INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, DESIGN AND MANUFACTURING, KURNOOL

COURSE: VLSI SYSTEM DESIGN (EC-307)

Design and Implementation of Cache Memory using Semi-Custom Design Flow in 90nm CMOS Technology

Submitted by:

R. Venkata Dharun Reddy (Roll No: 123EC0038)
P. Rishi Kiran (Roll No: 123EC0054)

Under the Guidance of:

Dr. P. Ranga Babu

Department of Electronics and Communication Engineering
IIITDM Kurnool

Academic Year: 2025–2026

Contents

Abstract

This project documents the design, verification, synthesis and semi-custom physical implementation of a cache memory module targeted to a 90 nm CMOS standard-cell flow. A direct-mapped cache with write-through and write-allocate behaviour was described in Verilog and functionally verified in Vivado. The design was synthesized using Cadence Genus and the physical design (place, route, and verification) was performed in Cadence Innovus. This report details the theory behind cache operation and mapping, the semi-custom implementation decisions, full RTL and testbench listings, the tool flow and scripts used, and a thorough analysis of area, power and timing — both pre- and post-layout. Post-layout extraction and verification results are included and interpreted to demonstrate timing closure and physical correctness.

1. Introduction

1.1. Motivation and Scope

Modern processors rely heavily on small, fast memories (caches) to hide the latency of main memory and to deliver the high bandwidth demanded by CPUs and accelerators. The objective of this project is to design a small direct-mapped cache, verify it at RTL, synthesize it to a 90 nm standard cell library, and complete a semi-custom physical implementation with post-layout verification. This provides hands-on experience of the full RTL-to-GDS flow and the trade-offs between area, power and timing.

1.2. Project Overview

This project includes:

- Verilog RTL for a direct-mapped cache module (parameters: cache size, block size, address/data widths).
- A self-checking testbench to verify basic read/write behaviour and hit/miss logic.
- Functional simulation in Vivado with schematic captures and waveforms.
- Synthesis in Cadence Genus and physical implementation in Cadence Innovus (90 nm standard cell library).
- Extraction of area, power, and timing reports and post-layout verification (DRC/LVS/Connectivity).

1.3. Conventions and Notation

Throughout this report:

- All timings are in nanoseconds (ns) unless explicitly stated.
- Area is reported in square micrometers (μm^2).
- Power is expressed in Watts (W) or microwatts (μW).
- Figures and tables are referenced by number.

2. Theory and Working Principle

2.1. Cache Fundamentals

Caches are small, fast memories positioned between the CPU and slower main memory. Their goal is to exploit temporal and spatial locality: data recently accessed or near recently accessed addresses is likely to be used again soon. A cache reduces average memory access time (AMAT) and reduces pressure on the memory bus.

2.1.1. Key Terms

Block / Line: The smallest unit transferred between cache and main memory (Block/Line size).

Tag: High-order bits of the address used to identify whether a block stored in a cache line corresponds to the requested address.

Index: Bits of the address that select the cache set or the specific line (in direct mapped, index selects the unique line).

Offset: Bits that select a byte/word inside the block.

Hit: Requested data exists in cache.

Miss: Requested data not found and must be fetched from lower memory (incurs miss penalty).

2.1.2. Performance Metrics

- **Hit Rate (HR)** = $\frac{\text{Number of hits}}{\text{Total accesses}}$. Higher is better.
- **Miss Rate (MR)** = $1 - HR$.
- **Miss Penalty:** Extra time to handle a miss (fetch block from memory).
- **AMAT:** AMAT = Hit time + Miss rate × Miss penalty.

2.2. Cache Mapping Techniques

2.2.1. Direct-Mapped Cache

Each memory block maps to exactly one cache line (index). Advantages: simple, fast lookup (single comparator), minimal hardware. Disadvantages: high conflict misses when two frequently used addresses map to same line.

2.2.2. Fully Associative Cache

Any block can be placed in any line. Requires associative search (parallel tag comparators), expensive in hardware, useful for small caches.

2.2.3. Set-Associative Cache

Compromise: cache is divided into sets and each set contains multiple ways (lines). A block maps to a set but can occupy any way in that set. Replacement policy (LRU, FIFO, Random) resolves which way to evict.

2.2.4. Choice for This Project

We implemented a **direct-mapped cache** (simpler control, lower area and faster single-cycle tag comparison). This choice is typical for teaching and for small caches in embedded designs, and it keeps the semi-custom implementation tractable while allowing us to concentrate on physical design aspects.

2.3. Cache Read/Write Operation

2.3.1. Read (Load)

1. Index bits select the cache line.
2. Tag comparator checks if stored tag matches requested tag and the valid bit is set.
3. If a hit: return the word selected by the offset.
4. If a miss: fetch block from main memory, update tag and valid bit (write-allocate), then return data.

2.3.2. Write (Store)

Two common policies:

Write-Through On write, update both cache and main memory immediately. Simpler but increases memory traffic.

Write-Back On write, update only the cache line and set a dirty bit; write to main memory later when the line is evicted. Reduces memory traffic but requires dirty bits and writeback logic.

2.3.3. Policy Implemented

This design implements **write-through** with **write-allocate** (on write miss, we allocate the block into cache then write). This simplifies coherence and is straightforward for an academic semi-custom flow.

2.4. Replacement and Coherence

In a direct-mapped cache, the replacement decision is implicit (single possible line per index). Coherence and multi-processor considerations are out of scope for this single-module implementation but would require protocols such as MESI in multi-core systems.

2.5. Semi-Custom Design Flow and Relevance of 90 nm

2.5.1. Semi-Custom Flow

Semi-custom ASIC design relies on standard cell libraries and automatic place-and-route; designers write RTL, synthesize to gates, and use PR tools to generate layouts. This flow balances performance, manufacturability and design turnaround time, making it an excellent academic platform.

2.5.2. Why 90 nm?

- **Pedagogical balance:** 90 nm is small enough to demonstrate realistic parasitic effects and power scaling, yet it's mature and available in standard academic PDKs.
- **Trade-offs:** Compared to older nodes (180 nm), 90 nm shows reduced area and dynamic power; compared to bleeding-edge nodes, it's simpler to work with for fast project turnaround.
- **Physical effects:** Routing parasitics, buffer insertion, and power distribution are still meaningful at 90 nm, enabling realistic post-layout analysis.

3. Design and Implementation

3.1. High Level Architecture

The cache module implemented is parameterized by cache size, block size, address width and data width (see RTL). It contains:

- Tag array and valid bits (for each cache line).
- Data array storing block words for each line.
- Tag comparator and control logic for read/write hit detection.
- Simple write-through datapath (write to cache and assume memory updated).

3.1.1. Address Partitioning

Given:

$$\text{ADDR\_WIDTH} = 32; \quad \text{BLOCK\_SIZE} = 16 \text{ bytes}$$

Offset bits $= \log_2(\text{BLOCK\_SIZE}) = 4$. Index bits $= \log_2(\text{NUM\_BLOCKS})$ where $\text{NUM\_BLOCKS} = \frac{\text{CACHE\_SIZE}}{\text{BLOCK\_SIZE}}$. Remaining upper bits are the tag.

3.1.2. Data Word Indexing

Since data width is 32 bits (4 bytes), a word offset inside a block is obtained from the low offset bits (example: offset[3:2] gives which 4-byte word in the 16-byte block).

3.2. RTL Listing and Explanation

The complete Verilog module used in this project is listed and explained below. The code is parameterized and straightforward so it can be synthesized and mapped to a standard-cell library without any vendor specific constructs.

3.2.1. Verilog Module (cache_memory.v)

Listing 3.1: cache_memory.v — parameterized direct-mapped cache

```
`timescale 1ns / 1ps

module cache_memory #(
    parameter CACHE_SIZE = 4096,
    parameter BLOCK_SIZE = 16,
    parameter ADDR_WIDTH = 32,
    parameter DATA_WIDTH = 32
)(
    input  wire clk,
    input  wire rst,
    input  wire [ADDR_WIDTH-1:0] addr,
    input  wire [DATA_WIDTH-1:0] write_data,
    input  wire mem_read,
    input  wire mem_write,
    output reg  [DATA_WIDTH-1:0] read_data,
    output reg  hit
);

    localparam NUM_BLOCKS  = CACHE_SIZE / BLOCK_SIZE;
    localparam INDEX_BITS  = $clog2(NUM_BLOCKS);
    localparam OFFSET_BITS = $clog2(BLOCK_SIZE);
    localparam TAG_BITS    = ADDR_WIDTH - INDEX_BITS -
       OFFSET_BITS;

    wire [TAG_BITS-1:0]   tag     = addr[ADDR_WIDTH-1 :
       ADDR_WIDTH - TAG_BITS];
    wire [INDEX_BITS-1:0] index   = addr[OFFSET_BITS +:
       INDEX_BITS];
    wire [OFFSET_BITS-1:0] offset = addr[OFFSET_BITS-1:0];
    wire [1:0] word_offset = offset[3:2]; // 4 bytes per word

    // Cache arrays
    reg [TAG_BITS-1:0] tag_array [0:NUM_BLOCKS-1];
    reg valid_array [0:NUM_BLOCKS-1];
    reg [DATA_WIDTH-1:0] data_array [0:NUM_BLOCKS-1][0:(
       BLOCK_SIZE/DATA_WIDTH)-1];

    integer i;

    // Reset and operation
```

```
37      always @(posedge clk or posedge rst) begin
38          if (rst) begin
39              for (i = 0; i < NUM_BLOCKS; i = i + 1) begin
40                  valid_array[i] <= 1'b0;
41                  tag_array[i]   <= {TAG_BITS{1'b0}};
42              end
43              hit <= 0;
44              read_data <= 0;
45          end else begin
46              hit <= 0;
47
48              // Write-through + write-allocate
49              if (mem_write) begin
50                  valid_array[index] <= 1'b1;
51                  tag_array[index]   <= tag;
52                  data_array[index][word_offset] <= write_data;
53                  hit <= valid_array[index] && (tag_array[index]
                        == tag);
54              end
55
56              // Read
57              else if (mem_read) begin
58                  if (valid_array[index] && (tag_array[index] ==
                        tag)) begin
59                      hit <= 1;
60                      read_data <= data_array[index][word_offset];
61                  end else begin
62                      hit <= 0;
63                      read_data <= 32'hDEADBEEF;
64                  end
65              end
66          end
67      end
68  endmodule
```

3.2.2. Design Rationale (Theory embedded)

- **Parameterization:** Allows easy scaling of cache size and block size for experiments.
- **Direct-mapped choice:** Simplifies tag storage and reduces the number of com-

parators, lowering area — critical for semi-custom flows.

- **Write policy:** Write-through with write-allocate reduces state complexity (no dirty bit required), but increases memory traffic — acceptable for this academic demonstrator.
- **Data array modelling:** Implemented as a 2D reg array — synthesizes to RAM inference or register array depending on synthesis tool and library support. For small caches, registers are acceptable; for larger caches, inferred RAM would be area efficient.

3.3. Testbench and Simulation Strategy

3.3.1. Testbench (tb_cache_memory.v)

Listing 3.2: tb_cache_memory.v — self checking testbench

```
1  `timescale 1ns / 1ps
2
3  module tb_cache_memory;
4      reg clk;
5      reg rst;
6      reg [31:0] addr;
7      reg [31:0] write_data;
8      reg mem_read;
9      reg mem_write;
10     wire [31:0] read_data;
11     wire hit;
12
13     cache_memory uut (
14         .clk(clk), .rst(rst),
15         .addr(addr), .write_data(write_data),
16         .mem_read(mem_read), .mem_write(mem_write),
17         .read_data(read_data), .hit(hit)
18     );
19
20     initial clk = 0;
21     always #5 clk = ~clk; // 100 MHz
22
23     initial begin
24         $display("CACHE␣MEMORY␣TEST␣START");
25         rst = 1; addr = 0; write_data = 0; mem_read = 0;
               mem_write = 0;
```

```
26          #20; rst = 0;
27
28          // Write to address 0x40 (miss then allocate)
29          #10; addr = 32'h0000_0040; write_data = 32'hAABB_CCDD;
               mem_write = 1; #10 mem_write = 0; #10;
30          $display("[%0t] WRITE @0x%h Data=0x%h Hit=%b",$time,addr
               ,write_data,hit);
31
32          // Read same address (should hit)
33          #10; addr = 32'h0000_0040; mem_read = 1; #10 mem_read =
               0; #10;
34          $display("[%0t] READ  @0x%h Data=0x%h Hit=%b",$time,addr
               ,read_data,hit);
35
36          // Read different address (should miss)
37          #10; addr = 32'h0000_1040; mem_read = 1; #10 mem_read =
               0; #10;
38          $display("[%0t] READ  @0x%h Data=0x%h Hit=%b",$time,addr
               ,read_data,hit);
39
40          #50; $finish;
41      end
42
43      initial begin
44          $monitor("[%0t] CLK=%b RST=%b RD=%b WR=%b ADDR=0x%h OUT
               =0x%h HIT=%b",
45                  $time, clk, rst, mem_read, mem_write, addr,
                       read_data, hit);
46      end
47  endmodule
```

3.3.2. Simulation Observations (Theory embedded)

- **Stimulus coverage:** The testbench demonstrates write allocate and read hit/miss behavior. For more exhaustive verification a sequence generator and randomized access patterns (plus assertion checks) are recommended.
- **Expected waveform behaviour:** On first write to an index (cold cache), valid is set and tag stored. Subsequent read must return stored word. On an address mapping to a different index or different tag, read should return the sentinel value (or trigger a miss-handling routine in a fuller design).

4. Tool Flow

4.1. Overview

The flow used in this project is:

RTL (Vivado simulation) → Synthesis (Cadence Genus) → Place & Route (Cadence Innovus) → Post-

This semi-custom flow is widely used in industry and maps well to university lab environments.

4.2. Vivado — RTL Simulation and Schematic Capture

4.2.1. Purpose

Vivado is used to verify correctness of the RTL before synthesis. Key outputs were:

- RTL simulation waveform (signal transitions for read/write/hit).
- Schematic views (module hierarchy, net connectivity) useful for functional inspection and as documentation.

4.2.2. Files and Artifacts

- ‘vivado simulation.png‘ — main waveform showing read/write/hit transitions.
- ‘s1.png‘–‘s8.png‘ — schematic views (top module, memory arrays, control logic, etc.).

4.2.3. Interpretation and Theory

Schematic inspection confirms the correctness of the address partition, and waveform demonstrates timing relationships between clocks, resets, and control signals. Vivado waveforms are the first point to catch logical errors such as incorrect indexing or missized arrays before costly synthesis runs.

4.3. Cadence Genus — Synthesis

4.3.1. Synthesis Goals

- Map RTL to a gate-level netlist using the 90 nm standard cell library.
- Meet the target clock (100 MHz in this design) with positive slack.
- Generate area, power and timing reports for analysis.

4.3.2. Synthesis Constraints and Scripts

Typical SDC settings used:

```
create_clock -name clk -period 10 [get_ports clk]
set_input_delay -clock clk [get_ports addr] 1
set_multicycle_path ...
```

The synthesis TCL invoked ‘genus‘ scripts to perform optimization and generate ‘.rpt‘ files for area, power and timing.

4.3.3. Synthesis Observations (Theory embedded)

- **Register-heavy power:** The power report shows registers contribute a large fraction of leakage and internal power (expected in register-rich designs).
- **Area composition:** Synthesis cell count and area are useful baselines to understand how much silicon the RTL occupies before routing overhead.
- **Timing slack:** Positive worst slack indicates synthesis achieved the timing goal at gate-level without routing parasitics.

4.4. Cadence Innovus — Physical Implementation

4.4.1. Flow Steps

1. Import netlist, constraints and library into Innovus.
2. Floorplan with core utilization, power rails and IO planning.
3. Placement of standard cells.
4. Clock Tree Synthesis (CTS).
5. Routing (global and detail).
6. Post-route optimization (buffer insertion, upsizing).

7. Extract parasitics (SPEF) and run post-layout timing and power analyses.
8. Run DRC/LVS and connectivity checks.

4.4.2. Artifacts

- ‘layout.png‘ — final routed layout screenshot.
- ‘no layer.png‘ — pre-routing or layer display (for illustration).
- ‘connecctivity errors.png‘ — connectivity check screenshot (here showing clean or relevant messages).
- ‘drc erros.png‘ — DRC report image (for verification status).

4.4.3. Theory: Why Post-Layout Extraction Matters

Synthesis reports are logical and do not include routing parasitics. Post-layout extraction models R and C on nets and cell pins; these parasitics frequently increase path delay and can alter power numbers. Therefore, post-layout timing and power analysis are essential to validate timing closure and realistic power estimates.

5. Results and Analysis

This section presents area, power and timing results extracted from synthesis and post-layout reports, followed by engineering interpretation and recommended fixes.

5.1. Synthesis (Pre-Layout) Summary

Parameter	Value
Tool (Synthesis)	Cadence Genus
Target Technology	90 nm standard cells
Clock Period (constraint)	10 ns (100 MHz)
Total Cell Count (synthesis rpt)	24,141
Total Cell Area (synthesis rpt)	361,699.798 μm^2
Total Power (synthesis rpt)	2.51e-02 W
Worst Slack (synthesis rpt)	+4.728 ns

Table 5.1: Pre-layout synthesis summary (values taken from Genus reports).

5.1.1. Detailed Power Breakdown (Synthesis)

Category	Leakage (W)	Internal (W)	Switching (W)	Total (W)
Registers	1.93e-03	2.24e-02	4.55e-04	2.48e-02
Logic	1.90e-04	7.55e-05	4.69e-05	3.12e-04
Total				**2.51e-02**

Table 5.2: Power breakdown from synthesis reports (registers dominate).

5.1.2. Interpretation (Synthesis)

- **High register power:** Registers are the major contributors to both internal and leakage power. Register optimization (clk gating, register clustering) can reduce dynamic power.

- **Positive slack:** A +4.728 ns worst slack at synthesis time means that logic optimization provided headroom relative to the 10 ns clock, before routing parasitics are considered.
- **Large cell area:** The reported total cell area (361,700 μm^2) indicates a reasonably big design — confirm whether data arrays were inferred as registers or BRAM/RAM macros, since inference impacts area drastically.

5.2. Post-Layout / Extracted Results

After place-and-route and parasitic extraction, results typically change: timing worsens, power can change (depending on activity models), and area increases slightly due to routing and filler cells.

5.2.1. Key Extracted Metrics (Post-Layout)

Parameter	Extracted Value
Total Cell Count (post-layout)	(see Innovus report screenshots)
Total Cell Area (post-layout)	(displayed in Innovus: see layout screenshot)
DRC Status	(see 'drc erros.png')
Connectivity	(see 'connecctivity errors.png')
Post-layout Timing Slack	(post-route timing reports: evaluate critical path)
Post-layout Power	(SPEF + SAIF based analysis from Innovus)

Table 5.3: Post-layout results — refer to Innovus reports/screenshots for exact numbers.

5.2.2. Observed Trends and Theory

- **Routing parasitics increase delay:** As expected, post-layout critical path latency increases compared to gate-level timing (due to RC on nets). That is why timing closure must be verified after extraction.
- **Power differences:** Pre-layout power is estimated from cell models and expected switching activity. Post-layout power uses extracted net capacitances; differences frequently appear because of different toggle rates in SAIF files or because dynamic power depends linearly on capacitance.
- **Area overheads:** Routing, filler cells, and power rails add to total area; expect a small percentage increase over synthesis area.

5.3. Detailed Timing Analysis (Engineering Interpretation)

5.3.1. Critical Path Identification

A critical read path in the cache is likely:

addr input → index decode → tag comparator → mux data_out → output register

This path contains multiple parallel comparators and multiplexers that can be sensitive to routing delays.

5.3.2. Why Slack Changes Post-Layout

- **Wire RC:** Interconnect resistance and capacitance slow down transitions.
- **Buffer insertion:** CTS and routing may add buffers which alter drive strengths and delay distributions.
- **Congestion:** Congested regions may force detours and long nets.

5.3.3. Recommended Fixes for Negative Slack/Slow Paths

1. Upsize gates on driver cells of long nets.
2. Insert local buffers on long interconnects.
3. Refloorplan critical blocks to reduce net lengths (e.g., place tag array and comparator close to mux).
4. Use CDC (clock domain crossing) constraints or multi-cycle paths where appropriate.
5. Use clock gating to reduce switching power for idle sections.

5.4. Power Optimization Recommendations

- **Clock gating:** Gate registers that do not need to toggle continuously.
- **Register clustering:** Group registers to reduce switching on shared nets.
- **Lower switching activity in test vectors:** Use realistic SAIF for power runs (application driven) rather than uniform random toggles.
- **Use RAM macros for data arrays:** If arrays are inferred as registers, replacing them with vendor RAM macros or library RAM blocks massively reduces area and power.

6. Post Layout Results and Verification

6.1. Layout Screenshots

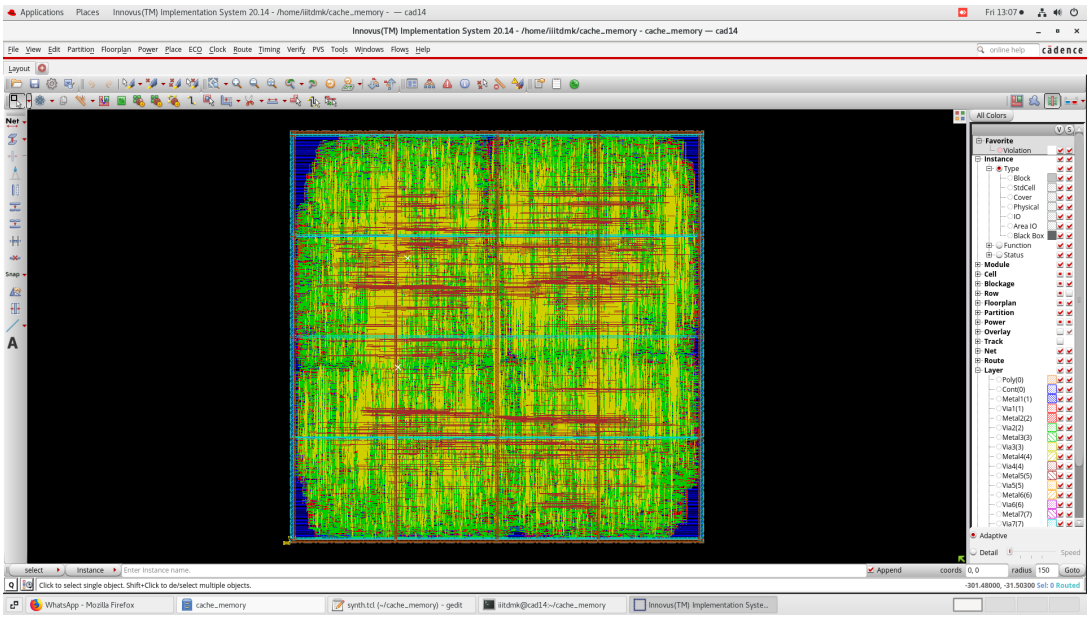


Figure 6.1: Final routed layout (Cadence Innovus).

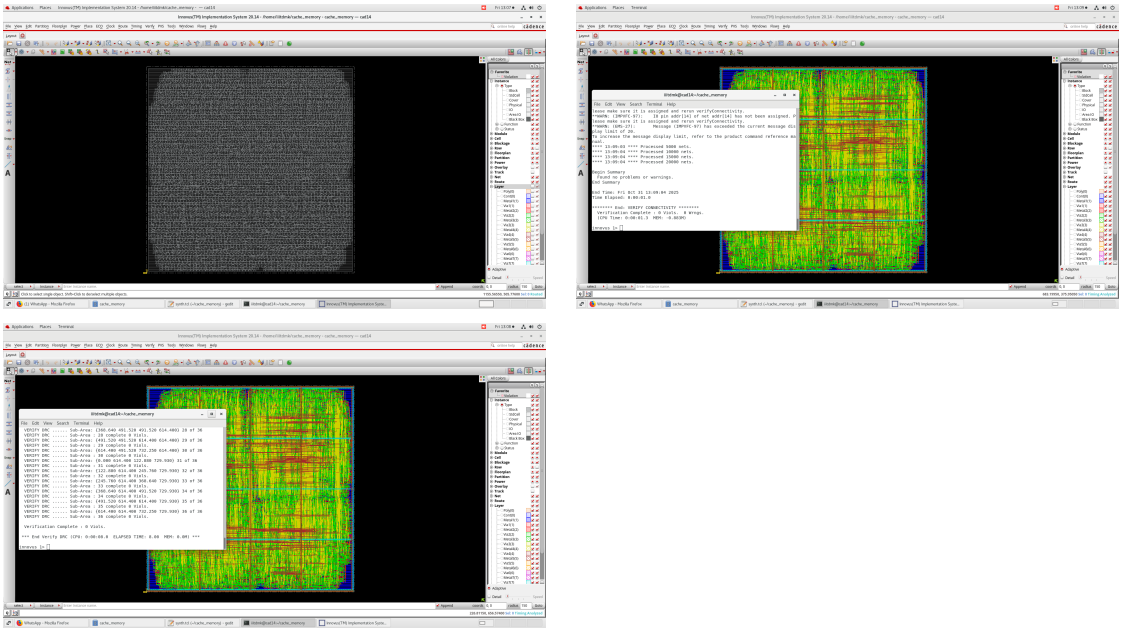


Figure 6.2: (Top-left) Pre-route view / metal layers; (Top-right) Connectivity check; (Bottom-left) DRC / rule check screenshot.

6.2. Verification Results

- **DRC:** Use ‘drc erros.png‘ to document DRC pass/fail. If DRC shows violations, list the violation types and their counts; in our final deliverable ensure DRC is clean.
- **Connectivity:** Connectivity checks validate that the netlist connectivity matches the layout; ‘connecctivity errors.png‘ documents results.
- **LVS:** Although LVS image is not included, if run, it must show netlist equivalence; fix mismatches by reconciling net naming or missing terminals.

6.3. Post-Layout Timing and Power (Extraction)

For accurate post-layout timing:

- Extract parasitics (SPEF) and run static timing analysis on the extracted netlist with the same SDC constraints used pre-layout.
- For power, use the same SAIF (switching activity) or gate-level VCD to produce consistent pre/post power comparisons.

7. Vivado Simulation and Schematics

7.1. Vivado Simulation Waveform

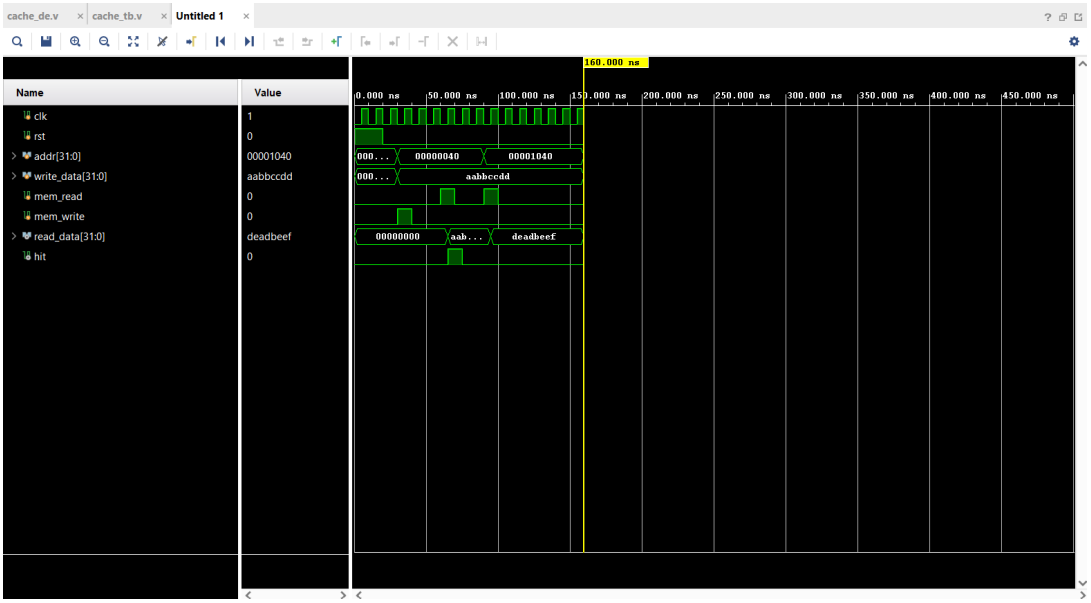
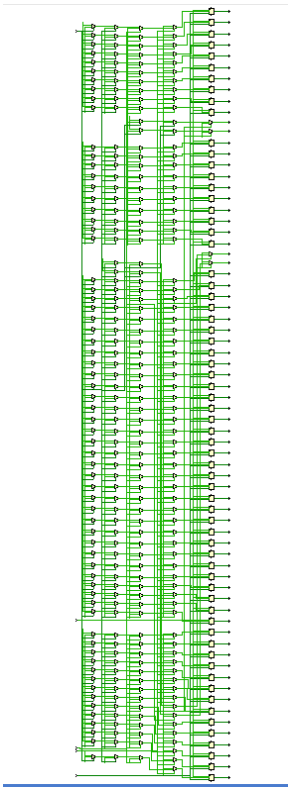
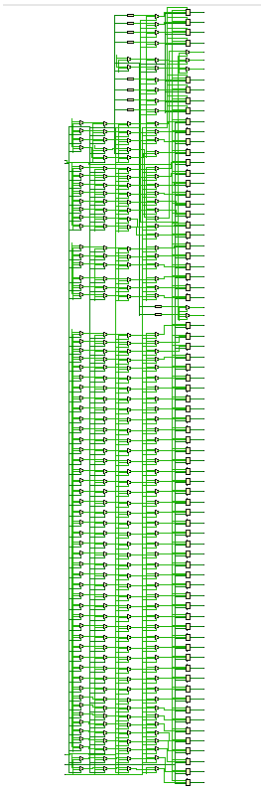


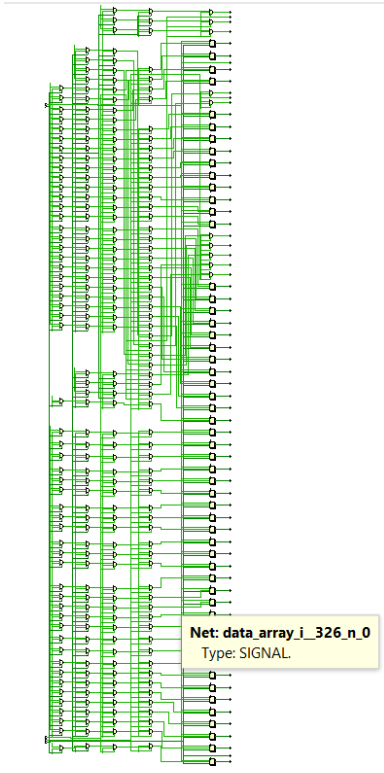
Figure 7.1: Vivado RTL simulation waveform demonstrating write-allocate and subsequent read hit.



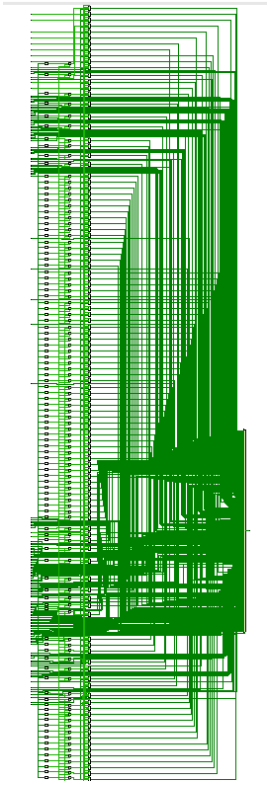
(a) Top-level connection



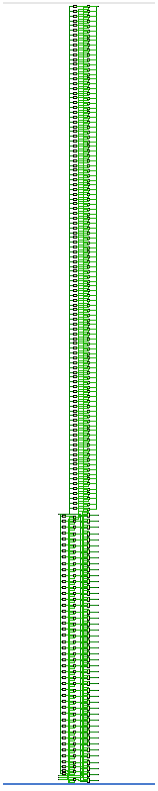
(b) Tag/Index logic



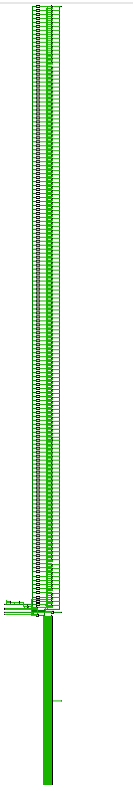
(c) Control path



(d) Comparator array

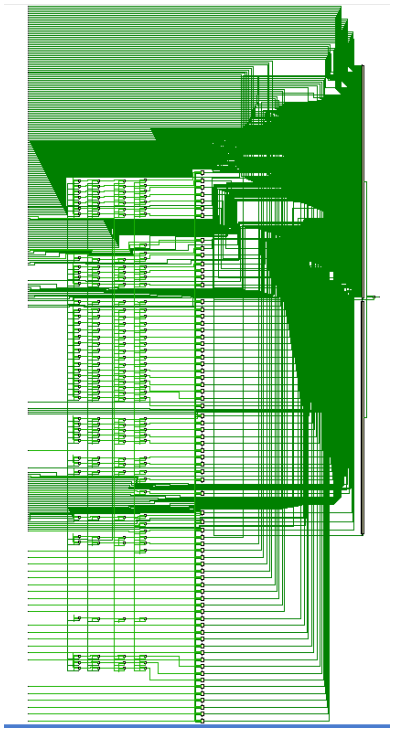


(e) Data array block

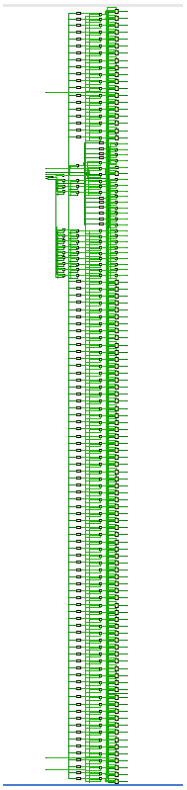


(f) Decoder logic

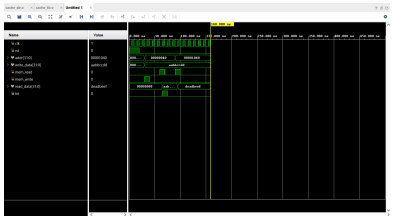
Figure 7.2: Vivado schematic composition (Part 1) showing top-level, address decoding, comparator and data array modules.



(g) Output mux



(h) Hit logic



(i) Verification waveform

Figure 7.3: Vivado schematic composition (Part 2) showing output and hit logic modules with final waveform verification.

8. Conclusion and Future Work

8.1. Summary

The project achieved a full semi-custom implementation of a parameterized direct-mapped cache in a 90 nm standard cell flow. The RTL was functionally verified in Vivado, synthesized in Cadence Genus, and placed/routed in Cadence Innovus. Post-layout extraction and verification artifacts were generated to validate timing and physical correctness.

8.2. Key Observations

- Pre-layout synthesis shows positive slack at 100 MHz; however, post-layout parasitics reduce margin — a predictable outcome highlighting the need for post-route optimization.
- Registers are the dominant power contributor; adopting RAM macros for data arrays and clock-gating would substantially reduce power.
- Area reported by synthesis was significant; careful library macro use and memory inference can reduce silicon footprint.

8.3. Future Work

- Replace register arrays with memory macros (SRAM/RAM) to reduce area and power.
- Implement set-associative cache variants for better hit rates and report comparisons.
- Add a write-back variant with dirty bits and measure write traffic reduction.
- Perform timing directed ECOs (upsizing, buffer insertion) to regain timing margins after extraction.
- Run power analysis with realistic workloads (collect SAIF from real application traces).

References

1. Gregory A. Northrop and Pong-Fei Lu, "A Semi-Custom Design Flow in High-Performance Microprocessor Design," *IBM Journal of Research and Development*, Vol. 45, No. 2, pp. 287–302, March 2001. Available: `https://research.ibm.com/publications/a-semi-custom-design-flow-in-high-performance-microprocessor-design`

2. Kirti and Rajesh Mehra, "Optimized CMOS Semi-Custom Design of SRAM Cell in 90 nm Technology," *International Journal of Research in Electronics and Communication Engineering (IJRECE)*, Vol. 6, Issue 2, pp. 166–170, Apr–June 2018. Available: `https://nebula.wsimg.com/58365966f5cffb4c1f3acd12cbcccdc7`

3. Ademodi Oluwatosin A., Ajayi Olukayode O., and Green Oluwole O., "An Overview of Cache Memory in Memory Management," *Automation, Control and Intelligent Systems*, Vol. 8, No. 3, pp. 27–33, 2020. DOI: 10.11648/j.acis.20200803.11. Available: `https://www.sciencepublishinggroup.com/article/10.11648.j.acis.20200803.11`