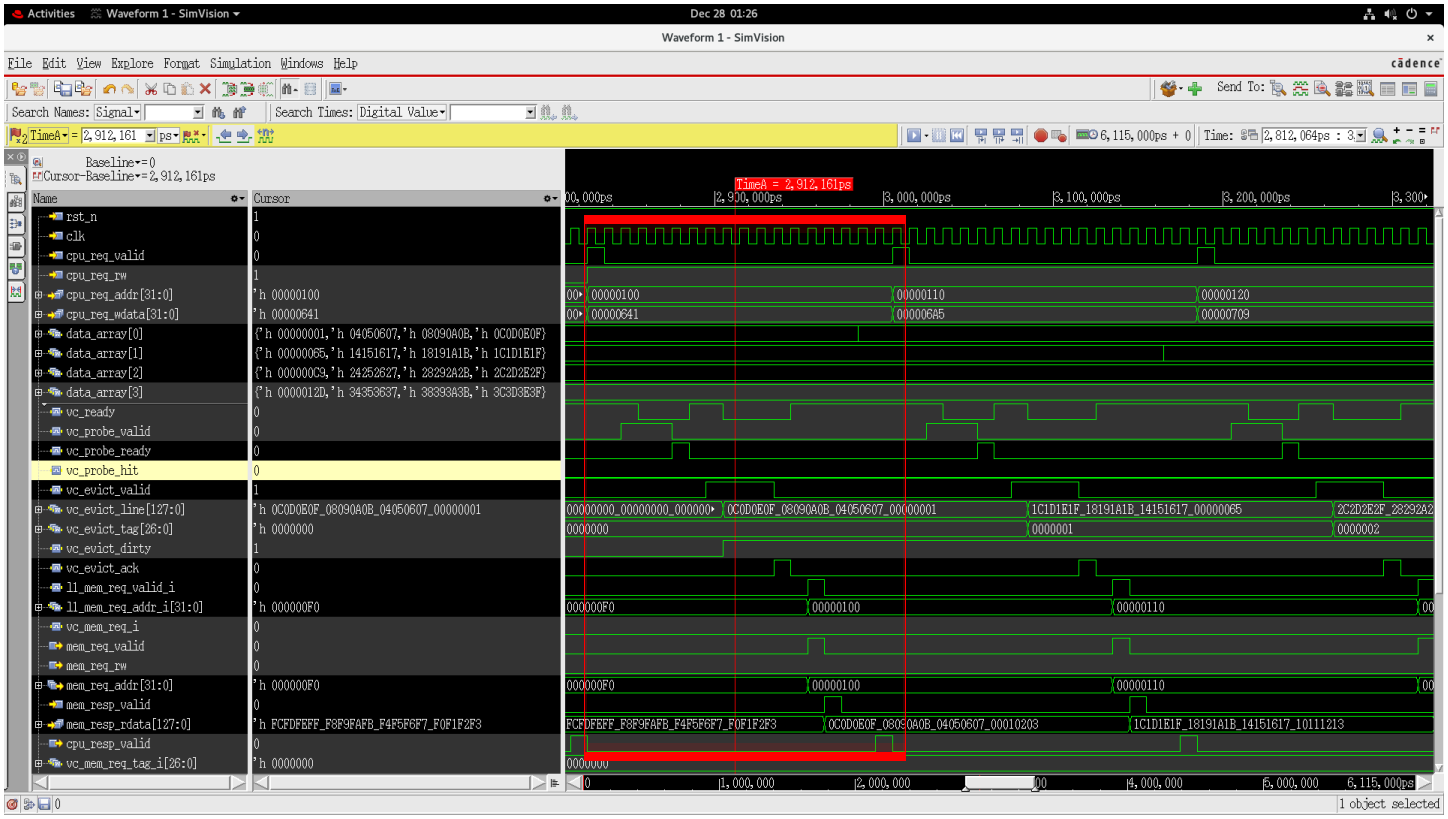
A testbench is written which models a CPU, main memory, and the cache hierarchy to validate the functionality and performance of the modified L1 Direct Mapped Cache with an integrated Victim Cache. The testbench provides a controlled simulation environment where read and write transactions can be issued from the CPU, and responses from both the L1 cache and Victim Cache can be observed and verified.

The testbench generates a clock (`clk`) with a 10 ns period and provides an active-low reset (`rst_n`) to initialize all components. It instantiates the top-level module, which contains both the modified L1 cache and the Victim Cache, and connects them to a simple byte-addressable memory model to respond to cache misses and Victim Cache fetches. Memory reads and writes are modeled at the cache line level, ensuring realistic behavior for refill and write-back operations.

Within the testbench, CPU transactions are performed using two tasks: `cpu_read` and `cpu_write`. These tasks drive the CPU request signals (`cpu_req_valid`, `cpu_req_rw`, `cpu_req_addr`, `cpu_req_wdata`) and wait for the corresponding response from the cache hierarchy (`cpu_resp_valid`, `cpu_resp_rdata`). The sequence of operations is carefully designed to exercise key cache behaviors, including:

1. **Cache Initialization and Reset Verification**
   The simulation begins by asserting the active-low reset (`rst_n`) to initialize the L1 cache, Victim Cache, and all control logic. This ensures that all valid and dirty bits are cleared, internal state machines return to their idle states, and no residual data remains from previous operations.
2. **Filling the L1 Cache (Write Misses and Allocations)**
   Sequential CPU write operations are issued to addresses that map to all L1 cache lines. Since the cache is initially empty, each access results in a write miss, triggering block allocation and refill from main memory. This scenario validates write-allocate behavior, correct tag installation, valid bit assertion, and dirty bit setting.
3. **Re-accessing Cached Lines (L1 Cache Hits)**
   After the L1 cache is fully populated, CPU read operations are performed on the same addresses. These accesses result in read hits, confirming that data was stored correctly, tag comparisons succeed, and the cache can serve data without accessing either the Victim Cache or main memory.
4. **Forcing L1 Cache Evictions into the Victim Cache**
   Additional write operations are issued to new addresses beyond the L1 cache capacity. These accesses force evictions of existing L1 cache lines. In the modified design, any valid line—regardless of whether it is clean or dirty—is evicted to the Victim Cache. This scenario verifies the eviction path, handshaking (`vc_evict_valid` / `vc_evict_ack`), and correct transfer of tag, data, and dirty status to the Victim Cache.
5. **Victim Cache Hit Verification**
   The testbench re-accesses addresses corresponding to recently evicted lines. On an L1 miss, the L1 cache probes the Victim Cache. A successful Victim Cache hit confirms correct probe signaling, hit detection, data return, and swap/installation of the cache line back into the L1 cache.
6. **Mixed Read and Write Access Patterns**
   A combination of reads and writes is applied to both cached and evicted addresses. This scenario validates correct handling of dirty data, preservation of modified values, and consistent behavior across repeated swaps between the L1 cache and Victim Cache.

EVICTING LINES INTO THE VICTIM CACHE SIMULATION WAVEFORM



EVICTING LINES FROM L1 TO VICTIM WAVEFORM

The above simulation shows the first iteration of evicting a line from the L1 cache into the Victim Cache. In this scenario, the CPU initiates a write operation by asserting `cpu_req_valid` along with `cpu_req_rw = 1`, indicating a write request. The target address `cpu_req_addr` corresponds to a new cache line that maps to an index already occupied in L1, and the write data is supplied on `cpu_req_wdata`.

Upon receiving the request, the L1 cache enters the `S_LOOKUP` stage and decodes the address into its tag, index, and offset fields. The cache determines a miss because the requested block does not match the tag of the existing line at that index (`hit = valid_array[addr_index] && (tag_array[addr_index] == addr_tag)` evaluates to 0).

Since the block is not found in L1, the cache transitions to `S_VC_PROBE` to check the Victim Cache. L1 asserts `vc_probe_valid` and drives `vc_probe_tag` with the concatenated tag and index of the requested block. The Victim Cache responds using the handshake signals `vc_probe_ready` and `vc_probe_hit`. In this first iteration, `vc_probe_hit = 0` indicating the requested block is not present in the Victim Cache.

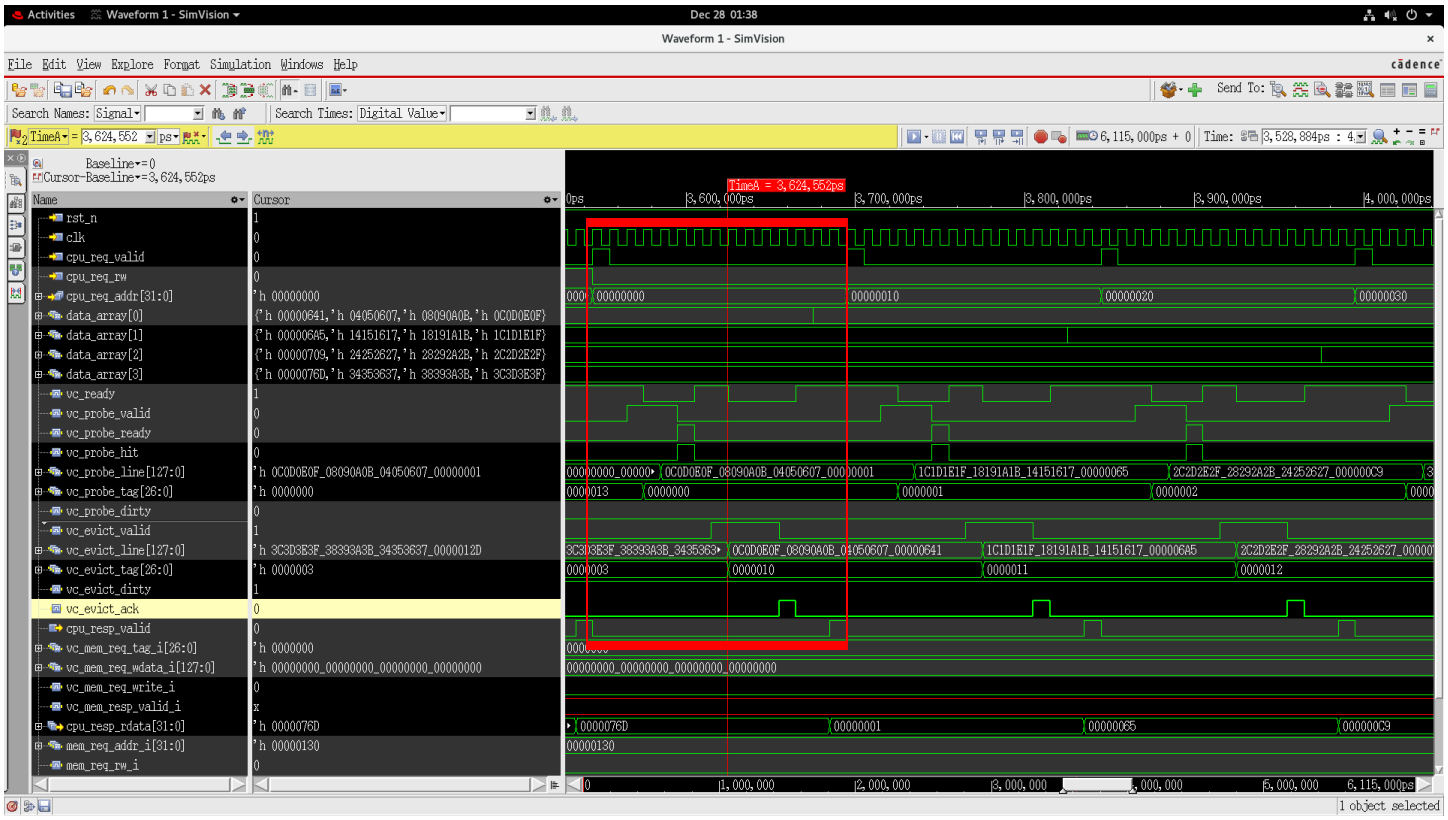
Next, the cache identifies that the current line at `addr_index` is valid and must be evicted. In the `S_EVICT_TO_VC` state, L1 asserts `vc_evict_valid` and sends the evicted line's tag (`vc_evict_tag`), data (`vc_evict_line`), and dirty status (`vc_evict_dirty`) to the Victim Cache. The cache then waits in `S_WAIT_EVICT_ACK` until the Victim Cache acknowledges the eviction via `vc_evict_ack`. Once acknowledged, L1 invalidates the local entry by clearing `valid_array[addr_index]`,

`dirty_array[addr_index]`, and `tag_array[addr_index]`.

Finally, L1 fetches the new block from memory by asserting `mem_req_valid` and `mem_req_rw = 0` in the `S_REFILL_REQ` state. Upon receiving `mem_resp_valid`, the new cache line is installed in `data_array[addr_index]` and tagged with `addr_tag`. If the original CPU request was a write, the corresponding word within the cache line is updated with `cpu_req_wdata` and `dirty_array[addr_index]` is set. The cache asserts `cpu_resp_valid` in the `S_RESPOND` state, signaling the completion of the write operation.

This process ensures that the evicted line is preserved in the Victim Cache, reducing the penalty of future accesses to recently displaced data.

VICTIM CACHE HITS SIMULATION WAVEFORM



VICTIM CACHE HIT TEST WAVEFORM

The above simulation shows the first iteration of a Victim Cache hit. In this scenario, the CPU initiates a read operation by asserting `cpu_req_valid` with `cpu_req_rw = 0`, indicating a read request. The target address `cpu_req_addr` corresponds to a cache line that was previously evicted from L1 and is currently stored in the Victim Cache.

Upon receiving the request, the L1 cache enters the `S_LOOKUP` stage and decodes the address into tag, index, and offset fields. The cache determines a miss because the requested block does not match the tag of the existing line at that index (`hit = valid_array[addr_index] && (tag_array[addr_index]`

`== addr_tag)` evaluates to 0).

Since the block is not found in L1, the cache transitions to `S_VC_PROBE` and asserts `vc_probe_valid` along with `vc_probe_tag` containing the concatenated tag and index of the requested block. The Victim Cache responds using the handshake signals `vc_probe_ready` and `vc_probe_hit`. In this first iteration, `vc_probe_hit = 1` indicating the requested block is present in the Victim Cache. The cache also captures the returned data on `vc_probe_line` and the dirty status on `vc_probe_dirty`.

The L1 cache then proceeds to the `S_AFTER_VC` state. Since the cache line at `addr_index` is already valid, it is first evicted to the Victim Cache using `vc_evict_valid`, `vc_evict_tag`, `vc_evict_line`, and `vc_evict_dirty`, and the cache waits for `vc_evict_ack`. Once acknowledged, the L1 cache installs the line fetched from the Victim Cache into `data_array[addr_index]`, updates `tag_array[addr_index]` with `addr_tag`, and sets the `valid_array[addr_index]` and `dirty_array[addr_index]` according to the captured `vc_probe_dirty`.

Finally, the requested word within the cache line is selected using the offset, and `cpu_resp_valid` is asserted along with `cpu_resp_rdata` containing the read data. This sequence completes the first Victim Cache hit, allowing the CPU to access the recently displaced data with minimal latency.

Performance Evaluation of L1 Cache With and Without Victim Cache

The performance comparison between the L1 cache without a Victim Cache and the modified L1 cache with an integrated Victim Cache was carried out by running the **exact same test scenarios** on both designs and carefully counting the total number of cycles consumed which ensured a fair comparison. In the current design, different access paths are modeled with realistic latency costs: an **L1 cache hit completes in 3 cycles**, a **Victim Cache hit completes in approximately 8 cycles**, while a **main memory access incurs a very large penalty, modeled as hundreds of cycles**. To capture this behavior, the testbench maintains a `cycle_count` variable that is incremented by **1 on every positive clock edge**, representing normal pipeline progress, while additional penalties are added when slower events occur. Specifically, whenever the L1 cache enters the refill state (`S_REFILL_REQ`), indicating a complete miss that goes to main memory, the cycle counter is incremented by 100 to model the high latency of memory access. Similarly, when a Victim Cache hit is detected through the `vc_probe_ready` and `vc_probe_hit` signals, a smaller penalty is added to reflect the intermediate latency of accessing the Victim Cache. By collecting statistics such as L1 hits, Victim Cache hits, complete misses, and total cycle count, the testbench quantitatively shows that the Victim Cache significantly reduces overall execution time. Many accesses that would have previously triggered expensive main memory transactions are instead serviced by the Victim Cache at a much lower cost. As a result, although Victim Cache hits are slower than pure L1 hits, they are orders of magnitude faster than main memory accesses, leading to a substantial improvement in average memory access time and overall system performance.

```
========== L1 CACHE PERFORMANCE WITHOUT VICTIM CACHE ==========
CPU accesses      : 54
L1 hits           : 18
VC hits           : 0 (not present)
Complete misses   : 36
Cycle Count       : 3912

Each main memory access takes 100s of cycles for access
L1 hit takes 3 cycles
================================================================
```

```
==== L1 CACHE PERFORMANCE WITH VICTIM CACHE ====
CPU accesses      : 54
L1 hits           : 18
VC hits           : 16
Complete misses   : 20
Cycle Count       : 2586

Each main memory access takes 100s of cycles for access
L1 hit takes 3 cycles
Victim Cache hit takes 8-9 cycles
================================================
```

CPU EXECUTION TIME

CPU Execution Time = CPU Clock Cycles for a program x Clock Period
For a 100MHz clock cycle, the time period is 10ns.

	L1 Without Victim Cache	**L1 With Victim Cache**
Clock Cycles Taken	3912	2586
Time Consumed	39.1 us	25.8 us

For the **L1 cache without a Victim Cache**, the program requires **3912 clock cycles**, resulting in a total execution time of approximately **39.1 µs**. In contrast, the **L1 cache with an integrated Victim Cache** completes the same workload in **2586 clock cycles**, corresponding to **25.8 µs**. This reduction in execution time clearly demonstrates the performance benefit of the Victim Cache, as it significantly lowers the number of costly main memory accesses and improves the overall average memory access time.

Synthesis (Front-End or Logical Design)

Synthesis is the process of **transforming a high-level digital design described in RTL (Register Transfer Level) into a gate-level implementation** using standard cells from a technology library. Essentially, synthesis translates **behavioral hardware descriptions** into a **structural netlist** that can be physically implemented on silicon.

PURPOSE OF SYNTHESIS

Synthesis is a critical step in digital chip design because of the following reasons

1. Translation to Hardware

Synthesis translates a high-level RTL description of a digital design into a gate-level implementation using standard cells from a **logical/timing library** (such as **.lib** or **Liberty** files). While RTL code specifies the functional behavior of a design—such as arithmetic operations, multiplexing, or conditional logic—it does not define the specific electrical gates required for silicon realization. The synthesizer maps these abstract behavioral operations to specific logic gate models defined in the library, such as AND, OR, XOR, and flip-flops. During this process, the tool uses the electrical data within the **.lib** files to ensure the design meets performance goals, effectively bridging the gap between a designer's abstract logic and a structural netlist ready for physical implementation.

2. Timing Analysis Preparation

Timing analysis is essential to ensure that the design meets its target clock frequency. RTL itself does not include timing information, so synthesis prepares the design for timing analysis by assigning delay values to each logic gate and flip-flop based on the **.lib** files. These files contain detailed timing models for each cell, including rise and fall delays, setup and hold times, and input-to-output propagation delays. During synthesis, the tool calculates the delay along each path of the design, generating timing reports that highlight critical paths and slack. This information forms the basis for Static Timing Analysis (STA) and subsequent optimization steps, allowing the designer to ensure that signals propagate within the required clock period.

3. Area and Power Estimation

Synthesis provides preliminary estimates of the chip area and power consumption, which are crucial for design planning. Each standard cell in the library has a specified physical area and power characteristics, including dynamic and leakage power. When the synthesizer maps RTL constructs to these cells, it sums the area of all instantiated cells to provide the **total cell area**. While this provides a foundational estimate, it is important to note that the final silicon area will also include routing overhead (the space required for metal interconnects) defined later in the physical layout stage. Similarly, the tool estimates power consumption based on cell activity and switching frequency, allowing designers to anticipate whether the design fits within the specified power budget.

4. Optimization

One of the key roles of synthesis is optimizing the design for timing, area, and power. Optimization involves adjusting the gate-level implementation to meet the constraints specified in the **SDC (Synopsys Design Constraints)** file. For example, if a path is too slow, the tool may restructure the logic, replace slower gates with faster (often larger) equivalents, or insert buffers. To reduce area, it may merge redundant logic or share

gates across multiple functions. The synthesizer decides which optimizations to apply by analyzing the design against the specified constraints, using heuristics and cost functions to balance the inherent trade-offs among timing, area, and power.

INPUTS AND OUTPUTS OF SYNTHESIS

Inputs / Required Files

- **RTL Source Files (Verilog/VHDL)**
  Describe the functional behavior and structure of the digital design at the register-transfer level.
- **Standard Cell Library Files (.lib)**
  Provide timing, power, and area information for technology-specific standard cells used during gate mapping and optimization.
- **Timing Constraint File (SDC)**
  Defines clock specifications, input/output delays, and timing requirements that guide synthesis optimization.
- **Synthesis Script (TCL)**
  Controls the synthesis flow by specifying design files, libraries, constraints, and optimization settings.

Outputs of Synthesis

- **Gate-Level Netlist**
  A structural representation of the design mapped to standard cells, ready for physical implementation.
- **Timing Reports**
  Provide setup, hold, slack, and critical path information for timing verification.
- **Area Report**
  Summarizes total standard cell area and cell utilization.
- **Power Report**
  Estimates dynamic, leakage, and total power consumption of the synthesized design.

SYNTHESIS OF THE L1 CACHE (DIRECT MAPPED WITH VICTIM CACHE)

Creation of Top Wrapper Module with I/O Pads

A **top-level wrapper module (`top_wrapper.sv`)** was created to encapsulate the entire cache system. This wrapper was necessary to **instantiate I/O pad cells** and connect internal design signals to the external chip interface. While core RTL modules operate on abstract input and output ports, physical implementation requires explicit pad cells to interface the core logic with off-chip signals.

I/O pads serve as the electrical interface between on-chip logic and external pins. They provide signal buffering, electrostatic discharge (ESD) protection, and proper voltage-level handling. By instantiating pad cells in the top wrapper, the design becomes compatible with synthesis and subsequent physical design stages, enabling correct timing constraint application and pin assignment.

Additional Requirements for Synthesis

Synthesis was performed assuming a **typical process, voltage, and temperature (PVT) corner**, using standard cell and I/O pad libraries characterized for the typical case. The libraries **`tcbn65lptc.lib`** for core logic

cells and **`tpzn65lpgv2od3tc.lib`** for I/O pad cells were selected to maintain consistency in timing and power estimation across the design. This corner represents nominal operating conditions and is commonly used during initial synthesis to obtain balanced timing, area, and power results. Additionally, the **SDC file specified timing constraints relative to the I/O pad pins rather than internal module ports**, ensuring that timing analysis during synthesis accurately reflected the chip-level input and output interfaces. For Synthesis, Cadence's Genus was used

Inputs

```
##############################################################################################
## Environment Setup
##############################################################################################

# Print environment info
if {[file exists /proc/cpuinfo]} {
    sh grep "model name" /proc/cpuinfo
    sh grep "cpu MHz"    /proc/cpuinfo
}
puts "Hostname : []"

# Library setup
set_db / .init_lib_search_path /home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/65_lp_libs

read_libs {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/65_lp_libs/tcbn65lptc.lib /home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/
frontend/65_lp_libs/tpzn65lpgv2od3tc.lib}

##############################################################################################
## Design Setup
##############################################################################################

set DESIGN top_wrapper ;# Top-level design module
set GEN_EFF  high          ;# Generic synthesis effort
set MAP_OPT_EFF high       ;# Mapping/Optimization effort

##############################################################################################
## Read and Elaborate RTL
##############################################################################################

read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/data_store.sv}
read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/tag_store.sv}
read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/victim_cache_controller.sv}
read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/l1_cache_dm_fsm.sv}
read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/top.sv}
read_hdl -sv {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/RTL/top_wrapper.sv}
elaborate $DESIGN
check_design -unresolved

##############################################################################################
## Constraints
##############################################################################################

# (Update constraints file if needed)
read_sdc {/home/cc/Downloads/Victim-Cache-Design-RTL-GDSII--main/victim_cache_layout_data/frontend/constraints/victim_cache_controller_io.sdc}

##############################################################################################
## Prevent Scan Flip-Flops
##############################################################################################

# Block scan flip-flops (various naming styles)
set_dont_use [get_lib_cells */S*]

##############################################################################################
## Synthesis Flow
##############################################################################################

# Step 1: Generic synthesis
set_db / .syn_generic_effort $GEN_EFF
syn_generic

# Step 2: Technology mapping
set_db / .syn_map_effort $MAP_OPT_EFF
syn_map

# Step 3: Optimization
set_db / .syn_opt_effort $MAP_OPT_EFF
syn_opt

##############################################################################################
## Write Outputs
##############################################################################################

file mkdir ./Netlist
file mkdir ./out
file mkdir ./RPT

# Netlist
write_hdl $DESIGN -mapped   >> ./Netlist/${DESIGN}_map.v

# Constraints for backend
write_sdc $DESIGN           >> ./out/${DESIGN}_map.sdc

# Delay annotation
write_sdf -design $DESIGN   >> ./out/${DESIGN}_map.sdf

# HTML report
report_metric -format html -file synthesis_report.html

##############################################################################################
## Reports
##############################################################################################

report timing  -full_pin_names   >> ./RPT/${DESIGN}_timing.rpt
report area                      >> ./RPT/${DESIGN}_area.rpt
report gates                     >> ./RPT/${DESIGN}_gates.rpt
report power                     >> ./RPT/${DESIGN}_power.rpt
report qor -levels_of_logic      >> ./RPT/${DESIGN}_qor.rpt

##############################################################################################
## GUI (Optional)
##############################################################################################
gui_show
# quit

Tcl  Tab Width: 4  Ln 27, Col 32  INS
```

script.tcl FILE

```
##############################################################
# Top Module – Timing Constraints
##############################################################

##############################################################
# Clock definition
##############################################################

# Single system clock
# 10.0 ns period = 100 MHz
create_clock -name clk -period 20.0 -waveform {0 10} [get_pins clk_pad/XC]

# Clock uncertainty (setup + hold margin for CTS)
set_clock_uncertainty -setup 0.2 [get_clocks clk]
set_clock_uncertainty -hold 0.05 [get_clocks clk]

# Clock transition (reasonable slew)
set_clock_transition 0.2 [get_clocks clk]

##############################################################
# Reset constraints
##############################################################

# Asynchronous active-low reset
set_false_path -from [get_ports rst_n]

##############################################################
# Input constraints - CPU Interface
##############################################################

set_input_delay -max 3.0 -clock clk [get_pins {cpu_req_valid_pad/C}]
set_input_delay -max 3.0 -clock clk [get_pins {cpu_req_rw_pad/C}]
set_input_delay -max 3.0 -clock clk [get_pins {cpu_addr_pad*/C}]
set_input_delay -max 3.0 -clock clk [get_pins {cpu_wdata_pad*/C}]

# Memory response interface
set_input_delay -max 3.0 -clock clk [get_pins {mem_resp_valid_pad/C}]
set_input_delay -max 3.0 -clock clk [get_pins {mem_rdata_pad*/C}]

# Set minimum input delays (if needed)
set_input_delay -min 0.5 -clock clk [get_pins {cpu_req_valid_pad/C}]
set_input_delay -min 0.5 -clock clk [get_pins {cpu_req_rw_pad/C}]
set_input_delay -min 0.5 -clock clk [get_pins {cpu_addr_pad*/C}]
set_input_delay -min 0.5 -clock clk [get_pins {cpu_wdata_pad*/C}]
set_input_delay -min 0.5 -clock clk [get_pins {mem_resp_valid_pad/C}]
set_input_delay -min 0.5 -clock clk [get_pins {mem_rdata_pad*/C}]

set_input_delay -clock clk -max 3.0 [get_pins rst_pad/C]
set_input_delay -clock clk -min 0.5 [get_pins rst_pad/C]

##############################################################
# Output constraints - CPU Response and Memory Request
##############################################################

# CPU response interface
set_output_delay -max 3.0 -clock clk [get_pins {cpu_resp_valid_pad/I}]
set_output_delay -max 3.0 -clock clk [get_pins {cpu_resp_data_pad*/I}]

# Memory request interface
set_output_delay -max 3.0 -clock clk [get_pins {mem_req_valid_pad/I}]
set_output_delay -max 3.0 -clock clk [get_pins {mem_req_rw_pad/I}]
set_output_delay -max 3.0 -clock clk [get_pins {mem_addr_pad*/I}]
set_output_delay -max 3.0 -clock clk [get_pins {mem_wdata_pad*/I}]

# Set minimum output delays (if needed)
set_output_delay -min 0.5 -clock clk [get_pins {cpu_resp_valid_pad/I}]
set_output_delay -min 0.5 -clock clk [get_pins {cpu_resp_data_pad*/I}]
set_output_delay -min 0.5 -clock clk [get_pins {mem_req_valid_pad/I}]
set_output_delay -min 0.5 -clock clk [get_pins {mem_req_rw_pad/I}]
set_output_delay -min 0.5 -clock clk [get_pins {mem_addr_pad*/I}]
set_output_delay -min 0.5 -clock clk [get_pins {mem_wdata_pad*/I}]

##############################################################
# Internal generated clocks (if any)
##############################################################

# No generated clocks in this design based on RTL

##############################################################
# Load capacitance assumptions
##############################################################

# Assuming typical load values (in pF)
set_load 0.05 [get_ports "cpu_resp_valid"]
set_load 0.05 [get_ports "cpu_resp_rdata[*]"]
set_load 0.05 [get_ports "mem_req_valid"]
set_load 0.05 [get_ports "mem_req_rw"]
set_load 0.05 [get_ports "mem_req_addr[*]"]
set_load 0.1  [get_ports "mem_req_wdata[*]"]

##############################################################
# Design-specific timing exceptions
##############################################################

# False paths between internal interfaces (if needed)
# Example: set_false_path -from [get_nets vc_probe_valid] -to [get_nets mem_req_valid]

##############################################################
# Multicycle paths (if any)
##############################################################

# If any paths require multiple cycles, specify them here
# Example: set_multicycle_path -setup 2 -from [get_pins ...] -to [get_pins ...]

##############################################################
# End of SDC
##############################################################
```


SYNOPSIS DESIGN CONSTRAINTS (.sdc) FILE

Outputs

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.18-s082_1
  Generated on:           Dec 28 2025  05:58:49 pm
  Module:                 top_wrapper
  Operating conditions:   NCCOM (balanced_tree)
  Wireload mode:          segmented
  Area mode:              timing library
============================================================

  Instance  Module  Cell Count   Cell Area  Net Area   Total Area      Wireload
-------------------------------------------------------------------------------
top_wrapper              13334 2300031.320     0.000  2300031.320 ZeroWireload (S)
  (S) = wireload was automatically selected
```

AREA REPORT

```
Instance: /top_wrapper
Power Unit: W
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
       memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     register     1.18440e-06  2.35909e-03  1.00220e-04  2.46049e-03   2.64%
        latch     2.86367e-08  8.30874e-06  1.24645e-06  9.58383e-06   0.01%
        logic     1.23980e-06  2.12337e-04  1.49069e-04  3.62646e-04   0.39%
         bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        clock     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pad     1.14923e-06  7.22068e-02  1.82172e-02  9.04251e-02  96.96%
           pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
     Subtotal     3.60207e-06  7.47866e-02  1.84677e-02  9.32579e-02 100.00%
   Percentage           0.00%       80.19%       19.80%      100.00% 100.00%
  ------------------------------------------------------------------------
```

POWER REPORT

```
============================================================
  Generated by:           Genus(TM) Synthesis Solution 21.18-s082_1
  Generated on:           Dec 28 2025  05:58:49 pm
  Module:                 top_wrapper
  Operating conditions:   NCCOM (balanced_tree)
  Wireload mode:          segmented
  Area mode:              timing library
============================================================


Path 1: MET (9358 ps) Setup Check with Pin victim_cache_core_VC_probe_line_reg_reg[127]/CP->D
          Group: clk
     Startpoint: (R) victim_cache_core_VC_fifo_age_reg[0][0]/CP
          Clock: (R) clk
       Endpoint: (R) victim_cache_core_VC_probe_line_reg_reg[127]/D
          Clock: (R) clk

                    Capture      Launch
       Clock Edge:+   20000           0
      Src Latency:+       0           0
      Net Latency:+       0 (I)       0 (I)
          Arrival:=   20000           0

              Setup:-       8
      Uncertainty:-     200
    Required Time:=   19792
     Launch Clock:-       0
        Data Path:-   10435
            Slack:=    9358
```

TIMING REPORT

Area Report:

The area report indicates that the synthesized **top_wrapper contains 13,334 standard cells** with a **total cell area of approximately 2.3 million square units**. This area reflects only the standard-cell logic (excluding routing), providing an early estimate of design size before physical implementation.

Power Report:

The power analysis shows a **total estimated power consumption of ~93.3 mW**, with the **I/O pads dominating power usage (~97%)**, while registers and logic contribute only a small fraction. This highlights that off-core interfaces are the primary power consumers in the current design configuration, which is typical for pad-heavy top-level wrappers.

Timing Report:

The timing analysis shows that the design **meets setup timing with a large positive slack of 9.358 ns**, indicating that the critical path between the FIFO age register and the probe line register comfortably satisfies the 20 ns clock period. This confirms that the synthesized L1 cache system operates reliably at the target clock frequency under typical PVT conditions.

SCHEMATICS GENERATED POSTSYNTHESIS

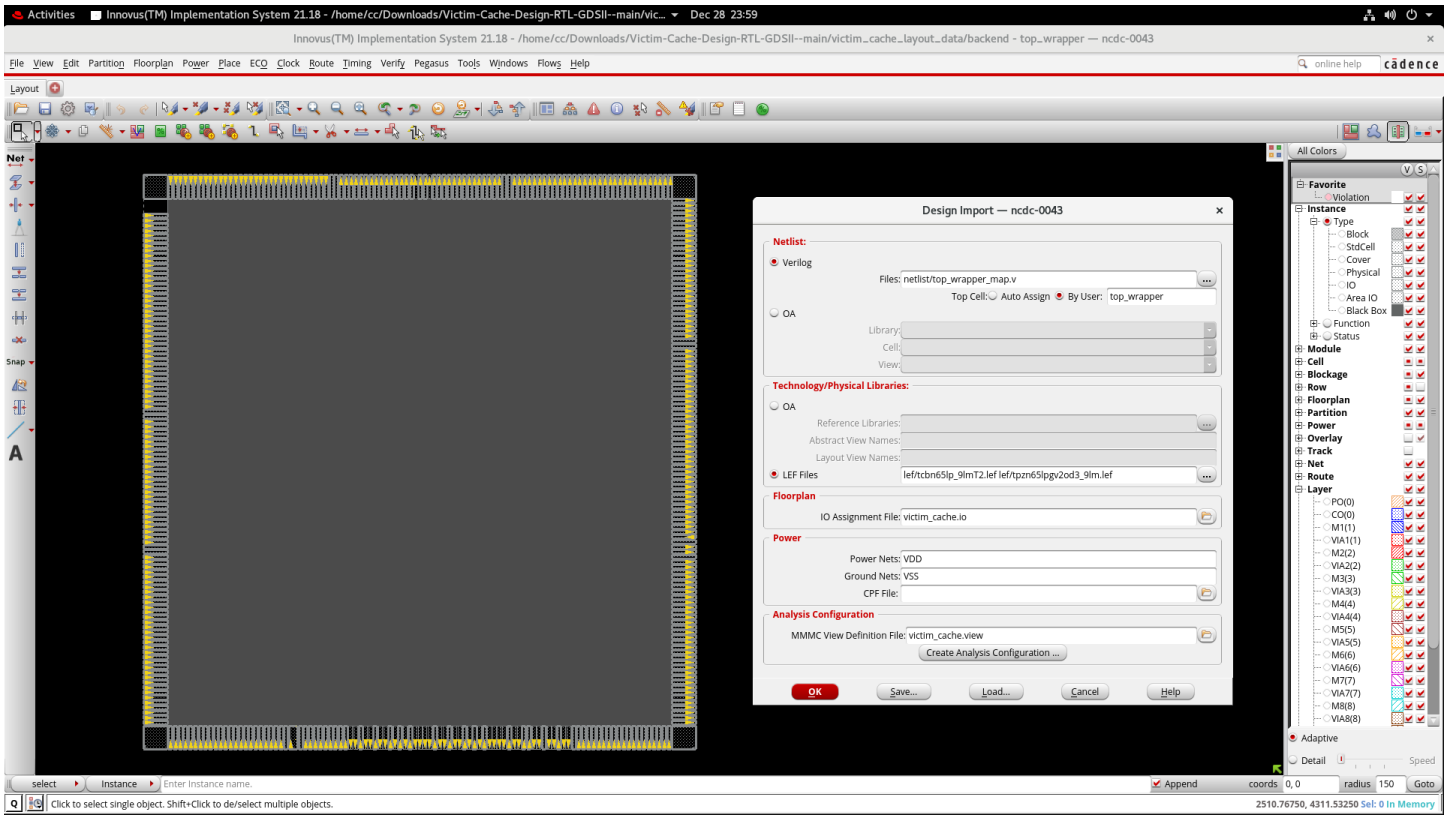
Physical Design (Back-End or Layout Design)

Physical Design is the back-end phase of the VLSI design flow in which the synthesized gate-level netlist is transformed into a **digital representation of the physical layout** of the chip. The netlist generated after synthesis serves as the primary input, and the standard cells are **placed and connected in software tools** while meeting timing, power, and area constraints.

Even though the process is done digitally, it produces a layout that specifies the **exact positions, routing, and geometries** for all cells and interconnects, which will later be used by the foundry to fabricate the actual silicon chip.

Physical design is performed to ensure that the logical design can be manufactured correctly and operates at the required speed. The goal is to produce a layout that satisfies timing closure, minimizes power consumption, adheres to fabrication rules, and enables reliable chip fabrication.

For Layout design or Physical Design, the Cadence's tool, Innovus was used. The design was imported into the tool



DESIGN IMPORT

Before starting the physical design flow in Cadence Innovus, several design, technology, and constraint files must be imported. Each file serves a specific role in enabling correct placement, routing, and timing analysis of the Victim Cache design.

`victim_cache.view` File (MMMC View Definition)

The `victim_cache.view` file is a **Multi-Mode Multi-Corner (MMMC) view definition file**. It specifies how different timing libraries, RC corners, and constraint files are grouped together into analysis views for setup and hold timing checks.

This file allows Innovus to perform timing analysis under multiple operating conditions by defining:

- Process corners (slow, typical, fast)
  RC corners for interconnect delay modeling
- Constraint views derived from SDC files

The `.view` file is essential for accurate timing closure, as it enables the tool to analyze worst-case and best-case timing scenarios across different modes of operation.

`tcbn65lp_9lmT2.lef & tbzn65lpgv2od3_9lm.lef` File (Standard Cell LEF)

The `tcbn65lp_9lmT2.lef(standard cells) & tbzn65lpgv2od3_9lm.lef(IO Pads)` file is the **technology LEF** provided by the foundry. They contain the **physical abstracts** of standard cells while used in the design, including:

- Cell dimensions
- Pin locations
- Metal layer information
- Routing blockages and design rules

Innovus uses this file to understand how standard cells are represented geometrically so that they can be correctly placed and routed during physical design.

`victim_cache.io` File (IO Placement File)

The `victim_cache.io` file defines the **logical-to-physical mapping of top-level ports** of the Victim Cache design. It specifies:

- IO pins / pads names
- Pins / Pads directions
- Placement of pins / pads along the chip boundary

This file ensures a structured and predictable IO layout, which is critical for signal integrity, routability, and integration with higher-level blocks.

Gate-Level Netlist (`top_wrapper_map.v`)

The gate-level netlist is generated after synthesis and represents the Victim Cache design using standard cells from the target library. It defines:

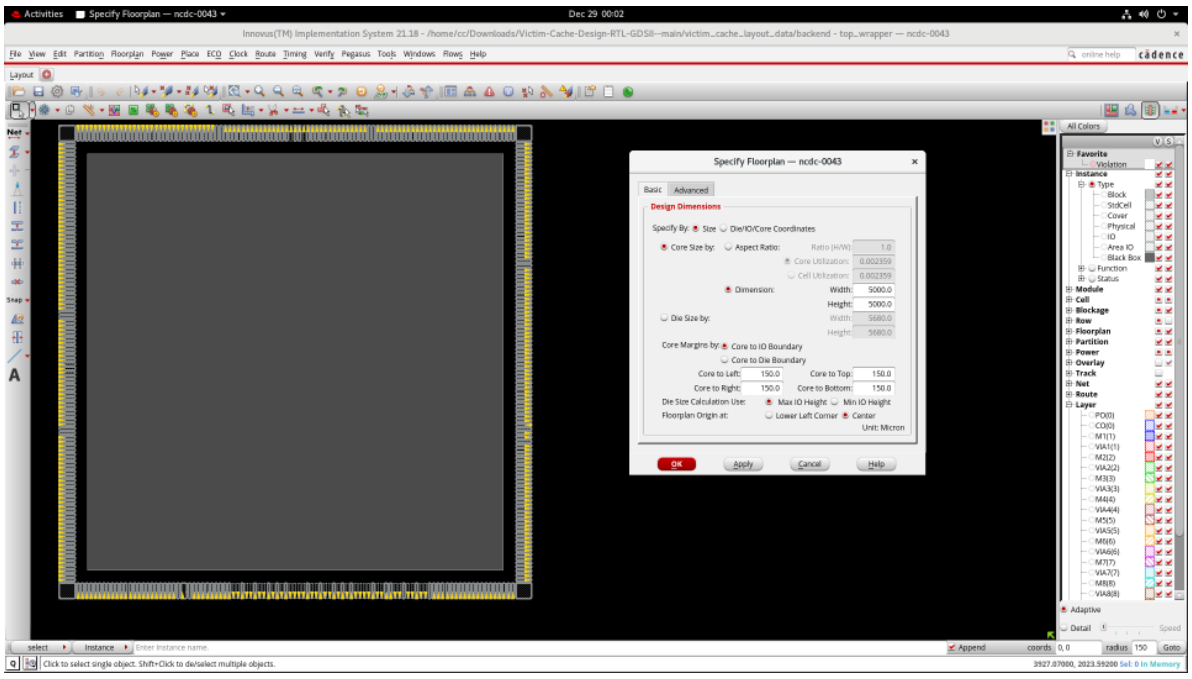
- Logical connectivity between gates
- Cell instances and their interconnections

This netlist is the **primary logical input** to Innovus and is used as the basis for placement, routing, and timing optimization.

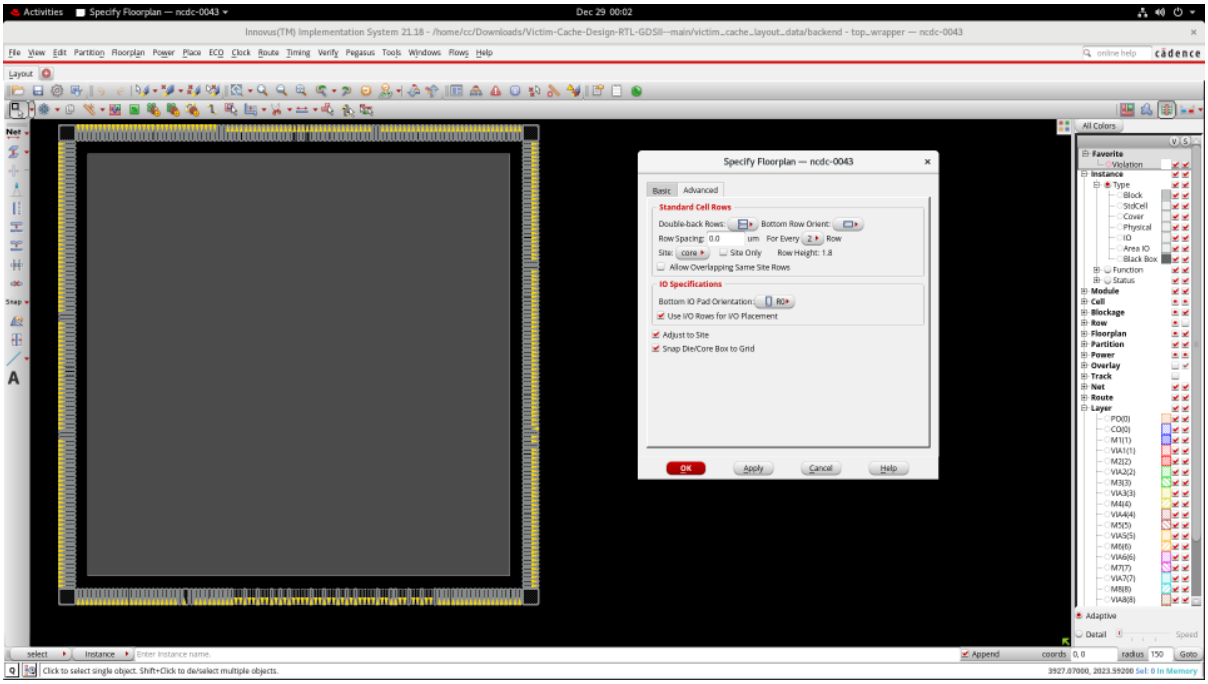
The key steps included in the Physical Design flow include

-
- Floorplanning
- Power planning
- Placement
- Clock Tree Synthesis (CTS)
- Routing
- Physical Verification.
- Fillers Placement
- Final DRC
- Exporting the GDS file

FLOORPLANNING



FLOORPLAN BASIC

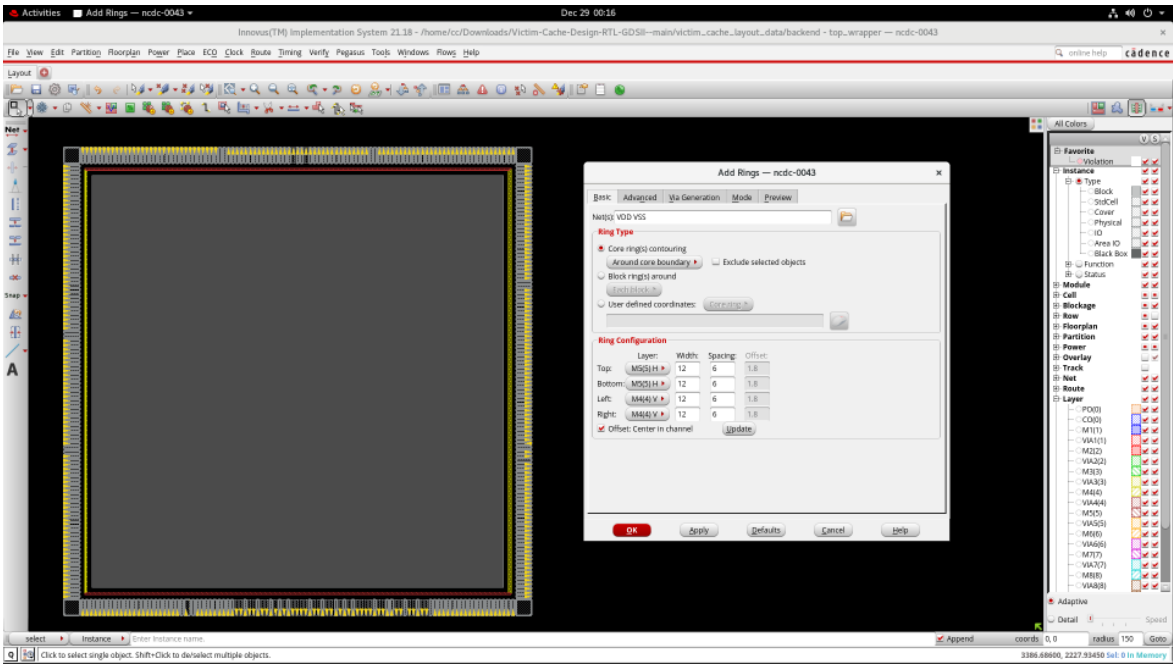


FLOORPLAN ADVANCED

The dimensions of core are adjusted with respect to the no of io pads such that the all the iopads fit correctly on the boundaries with enough space between them. Although the cell area was very low as compared to the core area but since the no of iopads is large, the design becomes pads limited. The width and height are 5000microns which makes the core squared while the core-to-io boundary is 150micron on all 4 sides for creating space for power rings.

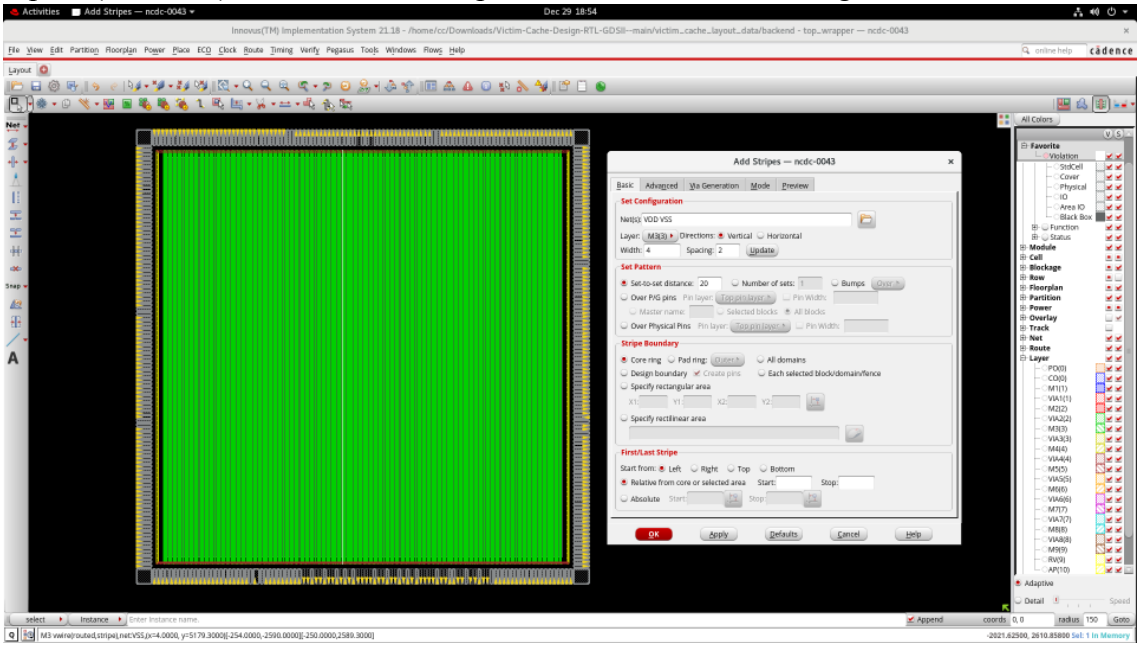
Use IO rows for IO placement is enabled, which **creates dedicated IO rows** around the boundary. **IO pads are placed only inside these IO rows**, not arbitrarily on the boundary.

POWER PLANNING

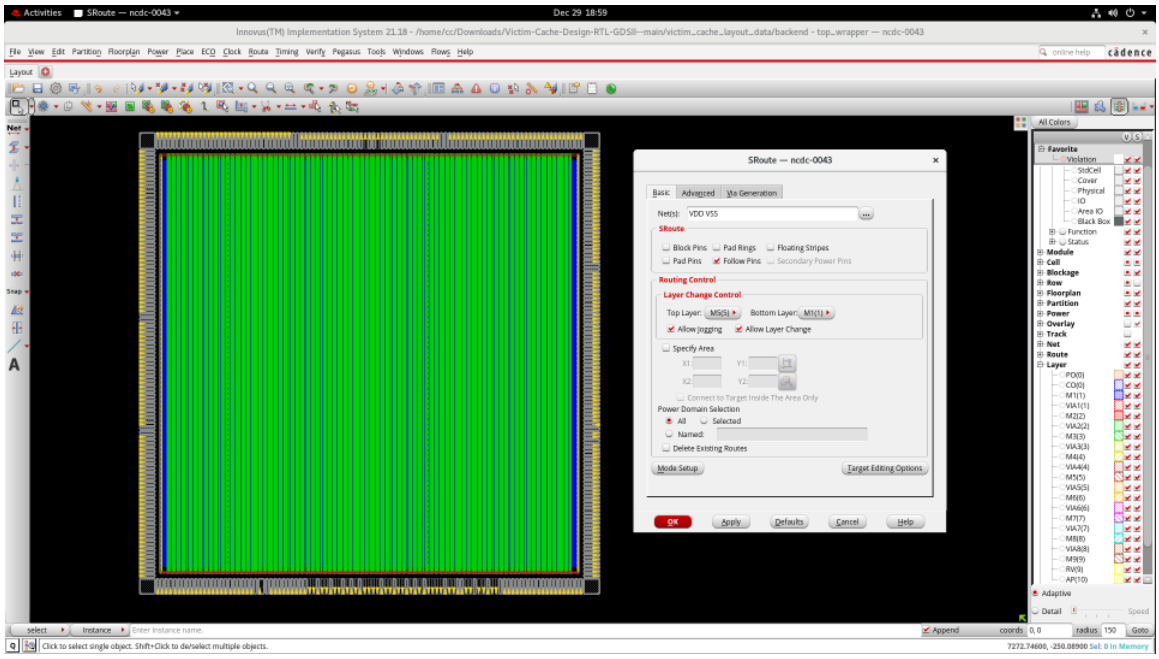


RINGS PLACEMENT

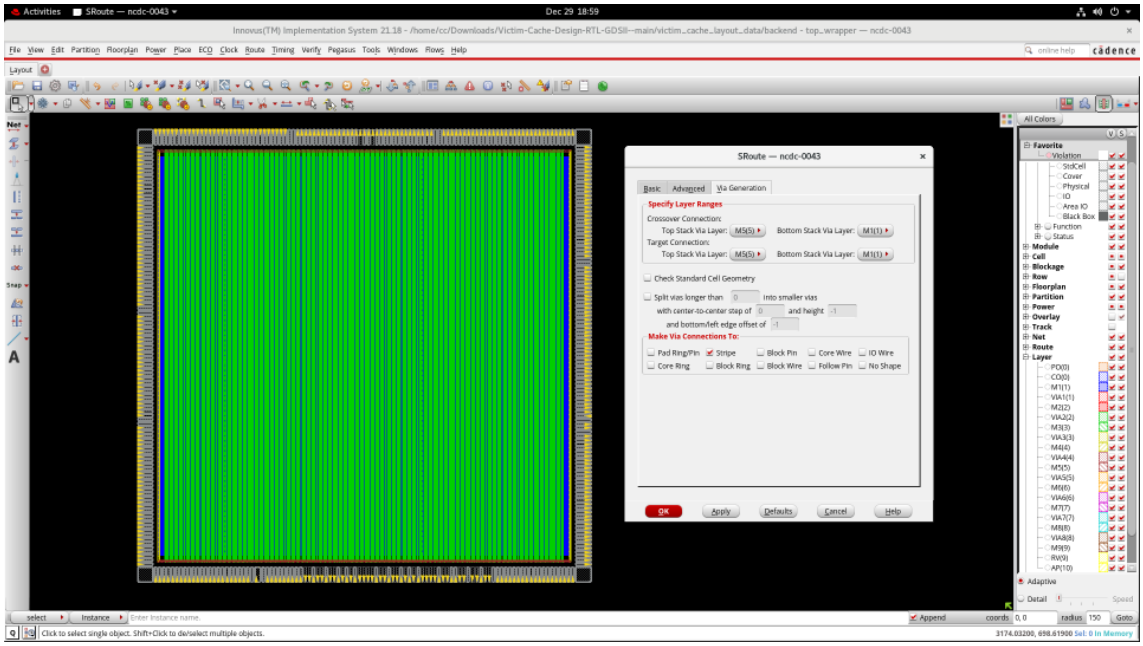
Rings are created on the **M5 and M4 metal layers** with the width of 12micron and spacing is 6 micron. The rings are kept in the middle or centre of the core and iopads. Since the core-io-boundary was 150micron, and the rings occupied **(12+12+6) = 30 micron**, the space on both sides of the rings is calculated to be **60 micron**.



STRIPES PLACEMENT



RAILS PLACEMENT (SROUTE BASIC)



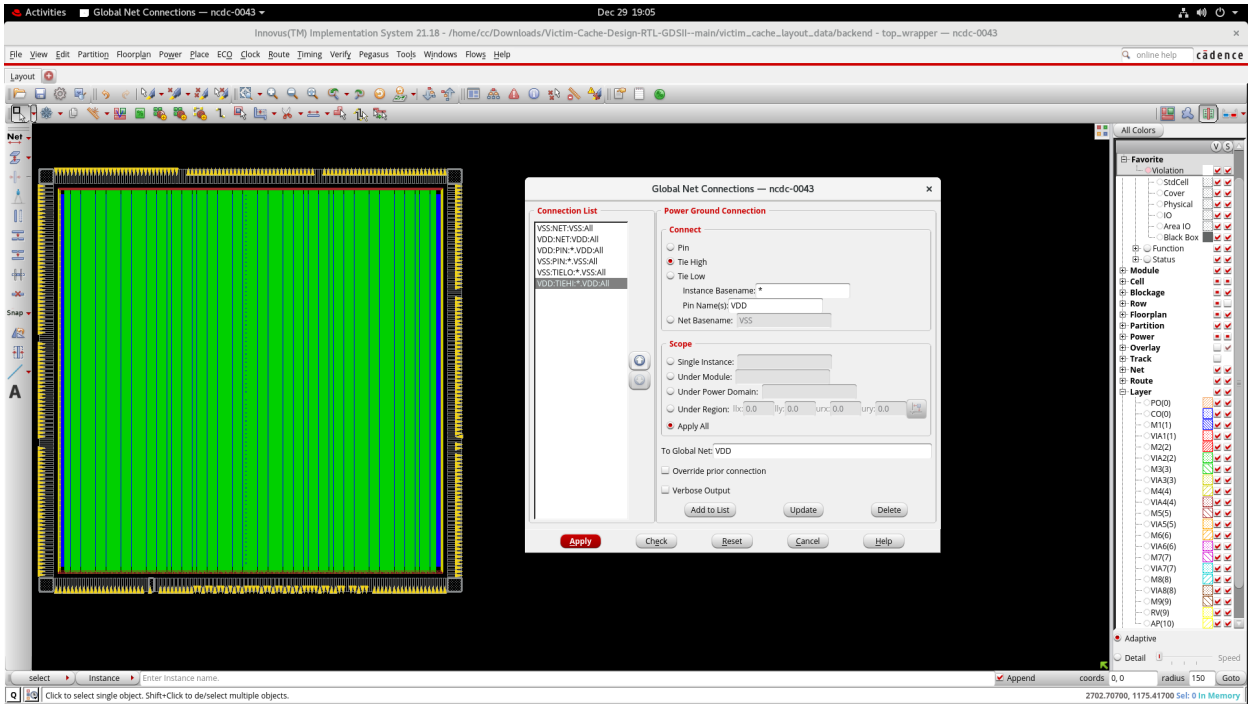
SROUTE VIA GEN

Power Stripes

- Power stripes are placed first to distribute VDD and VSS uniformly across the core area.
- They provide low-resistance, high-capacity paths for current flow from the power rings into the interior of the design.
- Stripes reduce IR drop and electromigration by spreading current over wider metal tracks.
- They act as intermediate power highways between the global power rings and local standard-cell rails.

Power Rails

- Power rails are embedded within standard-cell rows and directly supply power to individual cells.
- Rails connect the cell pins to the nearby power stripes, ensuring every cell receives stable VDD and VSS.
- Together with stripes, rails complete the power delivery network from the top-level supply to the smallest logic elements.
- During rail placement, **via generation from Metal-5 down to Metal-1** ensures a continuous vertical power path from global rings → stripes → standard-cell rails.
- **Follow pins** is enabled which ensures rails align precisely with standard-cell power pins, guaranteeing correct VDD/VSS connections.
- **Allow metal change** is enabled which allows switching between metal layers when needed, ensuring uninterrupted power routing despite obstacles.
-



GLOBAL NET CONNECTIONS

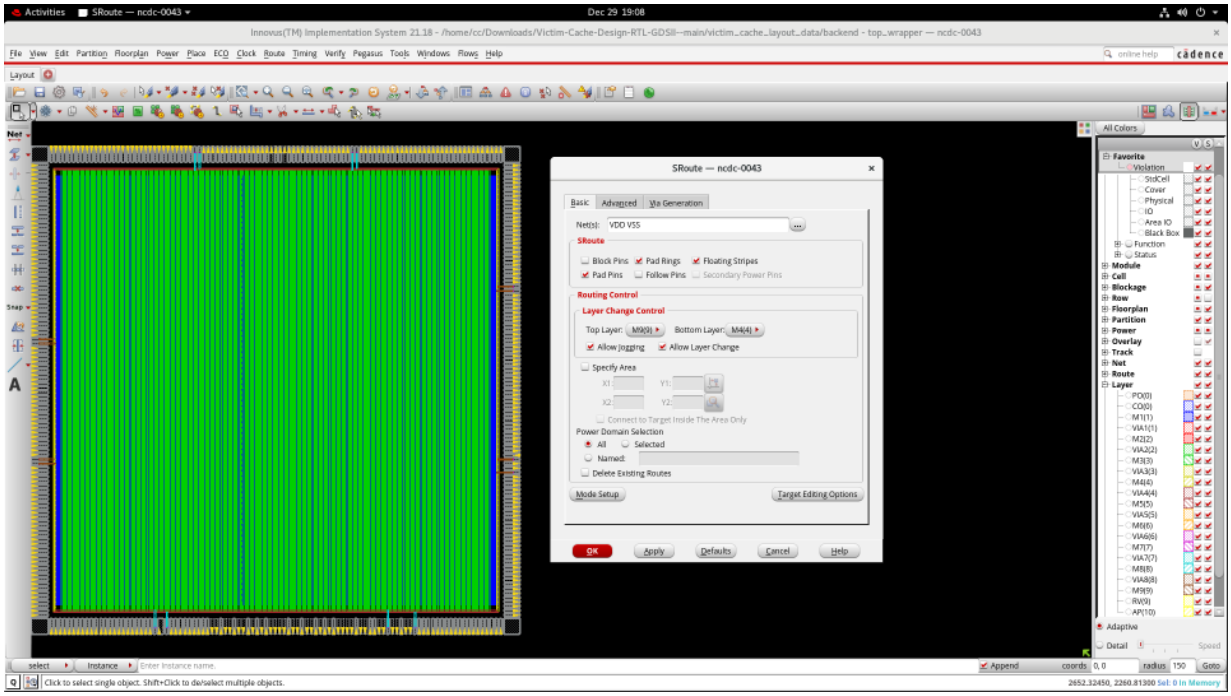
- Global nets are special signals such as **VDD, VSS (GND)** that must be available throughout the entire design.
- During physical design, these nets are explicitly declared as *global* so that the tool can automatically connect them to all relevant pins without requiring manual routing.
- Global net connection ensures that every standard cell, macro, and IO that requires power, ground, or control signals is correctly tied to the appropriate network.
- This step avoids floating power pins or unconnected control pins and guarantees consistency between the logical netlist and the physical layout.

Purpose of TIEHI and TIELO Cells

- **TIEHI cells** generate a constant logic '1' by tying their output to VDD.
- **TIELO cells** generate a constant logic '0' by tying their output to VSS (ground).

Examples:

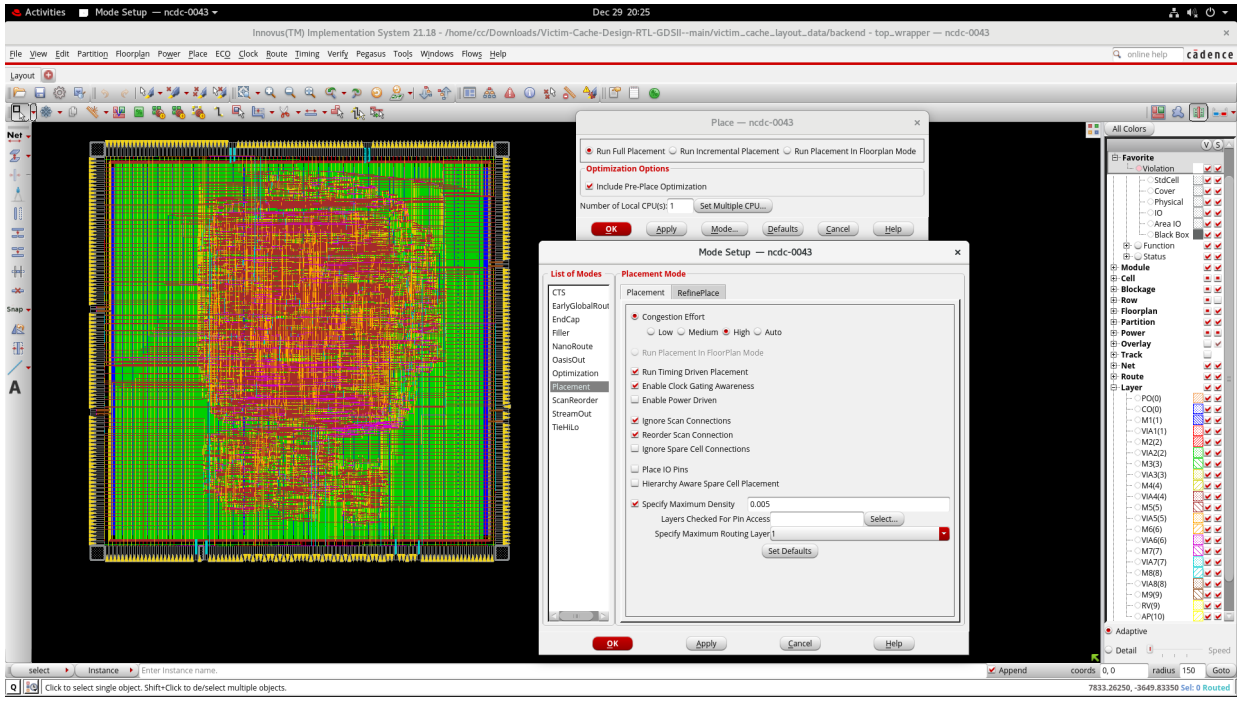
- `VSS : TIELO*.VSS : ALL`
  → Connect ground (VSS) to all TIELO cells.
- `VDD : *.VDD : ALL`
  → Connect VDD to **all** cells' VDD pins.
- `VSS : *.VSS : ALL`
  → Connect ground to **all** cells' VSS pins.



SROUTE FOR PADS CONNECTIONS

- The second SRoute is mainly to **connect global nets to I/O pads and finalize global connectivity** across the chip.
- Pad rings and I/O pins are connected on the **top-most metal layer** to avoid conflicts with internal routing.
- By selecting M9 as the top layer during the second SRoute, the tool can **route pad connections directly to the global nets** and tie them to stripes or rails below.

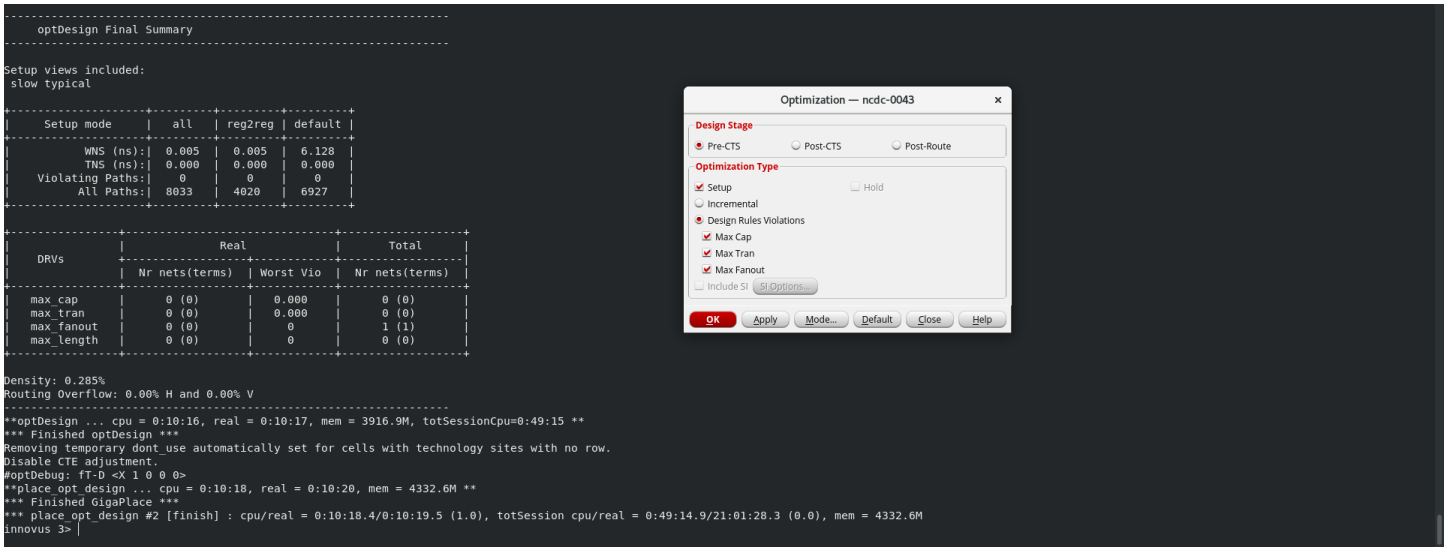
PLACEMENT



PLACEMENT

During placement, the density was kept 0.005 (5%) as the **design was pad-limited.** This leaves extra whitespace around cells, providing flexibility for routing. In pad-limited designs, a low density prevents cells from being too tightly packed, which is crucial because routing channels near the pads are limited and congested. This extra margin reduces the risk of DRC violations during routing and maintains uniformity.

CLOCK TREE SYNTHESIS (CTS)

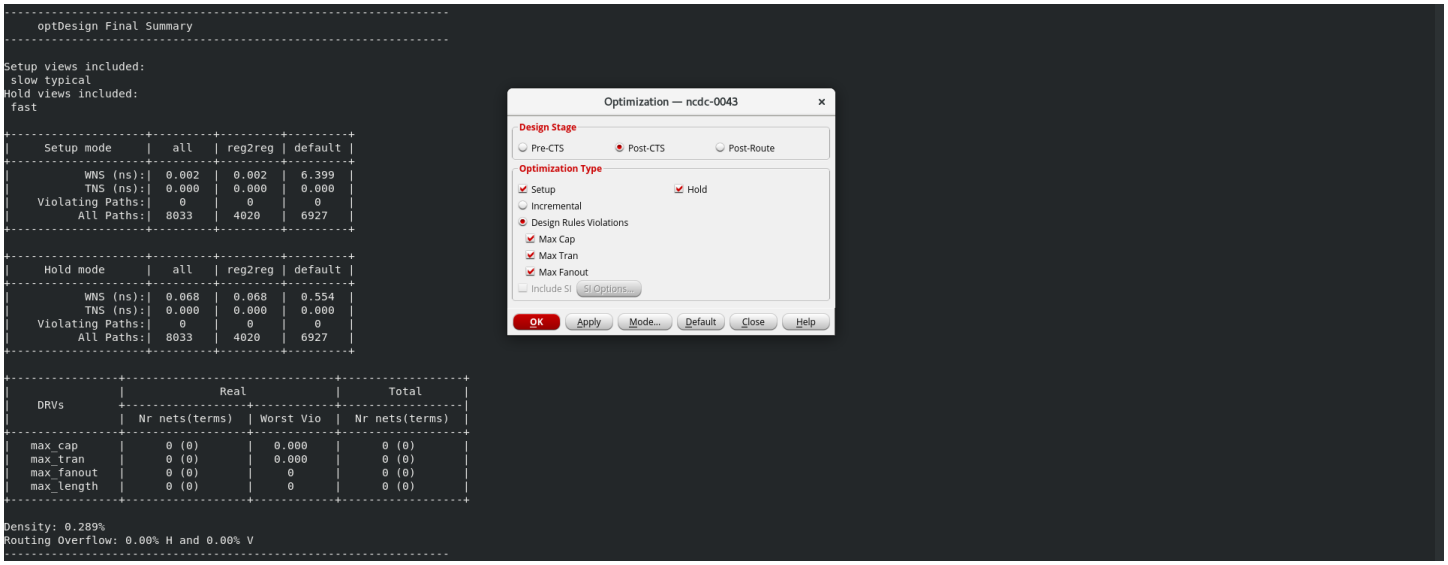


```
------------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------------

Setup views included:
 slow typical

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.005  |  0.005  |  6.128  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|  8033   |  4020   |  6927   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------+
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      1 (1)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 0.285%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------------
**optDesign ... cpu = 0:10:16, real = 0:10:17, mem = 3916.9M, totSessionCpu=0:49:15 **
*** Finished optDesign ***
Removing temporary dont_use automatically set for cells with technology sites with no row.
Disable CTE adjustment.
#optDebug: fT-D <X 1 0 0 0>
**place_opt_design ... cpu = 0:10:18, real = 0:10:20, mem = 4332.6M **
*** Finished GigaPlace ***
*** place_opt_design #2 [finish] : cpu/real = 0:10:18.4/0:10:19.5 (1.0), totSession cpu/real = 0:49:14.9/21:01:28.3 (0.0), mem = 4332.6M
innovus 3>
```


PRE-CTS OPTIMIZATION WITH ZERO SETUP VIOLATIONS

```
[NR-eGR]             Length (um)    Vias
[NR-eGR] ------------------------------------
[NR-eGR]  M1  (1H)             0   74454
[NR-eGR]  M2  (2V)        820280   89556
[NR-eGR]  M3  (3H)         51130   58064
[NR-eGR]  M4  (4V)       1391601   68506
[NR-eGR]  M5  (5H)       2115559    8234
[NR-eGR]  M6  (6V)        104672    2343
[NR-eGR]  M7  (7H)         44039     592
[NR-eGR]  M8  (8V)         36987     290
[NR-eGR]  M9  (9H)         24811       0
[NR-eGR]  AP  (10V)            0       0
[NR-eGR] ------------------------------------
[NR-eGR]      Total      4589079  302039
[NR-eGR] ------------------------------------------------------------------------------
[NR-eGR] Total half perimeter of net bounding box: 4083404um
[NR-eGR] Total length: 4589079um, number of vias: 302039
[NR-eGR] ------------------------------------------------------------------------------
[NR-eGR] Total eGR-routed clock nets wire length: 108932um, number of vias: 21212
[NR-eGR] ------------------------------------------------------------------------------
Early Global Route wiring runtime: 4.61 seconds, mem = 5050.4M
0 delay mode for cte disabled.
```

CLOCK TREE SYNTHESIS RESULTS



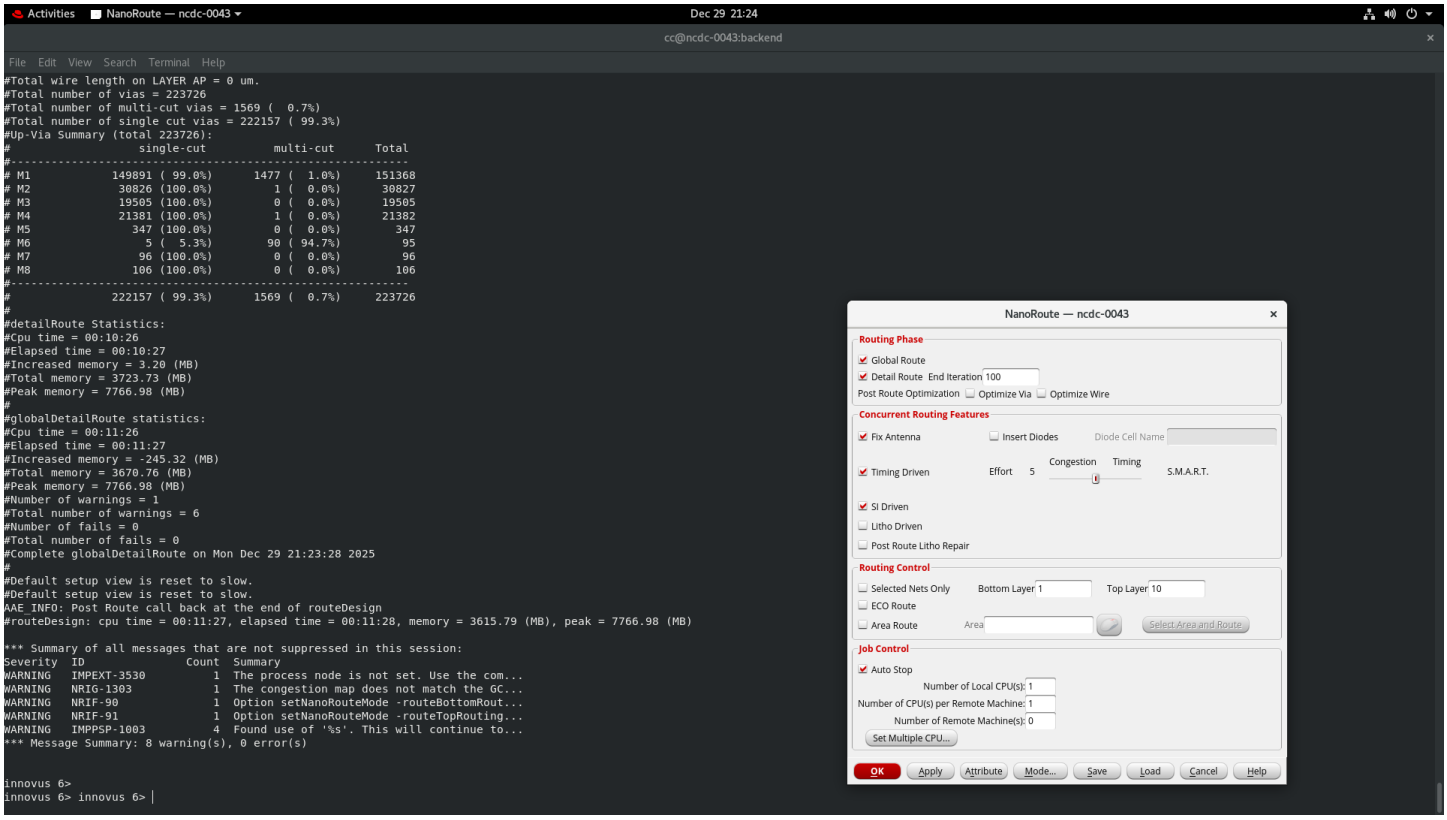
```
------------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------------

Setup views included:
 slow typical
Hold views included:
 fast

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.002  |  0.002  |  6.399  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|  8033   |  4020   |  6927   |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.068  |  0.068  |  0.554  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|  8033   |  4020   |  6927   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------+
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 0.289%
Routing Overflow: 0.00% H and 0.00% V
------------------------------------------------------------------
```


POST-CTS OPTIMIZATION WITH ZERO SETUP AND HOLD VIOLATIONS

- **Pre-CTS Optimization:** Performed **before clock tree synthesis**, focusing on logic restructuring, buffer insertion, and gate sizing to improve timing, reduce congestion, and prepare the design for an efficient clock tree. Setup and hold violations are **estimated based on the original netlist timing** but not final, since the clock network isn't balanced yet.
- **CTS (Clock Tree Synthesis):** The process of **building the clock distribution network**. Buffers and inverters are inserted to balance clock arrival times at all sequential elements, minimizing clock skew. After CTS, **setup and hold violations are checked accurately**, because the actual clock arrival times at flip-flops are known.
- **Post-CTS Optimization:** Done **after the clock tree is inserted**, addressing any timing violations caused by CTS. Includes re-buffering, resizing, or tweaking paths to fix **setup violations** (paths too slow) and **hold violations** (paths too fast), while preserving the clock network integrity.

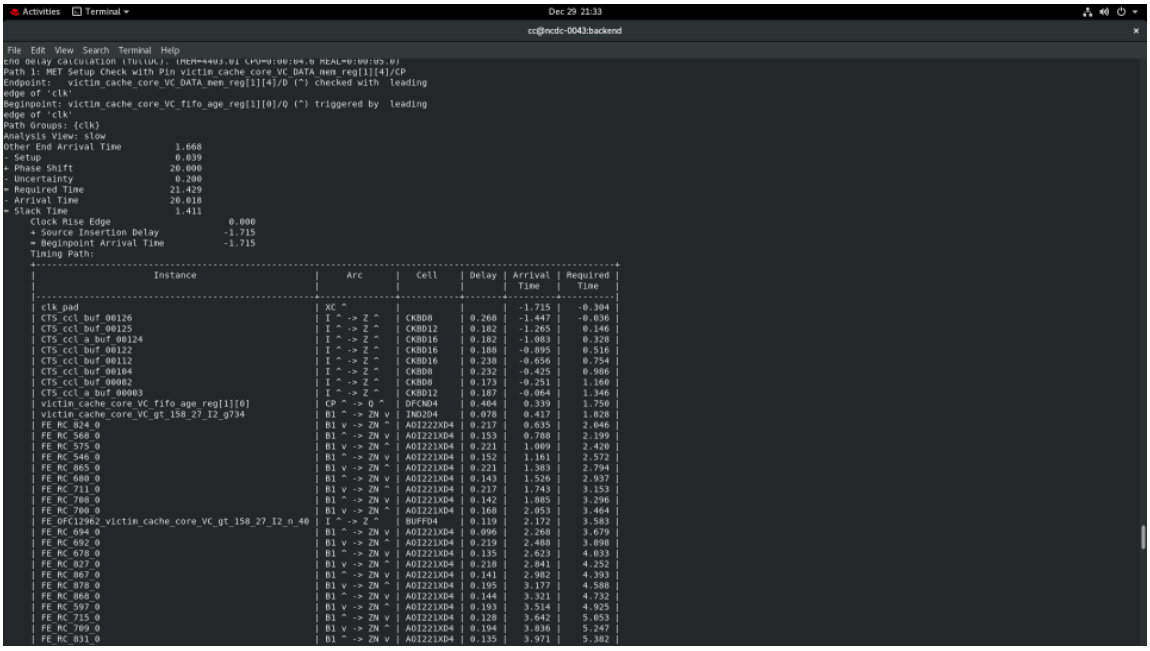
So in short: **Setup violations** are critical for slow paths, and **hold violations** are critical for fast paths; both are actively checked after CTS and addressed during post-CTS optimization.

ROUTING



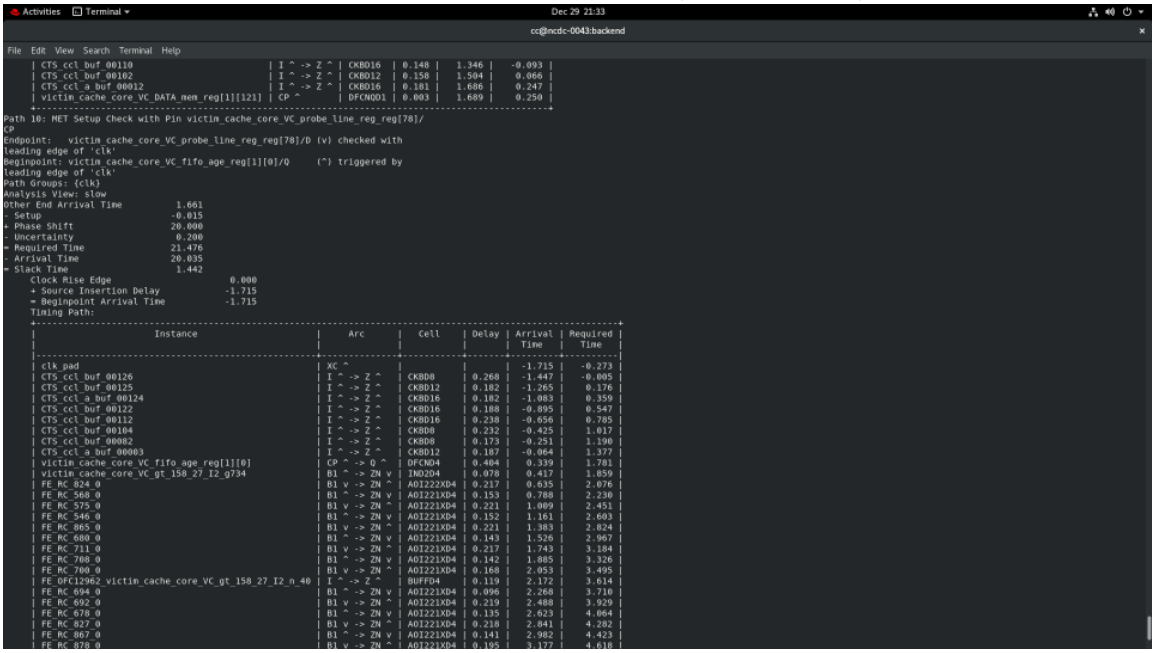
NANO ROUTE WITH RESULTS

NanoRoute finalizes routing by balancing timing (Timing Driven enabled), signal integrity (SI Driven enabled which avoids issues such as crosstalk and noise), and design rule correctness, with antenna fixes (Fix Antenna enabled) applied during this stage. **Iterations are set to 100** which means that the tool is allowed up to 100 iterations to optimize the routes, though it typically converges earlier, so the maximum is rarely reached.



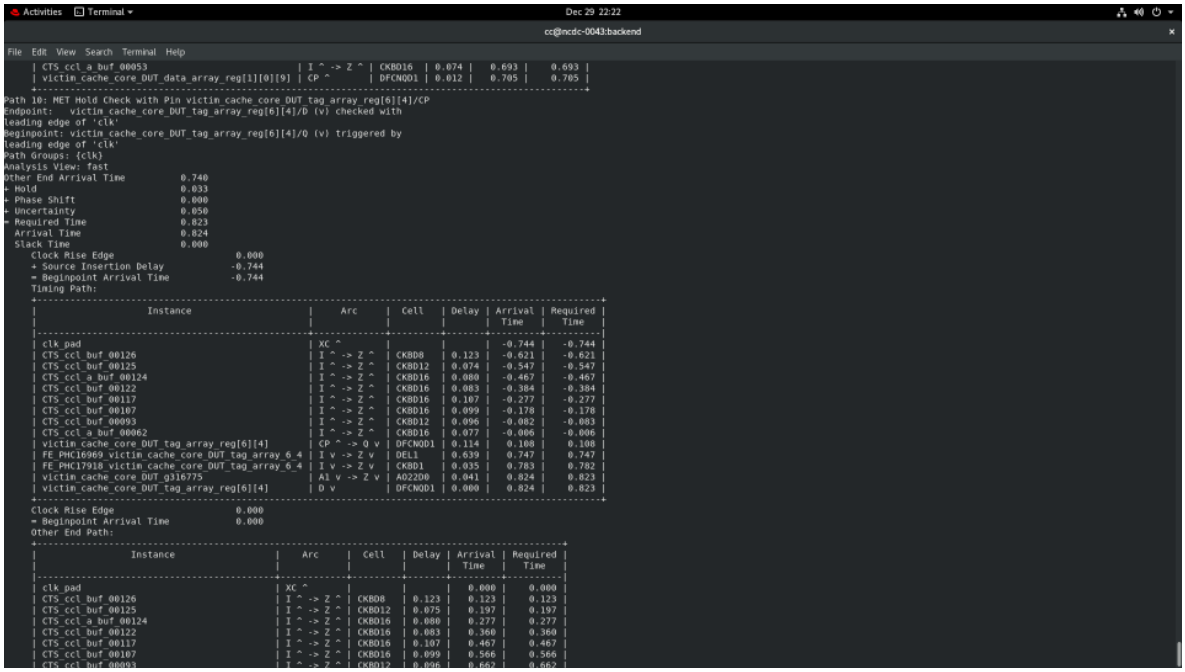
```
Activities  Terminal                                   Dec 29 21:33
                                                       cc@ncdc-0043:backend
File  Edit  View  Search  Terminal  Help
End delay calculation (fullDC): (MEM=4403.01 CPU=0:00:04.6 REAL=0:00:05.0)
Path 1: MET Setup Check with Pin victim_cache_core_VC_DATA_mem_reg[1][4]/CP
Endpoint:   victim_cache_core_VC_DATA_mem_reg[1][4]/D (^) checked with  leading
edge of 'clk'
Beginpoint: victim_cache_core_VC_fifo_age_reg[1][0]/Q (^) triggered by  leading
edge of 'clk'
Path Groups: {clk}
Analysis View: slow
Other End Arrival Time          1.668
- Setup                         0.039
+ Phase Shift                  20.000
- Uncertainty                   0.200
= Required Time                21.429
- Arrival Time                 20.018
= Slack Time                    1.411
     Clock Rise Edge                     0.000
     + Source Insertion Delay           -1.715
     = Beginpoint Arrival Time          -1.715
     Timing Path:
     +-------------------------------------------------------------------------------------------------------------+
     |                    Instance                    |     Arc      |   Cell   | Delay | Arrival | Required |
     |                                                |              |          |       |  Time   |   Time   |
     |------------------------------------------------+--------------+----------+-------+---------+----------|
     | clk_pad                                        | XC ^         |          |       |  -1.715 |   -0.304 |
     | CTS_ccl_buf_00126                              | I ^ -> Z ^   | CKBD8    | 0.268 |  -1.447 |   -0.036 |
     | CTS_ccl_buf_00125                              | I ^ -> Z ^   | CKBD12   | 0.182 |  -1.265 |    0.146 |
     | CTS_ccl_a_buf_00124                            | I ^ -> Z ^   | CKBD16   | 0.182 |  -1.083 |    0.328 |
     | CTS_ccl_buf_00122                              | I ^ -> Z ^   | CKBD16   | 0.188 |  -0.895 |    0.516 |
     | CTS_ccl_buf_00112                              | I ^ -> Z ^   | CKBD16   | 0.238 |  -0.656 |    0.754 |
     | CTS_ccl_buf_00104                              | I ^ -> Z ^   | CKBD8    | 0.232 |  -0.425 |    0.986 |
     | CTS_ccl_buf_00082                              | I ^ -> Z ^   | CKBD8    | 0.173 |  -0.251 |    1.160 |
     | CTS_ccl_a_buf_00003                            | I ^ -> Z ^   | CKBD12   | 0.187 |  -0.064 |    1.346 |
     | victim_cache_core_VC_fifo_age_reg[1][0]        | CP ^ -> Q ^  | DFCND4   | 0.404 |   0.339 |    1.750 |
     | victim_cache_core_VC_gt_158_27_I2_g734         | B1 ^ -> ZN v | IND2D4   | 0.078 |   0.417 |    1.828 |
     | FE_RC_824_0                                    | B1 v -> ZN ^ | AOI222XD4| 0.217 |   0.635 |    2.046 |
     | FE_RC_568_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.153 |   0.788 |    2.199 |
     | FE_RC_575_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.221 |   1.009 |    2.420 |
     | FE_RC_546_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.152 |   1.161 |    2.572 |
     | FE_RC_865_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.221 |   1.383 |    2.794 |
     | FE_RC_680_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.143 |   1.526 |    2.937 |
     | FE_RC_711_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.217 |   1.743 |    3.153 |
     | FE_RC_708_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.142 |   1.885 |    3.296 |
     | FE_RC_700_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.168 |   2.053 |    3.464 |
     | FE_OFC12962_victim_cache_core_VC_gt_158_27_I2_n_40 | I ^ -> Z ^ | BUFFD4   | 0.119 |   2.172 |    3.583 |
     | FE_RC_694_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.096 |   2.268 |    3.679 |
     | FE_RC_692_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.219 |   2.488 |    3.898 |
     | FE_RC_678_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.135 |   2.623 |    4.033 |
     | FE_RC_827_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.218 |   2.841 |    4.252 |
     | FE_RC_867_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.141 |   2.982 |    4.393 |
     | FE_RC_878_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.195 |   3.177 |    4.588 |
     | FE_RC_868_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.144 |   3.321 |    4.732 |
     | FE_RC_597_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.193 |   3.514 |    4.925 |
     | FE_RC_715_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.128 |   3.642 |    5.053 |
     | FE_RC_709_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.194 |   3.836 |    5.247 |
     | FE_RC_831_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.135 |   3.971 |    5.382 |
```

POST-ROUTE SETUP CHECK (PATH 01 MET)



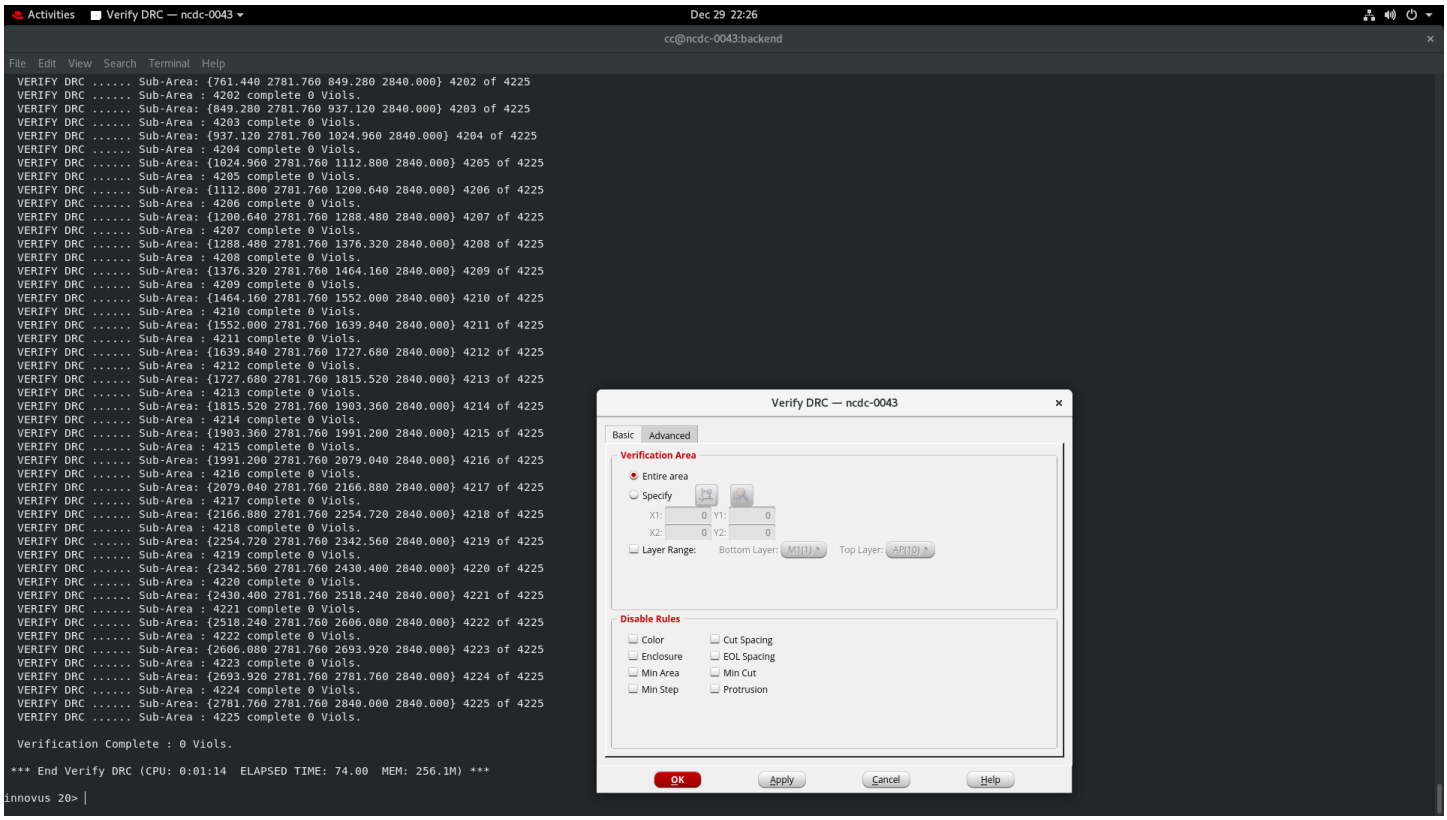
```
Activities  Terminal                                   Dec 29 21:33
                                                       cc@ncdc-0043:backend
File  Edit  View  Search  Terminal  Help
     | CTS_ccl_buf_00110                              | I ^ -> Z ^ | CKBD16   | 0.148 |   1.346 |   -0.093 |
     | CTS_ccl_buf_00102                              | I ^ -> Z ^ | CKBD12   | 0.158 |   1.504 |    0.066 |
     | CTS_ccl_a_buf_00012                            | I ^ -> Z ^ | CKBD16   | 0.181 |   1.686 |    0.247 |
     | victim_cache_core_VC_DATA_mem_reg[1][121]      | CP ^       | DFCND1   | 0.003 |   1.689 |    0.250 |
     +-------------------------------------------------------------------------------------------------------------+
Path 10: MET Setup Check with Pin victim_cache_core_VC_probe_line_reg_reg[78]/
CP
Endpoint:   victim_cache_core_VC_probe_line_reg_reg[78]/D (v) checked with
leading edge of 'clk'
Beginpoint: victim_cache_core_VC_fifo_age_reg[1][0]/Q     (^) triggered by
leading edge of 'clk'
Path Groups: {clk}
Analysis View: slow
Other End Arrival Time          1.661
- Setup                        -0.015
+ Phase Shift                  20.000
- Uncertainty                   0.200
= Required Time                21.476
- Arrival Time                 20.035
= Slack Time                    1.442
     Clock Rise Edge                     0.000
     + Source Insertion Delay           -1.715
     = Beginpoint Arrival Time          -1.715
     Timing Path:
     +-------------------------------------------------------------------------------------------------------------+
     |                    Instance                    |     Arc      |   Cell   | Delay | Arrival | Required |
     |                                                |              |          |       |  Time   |   Time   |
     |------------------------------------------------+--------------+----------+-------+---------+----------|
     | clk_pad                                        | XC ^         |          |       |  -1.715 |   -0.273 |
     | CTS_ccl_buf_00126                              | I ^ -> Z ^   | CKBD8    | 0.268 |  -1.447 |   -0.005 |
     | CTS_ccl_buf_00125                              | I ^ -> Z ^   | CKBD12   | 0.182 |  -1.265 |    0.176 |
     | CTS_ccl_a_buf_00124                            | I ^ -> Z ^   | CKBD16   | 0.182 |  -1.083 |    0.359 |
     | CTS_ccl_buf_00122                              | I ^ -> Z ^   | CKBD16   | 0.188 |  -0.895 |    0.547 |
     | CTS_ccl_buf_00112                              | I ^ -> Z ^   | CKBD16   | 0.238 |  -0.656 |    0.785 |
     | CTS_ccl_buf_00104                              | I ^ -> Z ^   | CKBD8    | 0.232 |  -0.425 |    1.017 |
     | CTS_ccl_buf_00082                              | I ^ -> Z ^   | CKBD8    | 0.173 |  -0.251 |    1.190 |
     | CTS_ccl_a_buf_00003                            | I ^ -> Z ^   | CKBD12   | 0.187 |  -0.064 |    1.377 |
     | victim_cache_core_VC_fifo_age_reg[1][0]        | CP ^ -> Q ^  | DFCND4   | 0.404 |   0.339 |    1.781 |
     | victim_cache_core_VC_gt_158_27_I2_g734         | B1 ^ -> ZN v | IND2D4   | 0.078 |   0.417 |    1.859 |
     | FE_RC_824_0                                    | B1 v -> ZN ^ | AOI222XD4| 0.217 |   0.635 |    2.076 |
     | FE_RC_568_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.153 |   0.788 |    2.230 |
     | FE_RC_575_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.221 |   1.009 |    2.451 |
     | FE_RC_546_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.152 |   1.161 |    2.603 |
     | FE_RC_865_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.221 |   1.383 |    2.824 |
     | FE_RC_680_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.143 |   1.526 |    2.967 |
     | FE_RC_711_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.217 |   1.743 |    3.184 |
     | FE_RC_708_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.142 |   1.885 |    3.326 |
     | FE_RC_700_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.168 |   2.053 |    3.495 |
     | FE_OFC12962_victim_cache_core_VC_gt_158_27_I2_n_40 | I ^ -> Z ^ | BUFFD4   | 0.119 |   2.172 |    3.614 |
     | FE_RC_694_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.096 |   2.268 |    3.710 |
     | FE_RC_692_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.219 |   2.488 |    3.929 |
     | FE_RC_678_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.135 |   2.623 |    4.064 |
     | FE_RC_827_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.218 |   2.841 |    4.282 |
     | FE_RC_867_0                                    | B1 ^ -> ZN v | AOI221XD4| 0.141 |   2.982 |    4.423 |
     | FE_RC_878_0                                    | B1 v -> ZN ^ | AOI221XD4| 0.195 |   3.177 |    4.618 |
```

POST-ROUTE SETUP CHECK (PATH 10 MET)

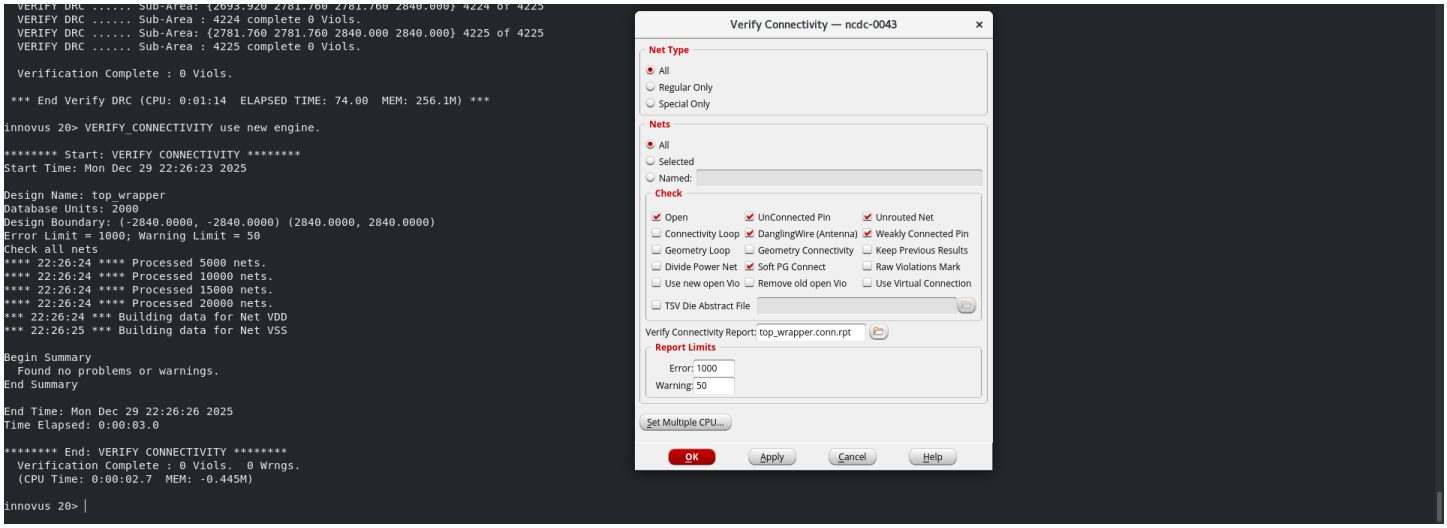


POST ROUTE HOLD CHECK (MET)

PHYSICAL VERIFICATION



DESIGN RULES CHECK (DRC VERIFICATION) (0 VIOLATIONS REPORTED)

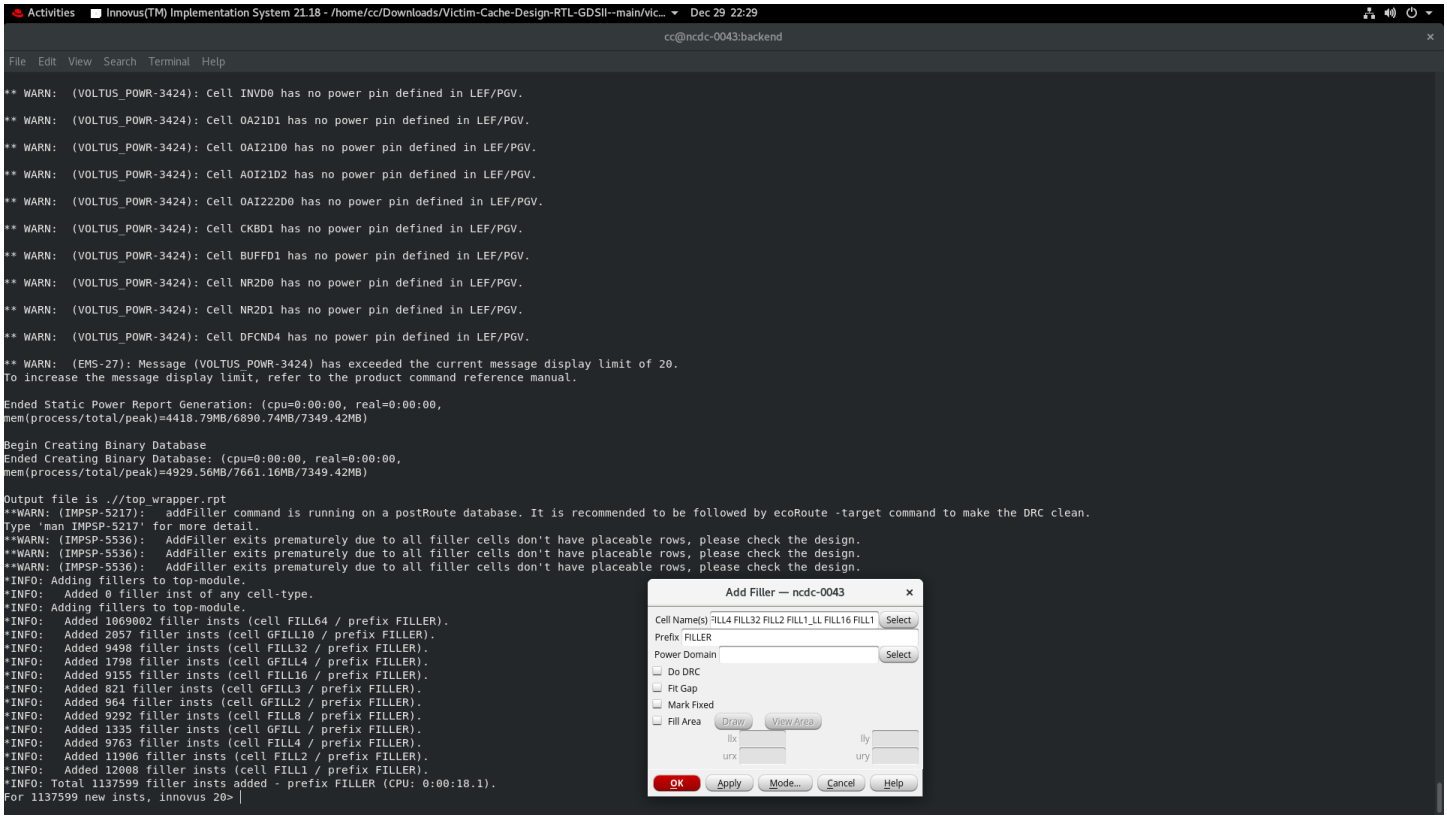


CONNECTIVITY VERIFICATION (LVS 0 VIOLATIONS REPORTED)

DRC (Design Rule Check) Verification:
Ensures that the layout **complies with all fabrication rules** specified by the foundry, such as minimum widths, spacing, via rules, and metal density. Any violations could cause manufacturing defects, so DRC is essential before tape-out.

Connectivity Verification:
Confirms that the **electrical connections in the layout match the intended netlist**, ensuring that all pins, vias, and nets are correctly connected. This prevents open circuits, shorts, or misconnected signals that could cause functional failures.

FILLERS PLACEMENT



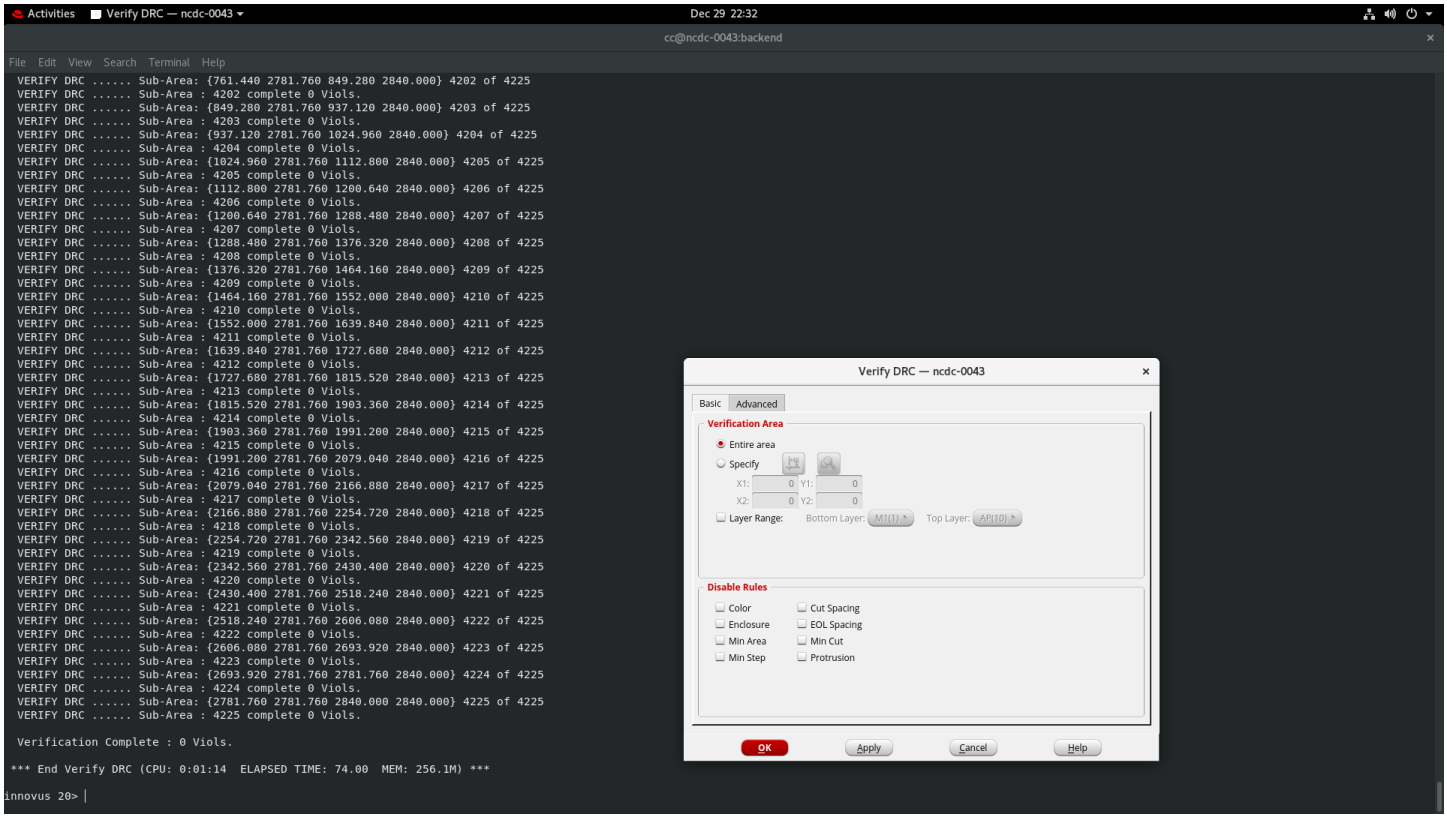
FILLER CELLS PLACEMENT

Filler cells are small standard-cell-like structures inserted in a chip layout to **fill empty spaces between functional cells**. Their primary purposes are:

- **Maintain continuity of power and ground rails:** Ensures that VDD and VSS lines are continuous across the design, preventing floating metal segments.
- **Improve manufacturability:** Helps meet metal density rules required by the foundry to avoid issues like dishing or erosion during chemical-mechanical polishing (CMP).
- **Enable predictable parasitics:** By filling gaps uniformly, they help in maintaining consistent capacitance and resistance, reducing timing variations.

In short, filler cells **do not perform any logic function** but are essential for reliable and manufacturable layouts.

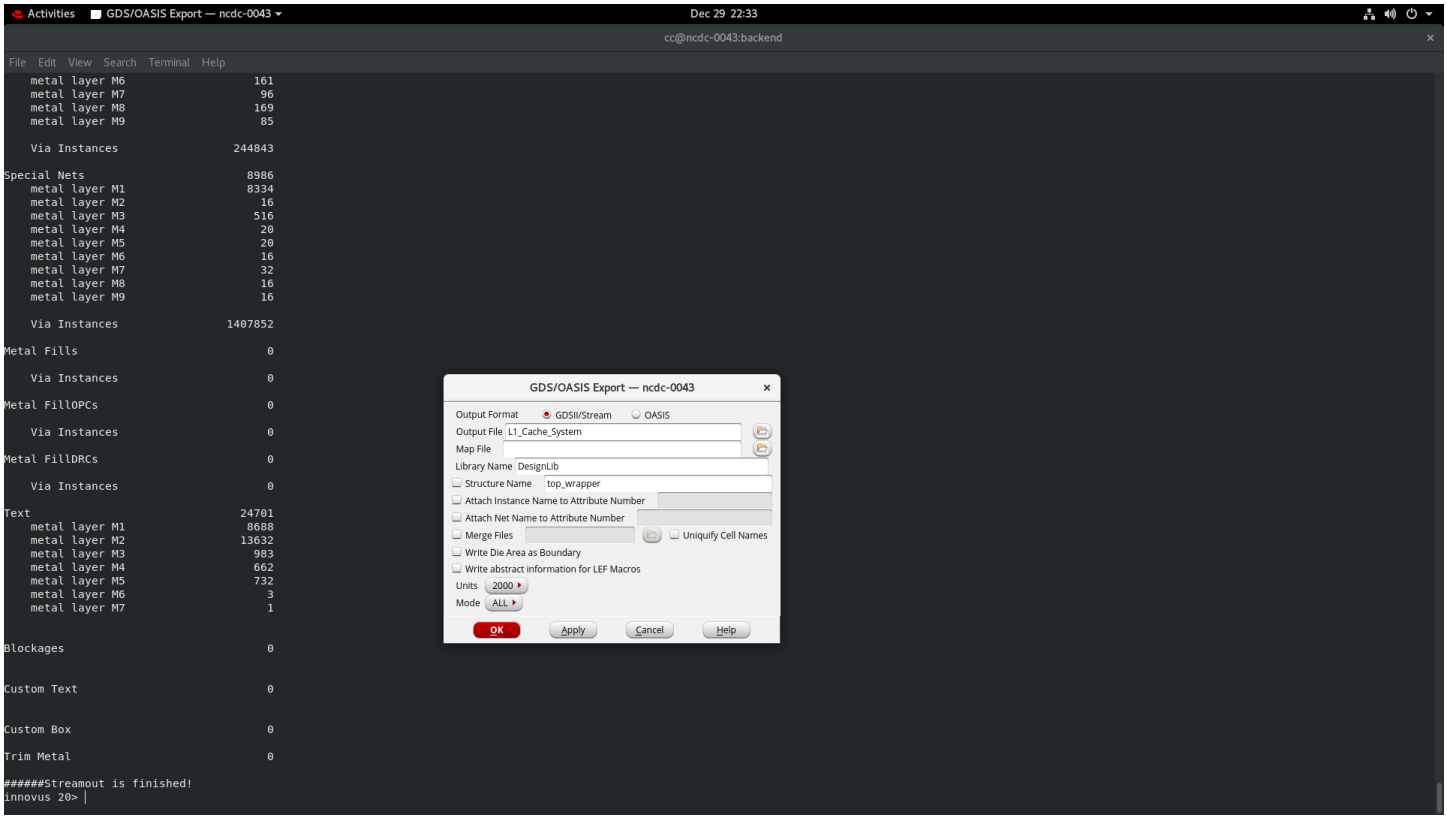
FINAL DRC



FINAL DRC VERIFICATION

DRC is performed again after filler cell placement to ensure that the newly inserted cells do not **violate any design rules**, such as spacing, width, or metal density. It also **verifies that power/ground continuity and layout integrity** are maintained across the design.

EXPORTING THE GDS FILE



GDS FILE EXPORTED

Final Layout