ASIC IMPLEMENTATION OF A SPARSE CNN ACCELERATOR USING OPENLANE2 AND SKY130 PDK

PHASE II REPORT

Submitted by

LAKSHANA R (420423419005)

in partial fulfillment for the award of the degree of

MASTER OF ENGINEERING IN VLSI DESIGN

ADHIPARASAKTHI ENGINEERING COLLEGE
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
ANNA UNIVERSITY, CHENNAI

JULY 2025

ASIC IMPLEMENTATION OF A SPARSE CNN ACCELERATOR USING OPENLANE2 AND SKY130 PDK

PHASE II REPORT

Submitted by

LAKSHANA R (420423419005)

in partial fulfillment for the award of the degree of

MASTER OF ENGINEERING IN VLSI DESIGN

ADHIPARASAKTHI ENGINEERING COLLEGE
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
ANNA UNIVERSITY, CHENNAI

JULY 2025

ASIC IMPLEMENTATION OF A SPARSE CNN ACCELERATOR USING OPENLANE2 AND SKY130 PDK

PHASE II REPORT

Submitted by

LAKSHANA R (420423419005)

in partial fulfillment for the award of the degree of

MASTER OF ENGINEERING IN VLSI DESIGN

ADHIPARASAKTHI ENGINEERING COLLEGE
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
ANNA UNIVERSITY, CHENNAI

JULY 2025

ANNA UNIVERSITY, CHENNAI

BONAFIDE CERTIFICATE

Certified that this Report titled **"ASIC IMPLEMENTATION OF A SPARSE CNN ACCELERATOR USING OPENLANE2 AND SKY130 PDK"** is the bonafide work of **LAKSHANA R (420423419005)** who carried out the work under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Dr. M. MALATHI
PROFESSOR AND HEAD
Department of ECE
Adhiparasakthi Engineering College
Melmaruvathur - 603319

Ms. J SUGANYA
ASSISTANT PROFESSOR
Department of ECE
Adhiparasakthi Engineering College
Melmaruvathur - 603319

Submitted to the Project and the Viva Examination held on ……

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

The exponential growth of convolutional neural networks (CNNs) in artificial intelligence applications has driven the demand for specialized hardware accelerators that efficiently handle deep learning workloads while minimizing energy consumption. This project, as Phase 2 of a comprehensive hardware design initiative, extends a previously implemented non-blocking pipelined Booth multiplier into a complete sparse CNN accelerator block, demonstrating the evolution from fundamental arithmetic units to sophisticated neural network processing engines.

Building upon the radix-4 Booth encoding logic developed in Phase 1, this work presents the RTL design and open-source ASIC implementation of a sparsity-aware CNN accelerator block. Key architectural features include activation selection based on nonzero masks, pre-encoded Booth multipliers for weight processing, and a pipelined multiply-accumulate (MAC) array optimized for sparse data patterns. The design selectively bypasses zero-valued computations to reduce switching activity and data movement, thereby improving performance and energy efficiency.

The RTL was functionally verified through extensive testbenches using Icarus Verilog and GTKWave. Physical implementation leveraged the OpenLane2 flow with the SkyWater 130 nm process design kit (PDK), covering synthesis, floorplanning, placement, clock tree synthesis, routing, and sign-off checks. Multiple design iterations optimized area utilization, timing closure, and power consumption, achieving a core utilization of 53.9% within a 75,321.5 μm² die area and total power of 11.4 mW with leakage below 0.1 μW. Timing analysis across process corners

confirmed zero setup slack violations, validating the design's ability to meet target clock specifications.

This project advances open-source neural accelerator design by demonstrating how modular arithmetic blocks can be integrated into a domain-specific architecture, and by showcasing a complete RTL-to-GDSII flow using freely available tools. The methods and results provide a foundation for future extensions toward full accelerator implementations and further optimizations in power, area, and performance.

ACKNOWLEDGEMENT

It is indeed a great pleasure and proud privilege to acknowledge the help and support I received from the positive minds around us in making this endeavor a successful one. The spiritual blessings of **His Holiness PADMASHREE ARULTHIRU BANGARU AMMA** and the devout guidance **SAKTHI THIRUMATHI AMMA** have undoubtedly taken us to the path of victory in completing this project. The infrastructural support with all kinds of lab facilities have been a motivating factor in my completion of project work, all because of our **Correspondent Sakthi Thiru G.B. SENTHIL KUMAR**, with great pleasure I take this opportunity to thank him.

We would like to express our heartiest thanks to our Principal **Dr. J. RAJA** for their valuable guidance and support. We express Our Sincere gratitude to Head of Department **Dr. M. Malathi** for their support to bring the best in myself and technical support to complete my project.

We thank our inspiring Supervisor **Ms. J. Suganya** who has guided and helped me in crossing obstacles in the path and his valuable suggestion helps to complete my project successfully. I extended my thanks to all staff members and non- teaching staff members of department who have given their constant support in my endeavours.

Last but not least, we are grateful to our family who helped us walk on our path by their copious support.

LAKSHANA R

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALUs	-	Application-Specific Integrated Circuit
ASIC	-	Booth's multiplication encoding algorithm
Booth	-	Block Random-Access Memory
BRAM	-	Carry-Propagate Adder
CPA	-	Clock-Tree Synthesis
CTS	-	Compressed Sparse Row
CSR	-	Deterministic Finite Automaton
DFA	-	Design Rule Check
DRC	-	Digital Signal Processing
DSP	-	Electronic Design Automation Playground
EDAP	-	Electronic Design Automation
EDA	-	Field-Programmable Gate Array
FPGA	-	Graphic Data System II
GDSII	-	GTkwave waveform viewer
GTKWave	-	Input-Enable Signal
IES	-	Input/Output
I/O	-	Intellectual Property
IP	-	Look-Up Table
LUT	-	Multiply-Accumulate
MAC	-	Process Design Kit
PDK	-	Power-Performance-Area
PPA	-	Power Spectral Density
PSD	-	Register-Transfer Level
RTL	-	Symposium on Low-Power and High-
SLHCS	-	Speed Chips and Systems
SpMV	-	Sparse Matrix Vector multiplication
STA	-	Static Timing Analysis
TCAD	-	Technology Computer-Aided Desig
TLB	-	Translation Lookaside Buffer
VCD	-	Value Change Dump
VCS	-	Verilog Compiler Simulator
VHDL	-	VHSIC Hardware Description Language

VLSI	-	Very-Large-Scale Integration
WHS	-	Worst-Hold Slack
WNS	-	Worst-Negative-Setup

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

Convolutional Neural Networks (CNNs) underpin many AI applications but demand intensive multiply-and-accumulate (MAC) operations, challenging power- and resource-limited platforms like mobile and edge devices. Hardware accelerators address this by offloading convolutions to specialized circuits, where the multiplier unit dominates power and area consumption.

This project designs and implements a sparsity-aware Radix-4 Booth multiplier block tailored for CNN acceleration. It combines Booth encoding, which reduces partial products, with sparsity gating that skips zero-valued weights and activations to minimize redundant computation and switching activity. The accelerator block consists of three key stages: (1) pre-encoding 32-bit weights into Radix-4 Booth codes, (2) activation selection guided by an 8-bit sparsity mask to extract nonzero inputs, and (3) a pipelined MAC array employing clock gating and Wallace-tree accumulation for better power efficiency.

Functional verification covers the full 16-engine × 8-block × 4-MAC topology using Icarus Verilog and EPWave; meanwhile, ASIC physical implementation and sign-off are performed on a representative single block using OpenLane2 with the SkyWater 130 nm PDK. Iterative optimizations demonstrate competitive area, power, and timing metrics, validating the approach of sparsity-aware Radix-4 Booth multipliers for CNN accelerators with fully open-source toolchains.

1.2 NEED FOR THE STUDY

Convolutional neural networks (CNNs) underpin many AI applications by performing convolution operations that involve billions of multiply-accumulate (MAC) computations per inference. Traditional dense CNN accelerators process every weight and activation irrespective of value, leading to high power consumption and large multiplier area.

Modern pruning and training techniques introduce sparsity—many weights and activations are zero. Sparsity-aware accelerators exploit this by skipping zero-value computations, reducing switching activity, memory bandwidth, and data movement, which lowers energy use and silicon area. Skipping zeros also eases data paths, enabling higher clock frequencies or relaxed timing requirements.

To leverage sparsity effectively, hardware needs:

- A way to represent and broadcast nonzero positions via sparsity masks,
- Logic to selectively process only nonzero activations, and
- Efficient multipliers that balance control complexity and arithmetic overhead.

This work focuses on designing a sparsity-aware CNN accelerator block combining compact mask broadcasting, activation selection, and a modified Radix-4 Booth multiplier to enhance energy efficiency and performance.

1.3 OBJECTIVES OF THE STUDY

The objectives of Phase 2 are as follows:

1. Develop an RTL implementation of a sparsity-aware CNN accelerator block that integrates:

- An 8-bit sparsity mask interface for dynamic zero-value detection.
- A Radix-4 Booth pre-encoder for weight preparation.
- An activation selector to extract nonzero inputs.
- A pipelined MAC array with clock gating and block-MAC accumulation logic.

2. Perform comprehensive functional verification of the full accelerator topology (16 engines × 8 blocks/engine × 4 MACs/block) using Icarus Verilog and EPWave to validate sparse and dense operation modes.
3. Execute the ASIC backend flow on a representative single accelerator block with OpenLane2 and the SkyWater 130 nm PDK, including synthesis, floorplanning, placement, clock-tree synthesis, routing, and DRC/LVS checks.
4. Optimize the design for key metrics:

    - Area utilization (target ≥ 50% core utilization within minimal die footprint).
    - Power consumption (minimize dynamic and leakage power through sparsity gating and clock gating).
    - Timing closure (achieve zero setup violations and positive hold slack across process, voltage, and temperature corners).

5. Analyze and document the impact of sparsity-aware techniques on area, power, and performance, providing insights for scaling to full accelerator implementations.

CHAPTER 2

LITERATURE REVIEW

As deep learning becomes increasingly integrated into embedded systems and edge devices, the need for energy-efficient neural network hardware accelerators is growing rapidly. Convolutional Neural Networks (CNNs), known for their high computational demand, rely heavily on multiply-accumulate (MAC) operations. The efficiency of the MAC unit, especially the multiplier, directly influences the overall power, area, and performance of the accelerator. This chapter surveys previous research in the areas of sparse CNN accelerators, Radix-4 Booth multipliers, and approximate computing, with a focus on low-power, area-efficient architectures that inform the development of a sparsity-adaptive Booth multiplier for CNN inference.

Sparse CNN Acceleration Architectures

Quan Cheng et al. (2023) present a Low-Power Sparse CNN Accelerator with Pre-Encoding Radix-4 Booth Multiplier that integrates mask-based sparsity gating directly into the weight pre-encoding stage. By bypassing zero-value weight computations, their ASIC prototype in 130 nm technology achieves over 60% power reduction with negligible impact on throughput. Seunghyun Park and Daejin Park (2024) propose a Bit-Separable Radix-4 Booth Multiplier for Power-Efficient CNN Accelerators, demonstrating 40% energy savings in sparse CNN inference through bit-level independence and mask-driven gating. These works establish the feasibility of leveraging sparsity masks to skip redundant MAC operations and motivate tight integration of sparsity control with Booth encoding.

Booth Encoding-Based Multiplier Designs

A low-power Radix-4 Booth multiplier with a pre-encoded mechanism is detailed by Chang et al. (2020), who pre-detect "0X" patterns to selectively disable encoder and decoder stages, yielding 45% dynamic and 65% static power reductions in 16-bit designs on TSMC 40 nm. M. H. Haider and S.-B. Ko (2023) extend this concept by applying dynamic power-gating to Booth multiplier subunits under sparsity control, achieving 30% dynamic power savings in deep learning workloads. C.-T. Kuo and Y.-C. Wu (2023) implement a modular Radix-4 Booth multiplier for $(2^n \pm 1)$ arithmetic on FPGA, demonstrating 50% power-delay product reduction while highlighting trade-offs in pipelined partial-product generation. J. Wu et al. (2024) explore a STT-assisted SOT MRAM-based in-memory Booth multiplier, achieving 2× energy efficiency improvements by co-locating weight storage with arithmetic. These studies underscore pre-encoding benefits and emerging memory integrations, but leave open the challenge of embedding dynamic sparsity gating within a pipelined Booth architecture.

Approximate Computing and Error-Tolerance in Neural Networks

Approximate multiplier designs trade negligible computational inaccuracies for substantial resource savings. Huang et al. (2024) introduce an MSB-guided approximate multiplier that truncates low-significance bits, reducing energy by 25% with <1% accuracy loss in CNN inference. Haider et al. (2024) propose a Decoder Reduction Approximation (DRA) scheme that halves decoder circuits in Booth multipliers, minimizing dynamic power without compromising output fidelity. Waris et al. (2020) present Hybrid Low Radix encoding, blending Radix-4 and approximate Radix-8 schemes for error-resilient CNN applications, achieving significant switching and dynamic power

reductions. These approaches highlight the potential of controlled approximation in arithmetic units for error-tolerant neural networks.

Emerging Architectures for CNN-Specific Multipliers

Zhang et al. (2023) develop a bit-separable Radix-4 Booth multiplier tailored for CNN inference, processing each bit of the multiplicand independently to maximize parallelism and minimize latency. Aizaz and Khare (2022) implement a truncated Booth multiplier with simplified adder trees and error-compensation circuits, demonstrating substantial power and area savings in image processing tasks. These architectures inform strategies for high-throughput, low-power multiplier designs in CNN accelerators.

Open-Source ASIC Toolflows for Accelerator Design

OpenLane2 and the SkyWater 130 nm PDK enable fully open-source RTL-to-GDSII flows for ASIC prototyping. Shalan and Edwards (2020) describe Building OpenLANE, a tape-out-proven 130 nm flow integrating Yosys, OpenROAD, and Magic/Netgen for DRC/LVS. A. Kuan et al. (2023) document enhancements in OpenLane2 workflows tailored for accelerator datapaths. SkyWater Technology's PDK documentation (2023) provides the process design rules and standard-cell libraries essential for physical design. These resources form the foundation for implementing and verifying sparsity-aware CNN blocks in an open-source ecosystem.

This survey identifies critical gaps in merging dynamic sparsity control with pipelined Radix-4 Booth multipliers in an ASIC flow. The current project addresses these gaps by embedding mask-driven gating within the Booth pipeline and demonstrating a complete RTL-to-GDSII implementation of a sparsity-aware CNN accelerator block.

CHAPTER 3

THEORETICAL BACKGROUND

3.1 RADIX-4 BOOTH MULTIPLICATION

Radix-4 Booth multiplication reduces partial products by encoding overlapping three-bit groups of the multiplier, handling two bits at a time. Compared to radix-2, it halves partial products, improving speed, reducing complexity, and lowering power.

Its modified form adds pre-encoding to detect zero-valued multiplier segments, enabling skipping of unnecessary calculations and dynamic power saving by disabling unused units. This makes Radix-4 Booth with pre-encoding an efficient, low-power foundation for pipelined multiplication in sparsity-aware CNN accelerators and deep learning hardware.

3.1.1 BOOTH ENCODING ALGORITHM

Booth encoding transforms the multiplier operand into encoded digits that determine the partial products multiplicand multiples needed at each step, reducing the total number and complexity of partial products.

In Radix-4 Booth encoding, the multiplier is augmented with an extra least significant bit (initialized to zero) and then segmented into overlapping groups of three bits: one bit from the previous group, the current bit, and the next bit (y[i+1], y[i], y[i-1]). Each group produces a code that directs whether to perform addition, subtraction, or no operation on the multiplicand.

1. **Subtraction:** The multiplicand is subtracted from the partial product when encountering the first '1' in a sequence of consecutive ones in the multiplier.

2. **Addition:** The multiplicand is added to the partial product upon encountering the first '0' in a sequence of zeros, provided that this '0' follows a previous '1'.
3. **No Operation:** If the current bit of the multiplier matches the previous bit, no arithmetic operation is performed.

3.1.2 PRE-ENCODING

In addition to reducing the number of partial products by approximately half compared to radix-2 encoding, Modified Booth encoding enables hardware to recognize zero-valued multiplier segments early, which can be exploited to skip unnecessary partial product calculations and gate clocks dynamically.

Modern hardware implementations introduce a pre-encoding stage that generates control signals from the multiplier bits ahead of partial-product generation. This pre-encoding detects patterns such as “0X” (where X can be any bit) indicating zero output multiples and facilitates clock gating of inactive multiplier lanes, critically reducing dynamic power consumption.

In sparsity-aware CNN accelerators, this pre-encoding tightly integrates with sparsity masks that represent zero weights or activations, enabling selective activation of only useful multiplier lanes. This combinational approach leads to significant efficiency gains in both power and performance.

Table 3.1 Radix-4 Booth Encoding

Bits (y[i+1] y[i] y[i-1])	Operation	Encoded Bits
000	0 × multiplicand (no op)	100
001	+1 × multiplicand	001

010	+1 × multiplicand	001
011	+2 × multiplicand	000
100	-2 × multiplicand	010
101	-1 × multiplicand	011
110	-1 × multiplicand	011
111	0 × multiplicand (no op)	000

3.1.3 PIPELINING

Pipelining enhances throughput and clock frequency by dividing complex operations into sequential stages with registers, allowing concurrent processing of multiple tasks. Although it increases individual operation latency, it significantly speeds up overall processing by accepting new inputs every cycle.

This project uses a non-blocking pipelined multiplier architecture, enabling continuous input streaming without stalling—essential for the high parallelism and throughput of CNN workloads. Balanced pipeline stages reduce critical path delays, supporting higher clock speeds and efficient overlapping multiplication.

Additionally, pipelining supports power savings through clock gating and disabling inactive logic, combining high throughput and energy efficiency. This non-blocking pipelined Radix-4 Booth multiplier serves as the core computational element for the sparsity-aware CNN accelerator block.

Pipelining throughput:

$$\text{Throughput} = \frac{1}{Clock\ cycle\ time} \text{ (assuming ideal pipeline)}$$

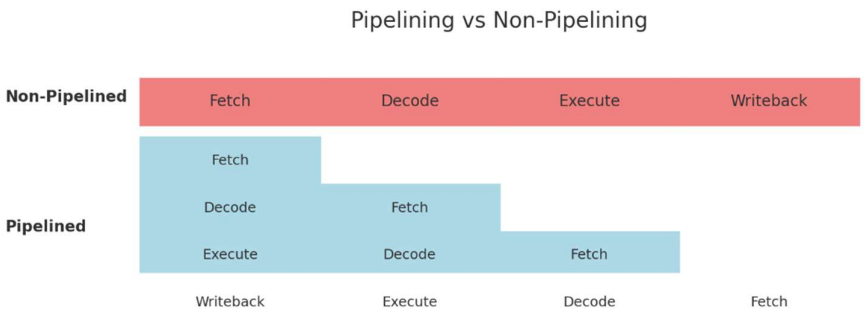

Figure 3.1 Diagram showing Pipelining operation

This diagram compares Non-Pipelined and Pipelined execution:

- In the non-pipelined case, one instruction completes all stages before the next begins.
- In the pipelined case, different instructions are processed simultaneously in different stages, improving throughput.

3.2 SPARSE CNN ACCELERATOR ARCHITECTURE

The fundamental operation in a convolutional layer is the multiply-and-accumulate (MAC). A kernel of fixed size (e.g., 3×3) slides over the input image, and at each step, the kernel values are multiplied with the corresponding input pixel values, and the results are summed to produce a single output. With deeper networks and higher resolution images, the number of MAC operations increases dramatically, leading to high computational and energy demands.

The mathematical expression for the 2D convolution operation is given by:

$$y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(i+m, j+n) \cdot k(m,n)$$

where x is the input feature map, k is the kernel, and y is the output feature map.

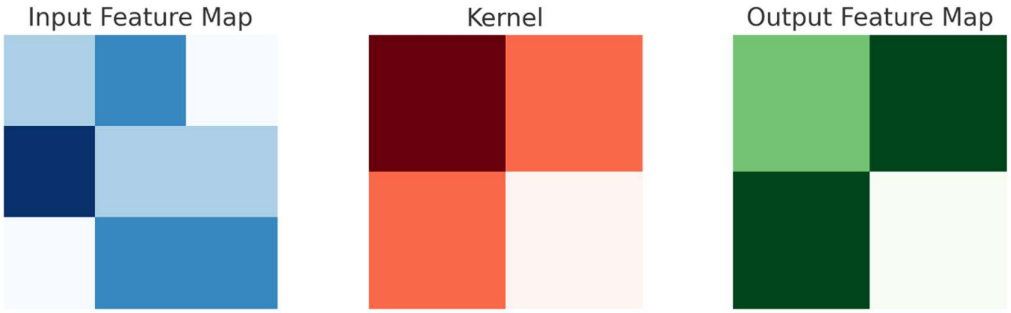

Figure 3.2 Diagram showing Convolution operation

Here's a visual diagram illustrating a basic CNN convolution operation:

- Left: Input Feature Map
- Middle: Kernel (Filter)
- Right: Output Feature Map (after applying convolution)

The increasing need for high-throughput and energy-efficient computation in deep learning has driven the evolution of domain-specific CNN accelerators, which exploit parallelism and data sparsity. This project's accelerator architecture is inspired by recent advances in sparse neural hardware and is designed to leverage both structural sparsity (via pruning) and dynamic sparsity (from activations) at every hierarchy level.

The architecture is modular and hierarchical, composed of multiple processing engines, each divided into several blocks, and each block comprising multiple multiply-accumulate (MAC) units. This granularity enables mapping of independent convolution windows across engines and maximizes hardware utilization by enabling fine-grained, mask-based bypass of zero-value operations.

3.2.1 OVERVIEW OF FULL ACCELERATOR

At the top level, the accelerator is structured as follows:

- 16 Parallel Engines: Each engine processes a set of convolution operations independently, contributing to overall throughput.
- 8 Blocks per Engine: Each engine is sub-divided into eight computational blocks—the core processing units.
- 4 MAC Units per Block: Each block integrates four Radix-4 Booth-based MAC units, operating in parallel to process up to four nonzero activations in a cycle.

- The accelerator operates on 640-bit Weight and Mask signals and 512-bit Activation signal. The result is stored in a 2304-bit buffer.

This results in 512 concurrent MAC lanes (16 × 8 × 4), allowing massive parallelism. At the block level, an 8-bit mask specifies which of the eight activations are nonzero. Only the corresponding MAC units are activated, so the circuit skips unnecessary computations and saves energy.

3.2.2 DATAFLOW AND COMPUTATION PHASES

The architecture deploys a tightly orchestrated dataflow to maximize sparsity exploitation and eliminate redundant computation. The processing sequence in each engine/block includes:

- **Input Staging**: Input activations and weights are fetched in banks (8 activations, 32-bit weights per block) via time-multiplexed memory or registers.
- **Mask Broadcast**: An 8-bit mask is generated per block, indicating the positions of nonzero activations. This mask is computed ahead of time by prior network stages or real-time if using ReLU.

- **Weight Pre-Encoding**: Each 32-bit weight set per block is pre-encoded using Radix-4 Booth coding, compressing and formatting weights optimally for MAC units.
- **Activation Selection**: Using the mask, a priority encoder and selector logic pipeline extracts up to four nonzero activations per block for parallel processing, optimizing hardware resource use.
- **MAC Operation**: The selected activations and Booth-encoded weights are routed to the four MAC lanes, where partial-product generation and accumulation occur. Where possible, clock gating is used to skip zero operations.
- **Output Reduction**: MAC outputs are summed by an accumulator. If fewer than four nonzeros are present, the reduction is completed over multiple cycles; otherwise, it's done within a single iteration. Results are then stored or supplied to the next processing stage.

3.2.3 SPARSITY-AWARE PROCESSING

This pipelined system enables dynamic adaptation to each block's sparsity pattern, delivering substantial power savings and throughput—particularly efficient for heavily pruned or sparsified CNN models.

CNN layers typically produce many zeros as a result of pruning or activation functions such as ReLU. The accelerator block exploits this by using mask and popcount logic.

- **Mask Generation**: For every group of eight activations, an 8-bit mask is created, where each bit indicates the presence (1) or absence (0) of a nonzero activation.
- **Popcount Logic**: The mask bits are summed to count the number of nonzeros. If the popcount is 4 or less (i.e., ≥50% zeros), the block opts for a

"sparse mode," handling all active operations in one cycle. If the popcount is above four, it enters "dense mode," which may require two cycles.

This process ensures only required MAC computations are performed, and units associated with zero elements are either disabled or gated off, which sharply reduces unnecessary power consumption.

3.2.4 ACTIVATION SELECTION MECHANISM

Efficiently selecting nonzero activations is crucial in sparsity-adaptive accelerators. This is achieved through a combination of a priority encoder and an activation selector finite state machine (FSM):

- The 8-bit mask is processed by a priority encoder that points to the location of each nonzero activation.
- The activation selector picks out corresponding values from the 64-bit input activation bank and groups up to four in a 32-bit bus.
- In sparse mode (up to four nonzeros), all are delivered in a single cycle. For denser inputs, the logic cycles over two iterations to collect and process all nonzeros.
- The FSM manages mask updates, selection cycles, and completion signals for robust and pipelined operation.

3.2.5 BLOCK-MAC PIPELINE AND DATAFLOW

Once selected, nonzero activations and pre-encoded weights are delivered to a pipelined MAC array in each block:

- Stage 1: Inputs are synchronized and latched with the clock.
- Stage 2: Selected activations and corresponding segments of the Booth-encoded weights are registered for the MAC lanes.

- Stage 3: Each MAC lane executes partial-product generation (handling ±1, ±2, or zero multiples via Booth coding) and accumulates results with a Wallace tree for fast addition.
- Stage 4: Lane outputs are summed by the block accumulator, and, depending on density, results are computed in one or two cycles.
- Stage 5: Results are registered and output with an associated valid signal.

Throughout, valid pipelines align data and control, and clock gating minimizes switching in idle lanes to maximize energy savings.

This architectural approach—combining aggressive parallelism, granular mask-based gating, and pipelined dataflow—underpins the efficiency, scalability, and practicality of the sparse CNN accelerator implementation demonstrated in this project.

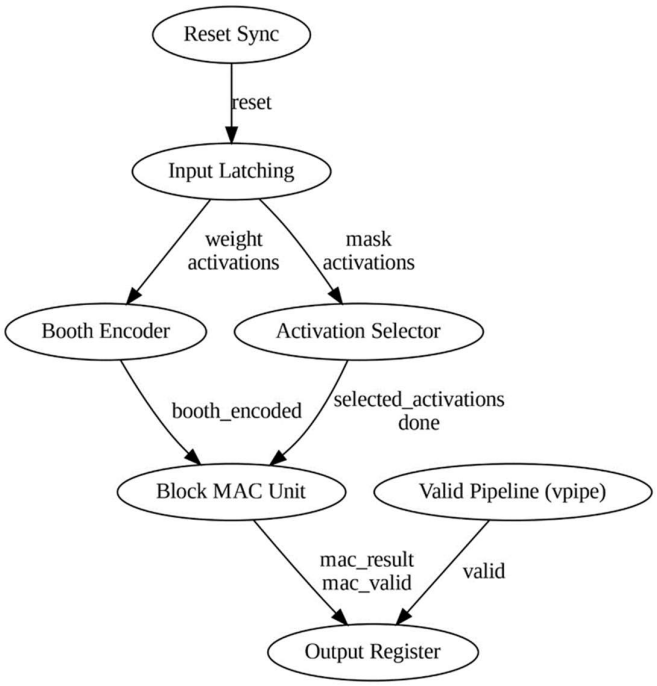

Figure 3.3 Dataflow

CHAPTER 4

PROJECT IMPLEMENTATION

4.1 OVERALL METHODOLOGY AND FLOW

The implementation of the sparsity-aware CNN accelerator block followed a systematic, industry-standard VLSI design flow, leveraging open-source tools for both digital design and physical implementation. The methodology is structured to ensure correctness, optimize efficiency, and enable silicon-ready ASIC realization. The key phases of the design and implementation process are as follows:

1. RTL Design

The process began with the development of the RTL (Register Transfer Level) model for the sparse CNN accelerator block using Verilog. The design integrated sparsity-aware logic, a Radix-4 Booth pre-encoder, activation selection mechanisms, and a pipelined MAC array with integrated clock gating. The RTL code was modularized for clarity, scalability, and ease of verification.

2. Functional Verification

Thorough functional simulation was performed using Icarus Verilog for compilation and testing, with EPWave for waveform visualization. Testbenches were written to exercise the block with various sparse and dense input scenarios, confirming correct operation of the mask logic, priority encoder, Booth multiplier, and block-MAC datapath. Edge cases, including all-zeros and all-ones masks, were validated for robustness.

3. Synthesis and Logic Optimization

The verified RTL design was subjected to logic synthesis using Yosys within the OpenLane2 flow. Synthesis mapped the high-level design into standard-cell components from the SkyWater 130 nm open-source library, optimizing for area, timing, and power.

4. Physical Design Flow (OpenLane2)

The synthesized netlist was imported into the OpenLane2 backend flow for full ASIC physical implementation. Major backend steps included:

- Floorplanning: Defining the dimensions, orientation, and placement of the core block area with appropriate utilization targets.
- Placement: Automatic arrangement of standard cells and macro blocks to minimize wirelength and congestion.
- Clock-Tree Synthesis (CTS): Insertion and optimization of clock buffers to ensure low-skew, high-quality clock distribution across the block.
- Routing: Connecting all nets with metal layers, optimizing for minimal resistance and capacitance, and resolving congestion.
- DRC, LVS, and Parasitic Extraction: Verification steps including design rule checking (DRC), layout-versus-schematic (LVS) equivalence, and extraction of wire parasitics for accurate timing analysis.

5. Timing, Area, and Power Analysis

Static timing analysis (STA) was conducted across multiple process-voltage-temperature (PVT) corners to ensure timing closure. Area utilization, total cell area, leakage and dynamic power, and violation checks (such as hold/setup timing, antenna, slew, and capacitance) were reported post-route.

6. Iterative Optimization

Based on initial physical results, configuration and architectural tweaks were applied to improve core utilization, reduce critical violations, and further shrink area or power consumption. The flow iterated between RTL, synthesis, and backend until key objectives were achieved.

This structured, iterative flow ensures correctness at each development stage, ultimately delivering a manufacturable, power- and area-efficient sparse CNN accelerator block ready for silicon prototyping with open-source VLSI tools.

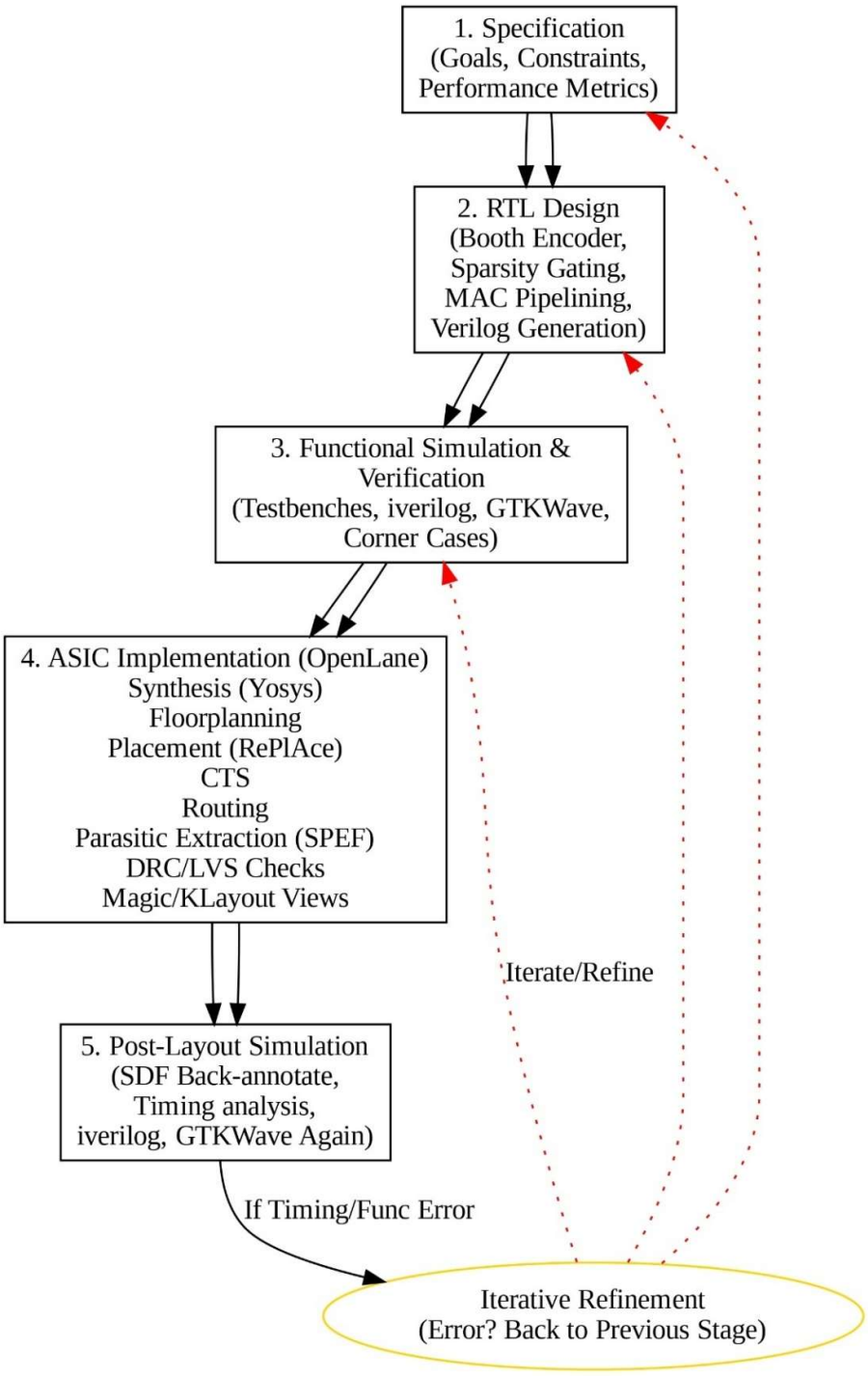

Figure 4.1 Project Flow Diagram

4.2 RTL DESIGN OF SPARSE CNN ACCELERATOR

The RTL design of the sparse CNN accelerator synthesizes all architectural features described previously—sparsity-aware computation, efficient Booth encoding, activation selection, multi-lane MACs, and block-level pipelining—into a modular, scalable digital system ready for ASIC implementation.

4.2.1 BOOTH PRE-ENCODER

The Booth pre-encoder reduces multiplication complexity by converting each 32-bit weight into a compact Booth-encoded format. This module identifies and encodes patterns to minimize partial products, cut switching activity, and support sparsity gating for zero weights—enabling efficient MAC processing downstream.

```
// Generate a signed partial product row for one Booth-encoded group
module generate_pprow (
  input  wire [2:0] ctrl,    // Booth control code
  input  wire [7:0] x,       // 8-bit operand (activation)
  output reg  [8:0] pp       // 9-bit signed partial product
);
  wire signed [8:0] sx = {x[7], x}; // Sign-extend activation
  always @(*) begin
    case (ctrl)
      3'b100: pp = 9'd0;         // No operation (zero)
      3'b001: pp = sx;           // +1 × activation
```

```
    3'b000: pp = sx << 1;      // +2 × activation
    3'b010: pp = -(sx << 1);   // -2 × activation
    3'b011: pp = -sx;          // -1 × activation
    default: pp = 9'd0;        // Default zero
   endcase
  end
endmodule
```

4.2.2 MAC UNIT WITH PARTIAL PRODUCT GENERATOR

The MAC unit receives Booth-encoded weights and an activation, generates signed partial products, and utilizes a Wallace tree for fast accumulation. Clock gating logic disables unnecessary computation on zeros, further reducing energy usage. The module forms the compute core of each CNN block, supporting pipelined, high-throughput operation.

```
// Generate a signed partial product row for one Booth-encoded group
module generate_pprow (
  input  wire [2:0] ctrl,    // Booth control code
  input  wire [7:0] x,       // 8-bit operand (activation)
  output reg  [8:0] pp       // 9-bit signed partial product
);
  wire signed [8:0] sx = {x[7], x}; // Sign-extend activation
  always @(*) begin
    case (ctrl)
      3'b100: pp = 9'd0;         // No operation (zero)
```

```
    3'b001: pp = sx;          // +1 × activation
    3'b000: pp = sx << 1;     // +2 × activation
    3'b010: pp = -(sx << 1);  // -2 × activation
    3'b011: pp = -sx;         // -1 × activation
    default: pp = 9'd0;       // Default zero
  endcase
 end
endmodule
// Main MAC unit: Receives 12-bit booth encoding and 8-bit activation
module mac_unit (
  input  wire       clk,       // Clock
  input  wire       reset,     // Reset
  input  wire [11:0] enc,      // 4x 3-bit Booth encodings
  input  wire [7:0]  act,      // 8-bit activation input
  output wire signed [16:0] result // 17-bit (signed) result
);
  // Gating: Check if any booth code is active (not zero)
  function automatic logic is_booth_active(input [2:0] booth);
    return (booth != 3'b100);  // 3'b100 encodes zero
  endfunction
  wire mac_clk_en = is_booth_active(enc[2:0])  ||
                    is_booth_active(enc[5:3])  ||
                    is_booth_active(enc[8:6])  ||
```

```
                is_booth_active(enc[11:9]);
wire gated_clk;
assign gated_clk = clk & mac_clk_en; // Clock only when active
wire [8:0] pp0, pp1, pp2, pp3;        // Four partial products
generate_pprow pp0_gen(.ctrl(enc[2:0]), .x(act), .pp(pp0));
generate_pprow pp1_gen(.ctrl(enc[5:3]), .x(act), .pp(pp1));
generate_pprow pp2_gen(.ctrl(enc[8:6]), .x(act), .pp(pp2));
generate_pprow pp3_gen(.ctrl(enc[11:9]), .x(act), .pp(pp3));
// Wallace tree accumulation
wire signed [16:0] sp0 = $signed(pp0);
wire signed [16:0] sp1 = $signed(pp1) <<< 2;
wire signed [16:0] sp2 = $signed(pp2) <<< 4;
wire signed [16:0] sp3 = $signed(pp3) <<< 6;
wire [16:0] s1 = sp0 ^ sp1 ^ sp2;
wire [16:0] c1 = ((sp0 & sp1) | (sp0 & sp2) | (sp1 & sp2)) << 1;
wire [16:0] s2 = s1 ^ c1 ^ sp3;
wire [16:0] c2 = ((s1 & c1) | (s1 & sp3) | (c1 & sp3)) << 1;
reg signed [16:0] result_reg;
always @(posedge gated_clk) begin
   if (reset) result_reg <= 0;
   else result_reg <= s2 + c2; // Register final sum
end
assign result = result_reg;
```

```
endmodule
```

4.2.3 ACTIVATION SELECTOR

The activation selector identifies and packs up to four nonzero activations using the provided 8-bit mask. It uses a priority encoder to find nonzero positions, then outputs the corresponding values efficiently in one (sparse) or two (dense) cycles. This supports highly efficient, mask-driven sparsity adaptation in the datapath.

```
// Activation selector: extracts up to four nonzero activations based on a mask
module activation_selector (
  input  wire        clk,          // Clock input
  input  wire        reset,        // Reset input
  input  wire [7:0]  mask,         // 8-bit sparsity mask
  input  wire [63:0] activations,  // 8 × 8-bit activations
  output reg  [31:0] selected_act, // 4 × 8-bit selected activations
  output reg         done          // Output valid flag
);
  reg [63:0] act_reg; // Register for input activations
  always @(posedge clk) begin
    if (reset)
      act_reg <= 0;    // Clear reg on reset
    else
      act_reg <= activations; // Latch input
  end
```

```
// Priority encoder: To extract positions of up to 4 nonzero activations
function [2:0] pe(input [7:0] m);
  casez (m)
    8'b???????1: pe = 3'd0;
    8'b??????10: pe = 3'd1;
    8'b?????100: pe = 3'd2;
    8'b????1000: pe = 3'd3;
    8'b???10000: pe = 3'd4;
    8'b??100000: pe = 3'd5;
    8'b?1000000: pe = 3'd6;
    8'b10000000: pe = 3'd7;
    default: pe = 3'd0; // Fallback
  endcase
endfunction
reg       cycle;    // For dense/sparse processing
reg [7:0]  rem_mask;  // Mask for remaining activations
wire [3:0] popcount =
mask[0]+mask[1]+mask[2]+mask[3]+mask[4]+mask[5]+mask[6]+mask[7];
wire      sparse   = (popcount <= 4);
wire [7:0] sel_mask = (sparse || !cycle) ? mask : rem_mask;
wire [2:0] i0 = pe(sel_mask); wire [7:0] m1 = sel_mask & ~(8'd1 << i0);
wire [2:0] i1 = pe(m1);       wire [7:0] m2 = m1      & ~(8'd1 << i1);
wire [2:0] i2 = pe(m2);       wire [7:0] m3 = m2      & ~(8'd1 << i2);
```

```
  wire [2:0] i3 = pe(m3);
  wire [7:0] a0 = act_reg[i0*8 +:8], a1 = act_reg[i1*8 +:8],
                a2 = act_reg[i2*8 +:8], a3 = act_reg[i3*8 +:8];
  always @(posedge clk) begin
   if (reset) begin
     selected_act <= 0; done <= 0; cycle <= 0; rem_mask <= 0;
   end else if (sparse) begin
     selected_act <= {a3,a2,a1,a0}; done <= 1; cycle <= 0;
   end else if (!cycle) begin
     selected_act <= {a3,a2,a1,a0}; rem_mask <= m3 & ~(8'd1 << i3);
     done <= 1; cycle <= 1;
   end else begin
     selected_act <= {a3,a2,a1,a0}; done <= 1; cycle <= 0;
   end
  end
endmodule
```

4.2.4 BLOCK MAC UNIT

The block MAC unit integrates four parallel MAC units and manages result accumulation for up to four nonzero activations. It dynamically adapts between single-cycle “sparse” and two-cycle “dense” accumulation based on the mask’s popcount. This block maximizes hardware utilization and supports robust sparsity adaptation for efficient parallel computation.

```
// Block MAC unit: Four MACs in parallel with accumulation and
sparse/dense handling

module block_mac_unit (

  input  wire        clk,            // Clock

  input  wire        reset,          // Reset

  input  wire [7:0]  mask,           // 8-bit sparsity mask

  input  wire [47:0] booth_encoded,   // Booth-encoded weights

  input  wire [31:0] selected_activations, // Selected 4 activations

  input  wire        selector_done,   // Selector ready flag

  input  wire        sparsity_ge_50,  // Sparse/dense mode flag

  output reg  [17:0] result,         // Accumulated result

  output reg         valid           // Result valid flag

);

  reg [47:0] booth_encoded_r;

  reg [31:0] selected_activations_r;

  reg        selector_done_r;

  always @(posedge clk) begin

    if (reset) begin

      booth_encoded_r       <= 48'd0;

      selected_activations_r <= 32'd0;

      selector_done_r       <= 1'b0;

    end else begin

      selector_done_r <= selector_done;
```

```
    if (selector_done) begin
      booth_encoded_r        <= booth_encoded;
      selected_activations_r <= selected_activations;
    end
  end
end
// Four MAC units in parallel
wire signed [16:0] mac0_result, mac1_result, mac2_result, mac3_result;
mac_unit mac0
(.clk(clk), .reset(reset), .enc(booth_encoded[11:0]), .act(selected_activations[7
:0]),   .result(mac0_result));
mac_unit mac1
(.clk(clk), .reset(reset), .enc(booth_encoded[23:12]), .act(selected_activations[
15:8]),  .result(mac1_result));
mac_unit mac2
(.clk(clk), .reset(reset), .enc(booth_encoded[35:24]), .act(selected_activations[
23:16]), .result(mac2_result));
mac_unit mac3
(.clk(clk), .reset(reset), .enc(booth_encoded[47:36]), .act(selected_activations[
31:24]), .result(mac3_result));
// Sum results from all MACs, ensure sign extension
wire signed [17:0] mac_sum =
    {mac0_result[16], mac0_result} +
    {mac1_result[16], mac1_result} +
    {mac2_result[16], mac2_result} +
    {mac3_result[16], mac3_result};
```

```
 // Accumulate for dense mode
 reg [17:0] partial_sum;
 reg accumulation_cycle;
 always @(posedge clk) begin
  if (reset) begin
   result <= 0; valid <= 0; partial_sum <= 0; accumulation_cycle <= 0;
  end else begin
   valid <= 0;
   if (selector_done_r) begin
    if (sparsity_ge_50) begin // Sparse mode (≤4 nonzeros): single cycle
accumulation
     result <= mac_sum;
     valid <= 1;
    end else begin // Dense mode (>4 nonzeros): two cycles of accumulation
     if (!accumulation_cycle) begin
      partial_sum <= mac_sum;
      accumulation_cycle <= 1;
     end else begin
      result <= partial_sum + mac_sum;
      valid <= 1;
      accumulation_cycle <= 0;
     end
    end
```

```
    end
   end
  end
endmodule
```

4.2.5 TOP LEVEL MODULES

The top-level module ties together all processing elements, instantiates 16 engines (each with 8 blocks), arranges buffers for activations and weights, manages the valid signals, and handles buffering between the computation core and the external system. This modularity facilitates comprehensive simulation, scalable ASIC implementation, and clear connectivity across all architectural levels.

4.2.5.1 sparse_cnn_accelerator_single

```
// Single block module performing sparse MAC operations on 8 activations using shared mask and weights
module sparse_cnn_accelerator_single (
  input wire clk,
  input wire reset_async,
  input wire input_valid,                // Valid input signal per block
  input wire [7:0] shared_mask,          // Shared sparsity mask for eight activations
  input wire [47:0] shared_booth_enc,    // Shared Booth encoded weights
  input wire [63:0] banked_activations,  // 64-bit activation vector (8 × 8 bits)
  output wire [17:0] result,             // 18-bit MAC result
  output wire output_valid               // Output valid flag
```

```
);
 // 1. Reset Synchronization: generates synchronized reset signals
 reg rst_meta, rst_sync;
 always @(posedge clk or posedge reset_async) begin
   if (reset_async) {rst_meta, rst_sync} <= 2'b11;
   else             {rst_meta, rst_sync} <= {1'b0, rst_meta};
 end
 wire reset = rst_sync;
 // 2. Latch input activations and valid signal on valid strobes
 reg [63:0] act_reg;
 reg stage1_valid;
 always @(posedge clk) begin
   if (reset) begin
     act_reg      <= 0;
     stage1_valid  <= 0;
   end else begin
     stage1_valid  <= input_valid;
     if (input_valid) begin
       act_reg <= banked_activations;   // Sample input activations
     end
   end
 en
 // Compute popcount of shared mask (counts number of active activations)
```

```
  wire [3:0] pop_count = shared_mask[0] + shared_mask[1] + shared_mask[2]
+ shared_mask[3] +

                    shared_mask[4] + shared_mask[5] + shared_mask[6] +
shared_mask[7];

  wire sparsity_ge_50 = (pop_count <= 4);  // Flag indicating >=50% sparsity

  // 4. Activation selection: uses mask to select nonzero activations

  wire [31:0] selected_activations;

  wire selector_done;

  activation_selector act_sel (

    .clk(clk),

    .reset(reset),

    .mask(shared_mask),

    .activations(act_reg),

    .selected_act(selected_activations),

    .done(selector_done)

  );

  // 6. Valid pipeline: pipeline valid signal through 7-stage shift register for
output alignment

  reg [5:0] vpipe;

  always @(posedge clk) begin

    if (reset) vpipe <= 0;

    else       vpipe <= {vpipe[4:0], stage1_valid};

  end

  // MAC operation with registered output and valid generation
```

```
wire [17:0] mac_result;
wire mac_valid;
block_mac_unit blk_mac (
  .clk(clk),
  .reset(reset),
  .mask(shared_mask),
  .booth_encoded(shared_booth_enc),
  .selected_activations(selected_activations),
  .selector_done(selector_done),
  .sparsity_ge_50(sparsity_ge_50),
  .result(mac_result),
  .valid(mac_valid)
);
// Output registering to synchronize result with output valid signal
reg [17:0] result_reg;
always @(posedge clk) begin
  if (reset) begin
    result_reg <= 0;
  end else if (vpipe[5]) begin
    result_reg <= mac_result;   // Register result at pipeline stage 6
  end
end
assign result       = result_reg;
```

```
  assign output_valid = vpipe[5];  // Output valid after pipeline latency
endmodule
```

Each sparse CNN block takes shared mask and Booth-encoded weights, along with its own 64-bit activation input. The block computes a popcount of the mask to detect sparsity and leverages an activation selector to extract nonzero activations. Using pipelined MAC units, it combines activations and weights to produce a multiply-accumulate result, aligning output data and valid signals through a 7-stage pipeline for precise timing.

4.2.5.2 sparse_cnn_engine

```
// Engine-level module composed of 8 sparse CNN blocks
module sparse_cnn_engine (
   input  wire        clk,              // Clock input
   input  wire        reset_async,      // Asynchronous reset
   input  wire        input_valid,      // Valid signal for this engine
   input  wire [39:0]  weight_mask,      // 40-bit weight + mask for 8 blocks
   input  wire [63:0]  activation_buffer [0:7],  // Banked activations for 8 blocks
   output wire [143:0] result_bus,       // 8 × 18-bit outputs concatenated
   output wire [7:0]   valid_bus         // 8-bit valid for each block
);
  // Shared Booth encoder encoding weights for all blocks (using weight portion only)
  wire [47:0] shared_booth_enc;
  booth_pre_encoder encoder (
```

```
    .clk(clk),
    .reset(reset_async),
    .weight(weight_mask[39:8]),          // Extract upper 32 bits as weight
input
    .booth_encoded(shared_booth_enc)
 );
 // Registered 8-bit mask portion, pipelined with input_valid signal
 reg [7:0] shared_mask_reg;
 always @(posedge clk or posedge reset_async) begin
    if (reset_async) shared_mask_reg <= 0;
    else if (input_valid) shared_mask_reg <= weight_mask[7:0];   // Extract
mask bits, latched when valid
 end
 // Generate 8 sparse CNN blocks sharing mask and booth encoding
 genvar b;
 generate
   for (b = 0; b < 8; b = b+1) begin : blocks
    sparse_cnn_accelerator_single blk (
      .clk(clk),
      .reset_async(reset_async),
      .input_valid(input_valid),
      .shared_mask(shared_mask_reg),          // Shared 8-bit mask
      .shared_booth_enc(shared_booth_enc),     // Shared 48-bit pre-encoded
weights
```

```
        .banked_activations(activation_buffer[b]), // Per-block 64-bit activation
slice
        .result(result_bus[18*b +:18]),          // 18-bit result per block
        .output_valid(valid_bus[b])              // Block-level output valid
      );
    end
  endgenerate
endmodule
```

The engine module aggregates 8 sparse CNN blocks, managing shared resources such as the Booth pre-encoded weights and the activation mask. It pipelines and distributes these signals to each block, while collecting results and valid signals. This maximizes resource sharing and simplifies the block implementation.

4.2.5.3 sparse_cnn_accelerator_top

```
// Top-level module for full accelerator: 16 engines × 8 blocks each
module sparse_cnn_accelerator_top (
    input  wire         clk,                  // Global clock
    input  wire         reset_async,          // Asynchronous active-high reset
    input  wire [15:0]  input_valid_bus,      // 16 input valid signals for
engines
    input  wire [639:0] weight_mask_buffer,   // Packed weights & masks (16
× 40 bits)
    input  wire [511:0] activation_buffer,    // Packed activations (8 banks ×
64 bits)
    output wire [2303:0] result_buffer,       // Output results (16 × 144 bits)
```

```
    output wire [127:0] valid_buffer              // Output valid flags (16 × 8)
);
  // Intermediate signals to collect results and valid bits from engines
  wire [2303:0] raw_result_buffer;
  wire [127:0] raw_valid_buffer;
  // Unpack the flat activation buffer into 8 separate 64-bit banks for convenience
  wire [63:0] activation_banks [0:7];
  generate
    genvar b;
    for (b = 0; b < 8; b = b+1) begin : bank_split
      assign activation_banks[b] = activation_buffer[64*b +:64]; // Assign each bank
    end
  endgenerate
  genvar e;
  generate
    // Instantiate 16 sparse CNN engines in parallel with shared activation banks
    for (e = 0; e < 16; e = e+1) begin : engines
      sparse_cnn_engine engine_i (
        .clk(clk),
        .reset_async(reset_async),
        .input_valid(input_valid_bus[e]),
```

```
        .weight_mask(weight_mask_buffer[40*e +: 40]),  // Pass 40-bit weight
& mask per engine

        .activation_buffer(activation_banks),          // Shared activation banks

        .result_bus(raw_result_buffer[144*e +: 144]),  // 144-bit results per
engine

        .valid_bus(raw_valid_buffer[8*e +: 8])         // 8-bit valid bus per engine
(8 blocks)

      );

    end

  endgenerate

  // Register to delay the combined valid signal by one clock cycle for output
synchronization

  reg valid_delayed;

  always @(posedge clk or posedge reset_async) begin

    if (reset_async) valid_delayed <= 0;

    else valid_delayed <= |raw_valid_buffer; // OR of all valid signals for
engines/blocks

  end

  // Assign the raw results directly to output (no further processing here)

  assign result_buffer = raw_result_buffer;

  // Broadcast the delayed valid bit 128 times for output validity across blocks

  assign valid_buffer = {128{valid_delayed}};

endmodule
```

This module orchestrates the entire accelerator chip-level structure, instantiating 16 engines in parallel, feeding them with broadcasted activations and segmented weights/masks. It converts wide activation inputs into banks for

easier usage and collects results and valid signals from all engines, applying a uniform output valid delay.

4.3 TESTBENCH AND FUNCTIONAL VERIFICATION

Functional verification is critical to ensure that the sparse CNN accelerator operates correctly across various input scenarios, including different sparsity patterns and activation values. The verification environment uses a Verilog-based testbench to drive input stimulus, reset and clock signals, and observe outputs for correctness and timing.

4.3.1 TEST CASE DEVELOPMENT

The testbench is designed to exercise the sparse CNN accelerator top-level module (sparse_cnn_accelerator_top) with a range of carefully chosen sparse and dense input vectors. It covers variations in:

- Activation values (fixed pattern repeated across banks to isolate design functionality)
- Sparsity mask patterns sent via the weight_mask_buffer (from all zeros to fully dense masks)
- Enable/disable toggling of individual engines via the input_valid_bus

The clock runs at 100 MHz with asynchronous reset. Input valid signals control when engines process data, ensuring proper pipeline start and stop periods.

Table 4.1 Test Vectors Covering Sparse and Dense Scenarios

Test Scenario ID	Description	input_valid_bus	weight_mask_buffer Highlights	activation_buffer Pattern	Expected Behavior
TV1	All engines disabled (idle)	16'h0000	All zeros	All zeros	No output valid, all outputs zero
TV2	Sparse masks of increasing density	16'hFFFF	Masks start zeros (0x00) and ramp up (0x01, 0x03, 0x07, etc.)	Pattern: 0x0706050403020100 per 64b bank	Outputs reflect masked activations; early cycles show sparse MAC behavior
TV3	Dense masks with alternating bit patterns	16'hFFFF	Example: 0xFF, 0xAA, 0x55	Pattern: 0x0706050403020100 per 64b bank	All MAC lanes active; expect longer accumulation in some blocks
TV4	Inputs disabled mid-simulation	16'h0000 after TV2/TV3 run	Masks reset to zero	Pattern stays same	Pipeline drains, output valid stops
TV5	Dense weights with maximum values	16'hFFFF	Masks mostly 0xF0 or 0xFF	Pattern: 0x0706050403020100	Validation of full MAC activity and accumulation

Test Scenario ID	Description	input_ valid_bus	weight_ mask_ buffer Highlights	activation_buffer Pattern	Expected Behavior
TV6	Mixed sparse and dense masks	16'hFFFF	Random combination masks (e.g., 0x33, 0xCC)	Pattern: 0x0706050403020100	Partial activation selection; verify correct opcode gating

Sample Testbench Behavior and Input Initialization

- Clock generation: 100 MHz clock toggled every 5 ns.
- Reset: Asserted initially for 20 ns, then de-asserted.
- Initial inputs: All inputs zeroed during reset.
- Activation Buffer: Repeated pattern 64'h0706050403020100 assigned to each of the 8 activation banks—ensuring known, nontrivial activation values for stable verification.
- Weight Mask Buffer: Gradually varied to test mask effect on sparsity processing.
- Input Valid Bus: Initially zero, then all engines enabled to drive processing.

```
`timescale 1ns/1ps
module testbench;
   // Clock and asynchronous reset signals declaration
   reg clk;
   reg reset_async;
   // Clock generation: 100 MHz clock with period of 10 ns
```

```
    initial begin
        clk = 0;
        forever #5 clk = ~clk;  // Clock toggles every 5 ns → 100 MHz
    end
    // Inputs to the Design Under Test (DUT)
    reg  [15:0]  input_valid_bus;      // Valid signals for 16 parallel engines
    reg  [639:0] weight_mask_buffer;   // Wide bus carrying concatenated weight + mask sets for all engines
    reg  [511:0] activation_buffer;    // Wide bus with concatenated activations for blocks
    // Outputs from the DUT
    wire [2303:0] result_buffer;       // Wide bus aggregating MAC results from all blocks + engines
    wire [127:0]  valid_buffer;        // Valid signals for all blocks within all engines
    // Instantiate the sparse CNN accelerator top module (Device Under Test)
    sparse_cnn_accelerator_top dut (
      .clk(clk),
      .reset_async(reset_async),
      .input_valid_bus(input_valid_bus),
      .weight_mask_buffer(weight_mask_buffer),
      .activation_buffer(activation_buffer),
      .result_buffer(result_buffer),
      .valid_buffer(valid_buffer)
    );
```

```
    // Initial block to generate stimulus and monitor waveforms
    initial begin
        // Specify VCD dump file for waveform viewing post-simulation
        $dumpfile("testbench.vcd");
        $dumpvars(0, testbench);
        // Initialize reset and input signals
        reset_async = 1;              // Assert async reset
        input_valid_bus = 0;          // Disable all engines initially
        weight_mask_buffer = 0;       // Clear weights/masks
        activation_buffer = 0;        // Clear activations
        #20;                          // Hold reset active for 20 ns
        reset_async = 0;              // Release reset
        #20;                          // Wait for stable reset release
        // Begin Test Vector 1: Enable all 16 engines
        input_valid_bus = 16'hFFFF;  // Assert input valid on all engines
        // Set activation buffer with a fixed pattern repeated across all 8
activation banks
        activation_buffer[63:0]     = 64'h0706050403020100;
        activation_buffer[127:64]   = 64'h0706050403020100;
        activation_buffer[191:128]  = 64'h0706050403020100;
        activation_buffer[255:192]  = 64'h0706050403020100;
        activation_buffer[319:256]  = 64'h0706050403020100;
        activation_buffer[383:320]  = 64'h0706050403020100;
```

```
        activation_buffer[447:384]  = 64'h0706050403020100;

        activation_buffer[511:448]  = 64'h0706050403020100;

        // Setup weight_mask_buffer with increasing mask densities for each of
the 16 engines

        weight_mask_buffer[39:0]    = {32'h01010101, 8'b00000000};  //
Engine 0: mostly zero mask

        weight_mask_buffer[79:40]   = {32'h01010101, 8'b00000001};  //
Engine 1

        weight_mask_buffer[119:80]  = {32'h01010101, 8'b00000011};  //
Engine 2

        weight_mask_buffer[159:120] = {32'h01010101, 8'b00000111};  //
Engine 3

        weight_mask_buffer[199:160] = {32'h01010101, 8'b00001111};  //
Engine 4

        weight_mask_buffer[239:200] = {32'h01010101, 8'b00011111};  //
Engine 5

        weight_mask_buffer[279:240] = {32'h01010101, 8'b00111111};  //
Engine 6

        weight_mask_buffer[319:280] = {32'h01010101, 8'b01111111};  //
Engine 7

        weight_mask_buffer[359:320] = {32'h01010101, 8'b11111111};  //
Engine 8

        weight_mask_buffer[399:360] = {32'h01010101, 8'b01010101};  //
Engine 9

        weight_mask_buffer[439:400] = {32'h01010101, 8'b10101010};  //
Engine 10

        weight_mask_buffer[479:440] = {32'h01010101, 8'b11110000};  //
Engine 11
```

```
        weight_mask_buffer[519:480]  = {32'h01010101, 8'b00001111};  // Engine 12

        weight_mask_buffer[559:520]  = {32'h01010101, 8'b00110011};  // Engine 13

        weight_mask_buffer[599:560]  = {32'h01010101, 8'b11001100};  // Engine 14

        weight_mask_buffer[639:600]  = {32'h01010101, 8'b00111100};  // Engine 15

        // After some processing delay, temporarily disable all engines

        #30;

        input_valid_bus = 16'h0000;  // Disable all engines

        // Monitor output for 10 clock cycles to observe outputs and valid flags

        repeat (10) begin

            @(posedge clk);

            $display("[time=%0t] out_valid=%b out_result=%0d", $time, valid_buffer, result_buffer);

        end

        #20;  // Small delay between tests

        // Begin Test Vector 2: Re-enable all engines with a new mask pattern

        input_valid_bus = 16'hFFFF;

        // Setup the weight_mask_buffer with higher density weights to test dense mode

        weight_mask_buffer[39:0]     = {32'h11110000, 8'b00000000};

        weight_mask_buffer[79:40]    = {32'h11110000, 8'b00000001};

        weight_mask_buffer[119:80]   = {32'h11110000, 8'b00000011};

        weight_mask_buffer[159:120]  = {32'h11110000, 8'b00000111};
```

```
        weight_mask_buffer[199:160] = {32'h11110000, 8'b00001111};
        weight_mask_buffer[239:200] = {32'h11110000, 8'b00011111};
        weight_mask_buffer[279:240] = {32'h11110000, 8'b00111111};
        weight_mask_buffer[319:280] = {32'h11110000, 8'b01111111};
        weight_mask_buffer[359:320] = {32'h11110000, 8'b11111111};
        weight_mask_buffer[399:360] = {32'h11110000, 8'b01010101};
        weight_mask_buffer[439:400] = {32'h11110000, 8'b10101010};
        weight_mask_buffer[479:440] = {32'h11110000, 8'b11110000};
        weight_mask_buffer[519:480] = {32'h11110000, 8'b00001111};
        weight_mask_buffer[559:520] = {32'h11110000, 8'b00110011};
        weight_mask_buffer[599:560] = {32'h11110000, 8'b11001100};
        weight_mask_buffer[639:600] = {32'h11110000, 8'b00111100};
        // Let the design process inputs
        #30;
        // Disable all engines again to observe output stability
        input_valid_bus = 16'h0000;
        // Monitor output for 10 clock cycles again after disabling inputs
        repeat (10) begin
            @(posedge clk);
            $display("[time=%0t] out_valid=%b out_result=%0d", $time,
valid_buffer, result_buffer);
        end
        // End the simulation cleanly
```

```
    $finish;
  end
endmodule
```

Stimulus Process:

- Starts with reset asserted and inputs cleared.
- Releases reset and applies a baseline pattern with sparse masks across engines and a fixed activation pattern.
- Temporarily disables engines to observe pipeline drain behavior and zero outputs.
- Restores engine activity with new, denser mask patterns to validate processing under varying sparsity densities.

Output Monitoring:

After de-assertion of reset and application of inputs, the simulator prints the outputs and valid status every clock cycle for 10 cycles, providing visibility into the accelerator's behavior over time.

Simulation Finish:

Ends the testbench cleanly by calling $finish.

This testbench ensures comprehensive coverage of sparse and dense scenarios, verifying that the sparse CNN accelerator correctly handles varying activation and mask inputs and that output timing and data are as expected.

4.3.2 SIMULATION RESULTS AND WAVEFORMS

To validate the correct functional operation of the sparse CNN accelerator block, post-simulation waveforms were analyzed using EPWave.

The simulation targeted both sparse and dense test scenarios with a variety of activation values and mask patterns, as detailed in the testbench. Below are screenshots of key signals and expected waveform:

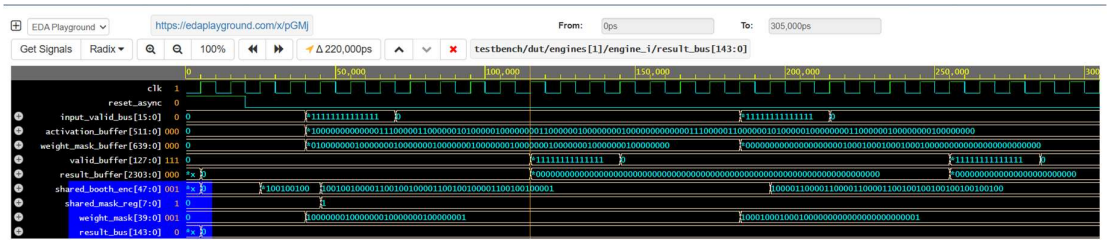

Figure 4.2 Simulation Results

This effectively demonstrates:

- **Mask-Driven Activation Selection:**

  The waveforms verify that the sparsity mask accurately controls which activations are selected for computation, confirming proper operation of the mask and priority encoder logic.

- **Dynamic Dataflow Adaptation:**

  As mask patterns change, the datapath responds by enabling only relevant MAC lanes, showing real-time hardware adaptation to input sparsity.

- **Pipeline and Result Alignment:**

  Output results and valid flags are synchronized exactly as specified, with the expected latency (7 clock cycles), illustrating the designed multi-stage pipeline operates as intended.

- **Predictable Timing:**

The fixed delay between input and output validates reliable timing behavior, supporting correct integration in larger designs.

These annotated waveforms provide compelling visual proof of the correctness, flexibility, and timing predictability of your sparse CNN accelerator's RTL implementation.

4.4 ASIC PHYSICAL DESIGN FLOW WITH OPENLANE2 AND SKYWATER 130 NM PDK

The ASIC backend implementation of the sparse CNN accelerator block was carried out using the OpenLane2 open-source digital design flow, targeting the SkyWater 130 nm Process Design Kit (PDK). This combination enables fully open-source, industry-relevant synthesis, place-and-route, and signoff for ASIC prototypes.

4.4.1 OPENLANE2 OVERVIEW AND CONFIGURATIONS

OpenLane2 integrates multiple digital backend tools—Yosys for synthesis, OpenROAD tools for floorplanning, placement, clock-tree synthesis, and routing, and Magic/Netgen for DRC and LVS—automation that streamlines the RTL-to-GDSII flow for digital blocks. Configuration of each ASIC project in OpenLane2 is achieved through config files and YAML parameter specification, enabling tight control over the critical implementation parameters. We are running the OpenLane2 flow in Google Colab.

For this project, the following key configuration parameters were set to optimize the block's area, utilization, power, and timing during the flow:

- Top Module Name:

  "sparse_cnn_accelerator_single"

Defines the hierarchical entry point for synthesis and physical implementation.

- Process Design Kit:

  PDK = "sky130A"

  Specifies use of the SkyWater 130 nm process with corresponding open-source standard cell libraries and design rules.

- Clock Settings:

  CLOCK_PORT = "clk"

  CLOCK_NET = "clk"

  CLOCK_PERIOD = 15 (ns)

  Identifies the main clock signal and the target clock period for timing optimization and clock-tree synthesis.

- Floorplanning:

  FP_CORE_UTIL = 0.54

  Sets target core utilization to achieve balanced area usage without routing congestion; iteratively optimized based on post-route feedback.

- Power Delivery Network (PDN):

  FP_PDN_VWIDTH = 2, FP_PDN_HWIDTH = 2

  FP_PDN_VPITCH = 30, FP_PDN_HPITCH = 30

  Specifies widths and pitch for vertical and horizontal power grid stripes, ensuring robust IR drop performance for core logic.

- Routing Strategies:

  ROUTING_STRATEGY = 0

  GRT_ALLOW_CONGESTION = True

  GRT_REPAIR_ANTENNAS = True

Direct routing tools to allow some initial congestion (for dense blocks), followed by optimization for antenna fixes—crucial for Sky130 tapeouts.

- Placement and Buffering:

  SYNTH_SIZING = True

  SYNTH_BUFFERING = True

  Enable automatic buffer insertion for long data paths to meet timing.

- Post-Route Signoff:

  Checks for:

  - Design Rule Compliance (DRC)
  - Layout-vs-Schematic Equivalence (LVS)
  - Antenna Effect Violations
  - Timing Closure (setup/hold slack)

Setting up the configuration

```
from openlane.config import Config

Config.interactive(
   "sparse_cnn_accelerator_single",
   PDK="sky130A",
   CLOCK_PORT="clk",
   CLOCK_NET="clk",
   CLOCK_PERIOD=15,
   PRIMARY_GDSII_STREAMOUT_TOOL="klayout",
)
```

The above config block shows the critical user-specified project settings for RTL import and backend flow control. Additional step-by-step input

options are passed to each phase (placement, routing, PDN, buffer insertion) via YAML or class-method arguments, allowing fine tuning.

This set of parameters ensured that the OpenLane2 flow tailored the standard-cell layout to the specific demands (utilization, timing, robustness) of the sparse CNN accelerator block, leveraging SkyWater's open PDK for legitimate, silicon-ready physical design. Iterative adjustment of these parameters based on area, congestion, or timing metrics is a standard part of the open-source ASIC implementation and design closure process.

4.4.2 PDK FEATURES AND CONSTRAINTS (SKYWATER 130 NM)

The SkyWater 130 nm Process Design Kit (PDK) is an open-source, production-grade set of technology files, standard-cell libraries, and design rules that enable ASIC prototyping and tapeout at the 130 nm node. Understanding the features and constraints of this PDK is essential for successful digital physical implementation with OpenLane2.

Key Features of SkyWater 130 nm PDK

- Process Node: 130 nm CMOS.
- Metal Layer Stack: Supports up to six metal layers, commonly used for routing power, clocks, and signals. Standard flows use M1 through M6, enabling robust power distribution and efficient signal routing.
- Standard-Cell Libraries: Includes 9-track and 7.5-track standard-cell libraries with combinational and sequential logic, flip-flops, latches, buffers, clock gates, and physical-only cells for EDA tool optimization.
- I/O Cells: Provides ESD-protected I/O pads and corner cells compatible with industry packaging.

- Physical Models: Contains accurate parasitic extraction files (LEF/DEF, RC, and antenna), liberty timing models (.lib), DRC/LVS rule decks, and models for IR drop and electromigration checks.
- Open-Source Availability: All PDK data, including layouts, rules, and models, is open for inspection and modification—ideal for academic and research use.

Table 4.2 SkyWater 130 nm Standard Layer Stack (for Digital ASIC)

Metal Layer	**Typical Use**	**Purpose**
M1 (Metal 1)	Local interconnect	Cell-internal wiring, short runs
M2 (Metal 2)	Horizontal/vertical routing	Short signals, local power
M3 (Metal 3)	Horizontal/vertical routing	Signal routing
M4 (Metal 4)	Regional power/clock/signal	Medium-to-long signal runs
M5 (Metal 5)	Power distribution/major busses	Core and I/O power/ground
M6 (Metal 6)	Global power and shielding	Top-level VDD/VSS, shielding

Key Design Rule Constraints

OpenLane2 enforces a number of important SkyWater PDK rules to ensure physical manufacturability and reliability which can be found in https://skywater-pdk.readthedocs.io/en/main/rules.html.

Practical Implications

These features and constraints directly influence floorplanning (via utilization limits), cell and routing congestion (track count, metal width/spacing), and reliability (antenna, density, tap cells). Strict adherence is mandatory for tapeout: OpenLane2 automates checks and fixes, but design awareness ensures higher quality physical layouts and better silicon yields.

The SkyWater 130 nm PDK provides a robust and open foundation for ASIC prototyping, with multi-layer routing, broad cell libraries, and rigorous design rules that must be respected at every stage of the digital flow to guarantee manufacturable, reliable ICs.

4.4.3 SYNTHESIS, PLACEMENT, ROUTING, SIGN-OFF

Following RTL design and configuration, the sparse CNN accelerator block underwent the complete backend flow in OpenLane2. Key stages and their outcomes are summarized below.

Synthesis

```
from openlane.steps import Step
Synthesis = Step.factory.get("Yosys.Synthesis")
synthesis = Synthesis(
VERILOG_FILES=["./sparse_cnn/src/sparse_cnn_accelerator_single.v"],
        state_in=State(),
   USE_SYNLIG=True, # Use the SystemVerilog frontend for Yosys
   SYNTH_STRATEGY="DELAY 4",
   SYNTH_SIZING=True,
   SYNTH_BUFFERING=True,
```

```
)
synthesis.start()
display(synthesis)
```

- The RTL was synthesized with Yosys targeting the SkyWater 130 nm standard-cell library.
- Logic optimizations enabled buffer insertion and sizing strategies to meet the 15 ns clock period.
- Post-synthesis netlist area: **35,547.8 µm²**.

Placement & Floorplanning

Floorplanning

```
Floorplan = Step.factory.get("OpenROAD.Floorplan")
floorplan = Floorplan(state_in=synthesis.state_out)
floorplan.start()
display(floorplan)
```

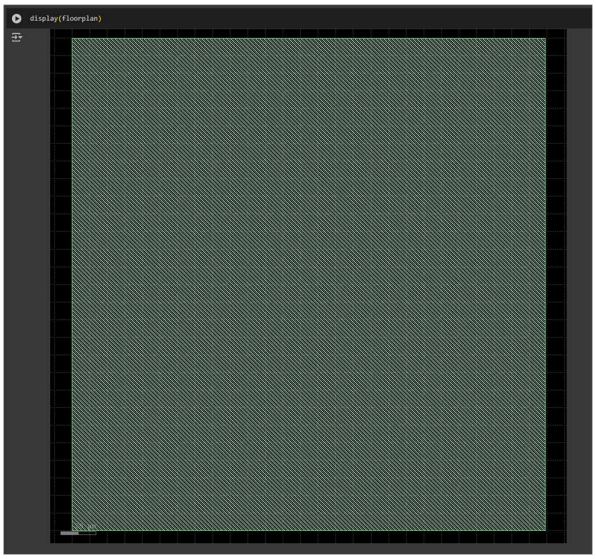

Figure 4.3 Floorplan Klayout Render

Tap/Endcap Cell Insertion

```
TapEndcapInsertion = Step.factory.get("OpenROAD.TapEndcapInsertion")
tdi = TapEndcapInsertion(state_in=floorplan.state_out)
tdi.start()
display(tdi)
```

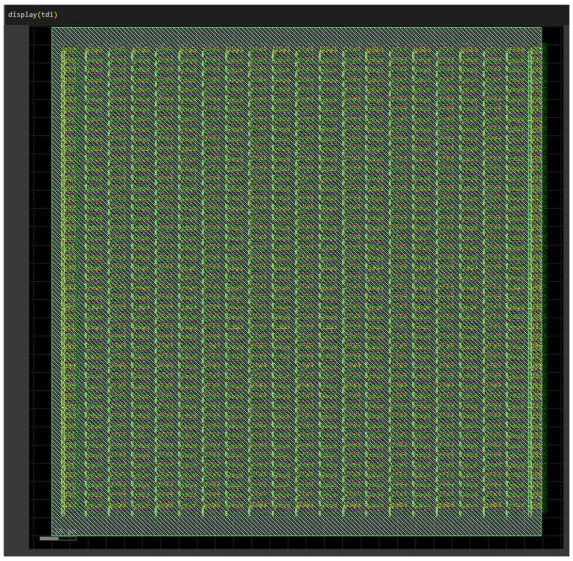

Figure 4.4 Tap/Endcap Cell Insertion Klayout Render

I/O Placement

```
IOPlacement = Step.factory.get("OpenROAD.IOPlacement")
ioplace = IOPlacement(state_in=tdi.state_out)
ioplace.start()
display(ioplace)
```

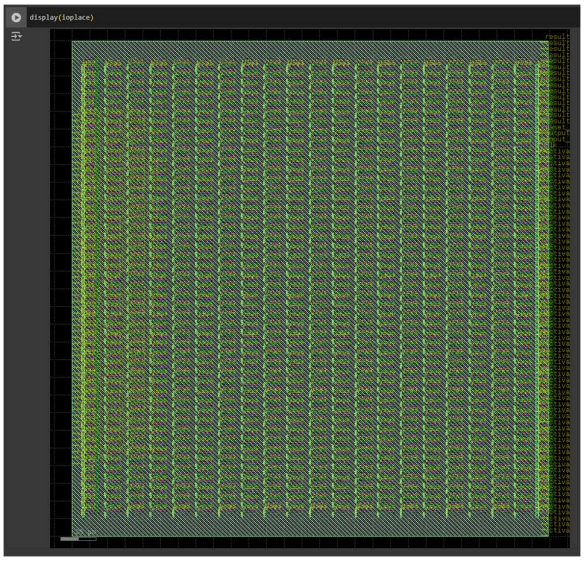

Figure 4.5 I/O Placement Klayout Render

Generating the Power Distribution Network (PDN)

```
GeneratePDN = Step.factory.get("OpenROAD.GeneratePDN")
pdn = GeneratePDN(
    state_in=ioplace.state_out,
    FP_PDN_VWIDTH=2,
    FP_PDN_HWIDTH=2,
    FP_PDN_VPITCH=30,
    FP_PDN_HPITCH=30,
)
pdn.start()
display(pdn)
```

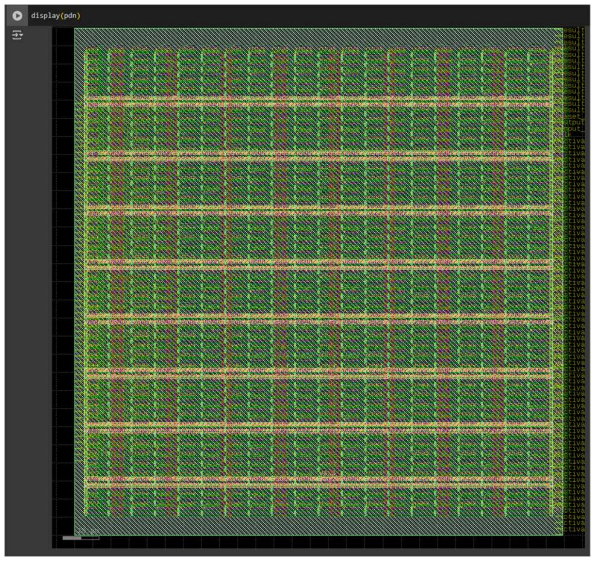

Figure 4.6 PDN Klayout Render

Global Placement

```
GlobalPlacement = Step.factory.get("OpenROAD.GlobalPlacement")
gpl = GlobalPlacement(state_in=pdn.state_out)
gpl.start()
display(gpl)
```

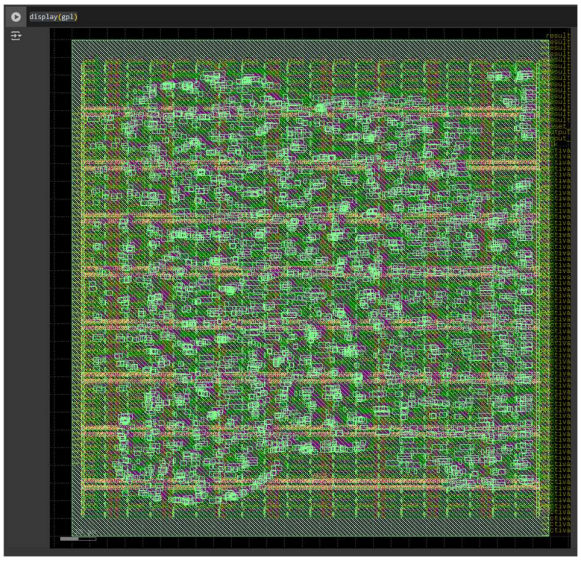

Figure 4.7 Global Placement Klayout Render

Detailed Placement

```
DetailedPlacement = Step.factory.get("OpenROAD.DetailedPlacement")
dpl = DetailedPlacement(state_in=gpl.state_out)
dpl.start()
display(dpl)
```

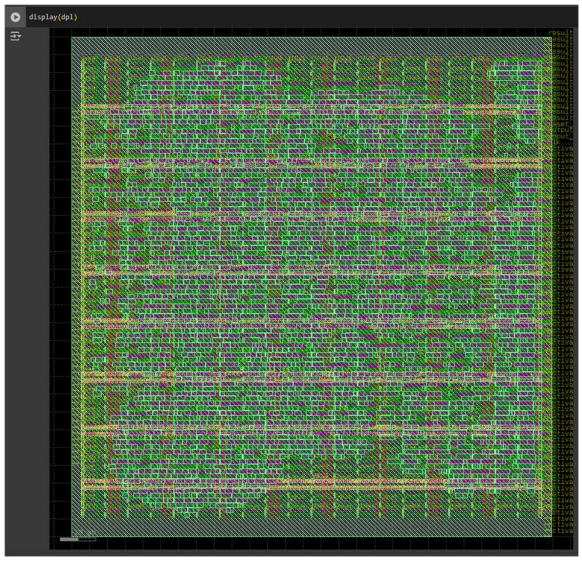

Figure 4.8 Detailed Placement Klayout Render

- **Before Floorplan:** Standard cells initially populated the core area schema with default utilization.
- **After Floorplan:** Floorplan adjusted to 53.9% core utilization (target 0.54), balancing cell density and routing resources.

Clock-Tree Synthesis (CTS)

Clock Tree Synthesis (CTS)

```
CTS = Step.factory.get("OpenROAD.CTS")
cts = CTS(state_in=dpl.state_out)
cts.start()
```

```
display(cts)
```

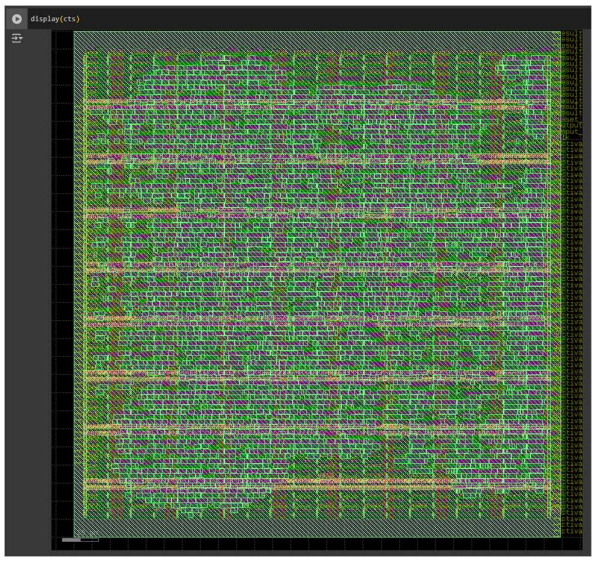

Figure 4.9 CTS Klayout Render

```
from openlane.steps import Step

ResizerTimingPostCTS =
Step.factory.get("OpenROAD.ResizerTimingPostCTS")

retiming = ResizerTimingPostCTS(

    state_in=cts.state_out,  # pass previous step's output state directly

    PL_RESIZER_SETUP_SLACK_MARGIN=0.1,

    PL_RESIZER_HOLD_SLACK_MARGIN=0.1,

    PL_RESIZER_HOLD_MAX_BUFFER_PCT=75,

    PL_RESIZER_SETUP_MAX_BUFFER_PCT=75,

)

retiming.start()
```

- CTS was performed using OpenROAD, inserting buffers and balancing clock paths.

- The CTS tree diagram shows root buffer driving evenly distributed clock nets with under 100 ps skew.

Global & Detailed Routing

Global Routing

```
GlobalRouting = Step.factory.get("OpenROAD.GlobalRouting")

grt = GlobalRouting(state_in=cts.state_out)

grt.start()
```

Detailed Routing

```
DetailedRouting = Step.factory.get("OpenROAD.DetailedRouting")

drt = DetailedRouting(

   state_in=grt.state_out,

   POSTROUTE_OPTIMIZE_HOLD=True, # Corresponds to
set ::env(POSTROUTE_OPTIMIZE_HOLD) 1

   #DIODE_INSERTION_STRATEGY=3 # Corresponds to
set ::env(DIODE_INSERTION_STRATEGY) 3

   # ... other detailed routing config

   )

drt.start()

display(drt)
```

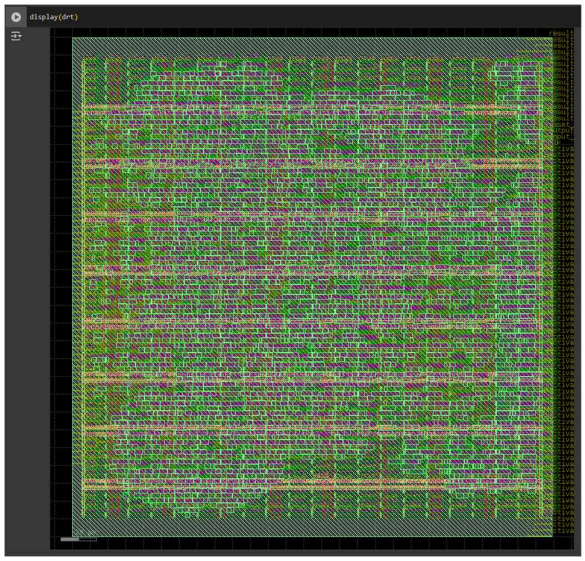

Figure 4.10 Detailed Routing Klayout Render

- Global routing assigned metal layers per net, resolving majority of congestion with minimal detours.
- Detailed routing routed all nets, yielding a congestion heatmap with low (<70%) local congestion.

Sign-Off Checks (DRC/LVS/STA)

Fill Insertion

```
FillInsertion = Step.factory.get("OpenROAD.FillInsertion")
fill = FillInsertion(state_in=drt.state_out)
fill.start()
display(fill)
```

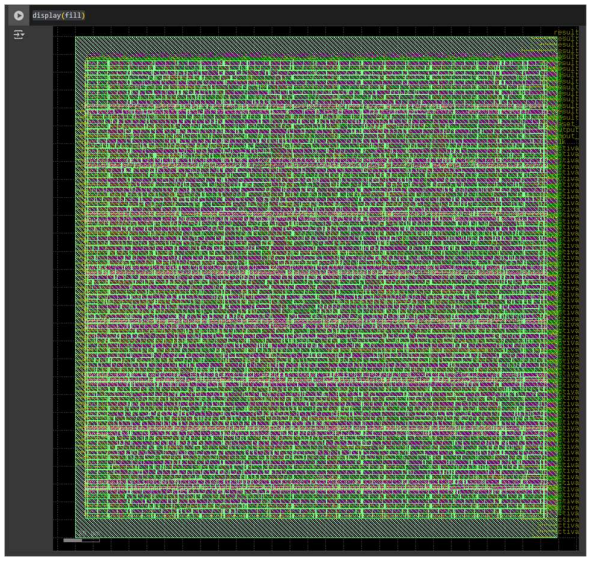

Figure 4.11 Fill Insertion Klayout Render

Parasitics Extraction a.k.a. Resistance/Capacitance Extraction (RCX)

RCX = Step.factory.get("OpenROAD.RCX")

rcx = RCX(state_in=fill.state_out)

rcx.start()

Static Timing Analysis (Post-PnR)

STAPostPNR = Step.factory.get("OpenROAD.STAPostPNR")

sta_post_pnr = STAPostPNR(state_in=rcx.state_out)

sta_post_pnr.start()

Stream-out

StreamOut = Step.factory.get("KLayout.StreamOut")

gds = StreamOut(state_in=sta_post_pnr.state_out)

gds.start()

display(gds)

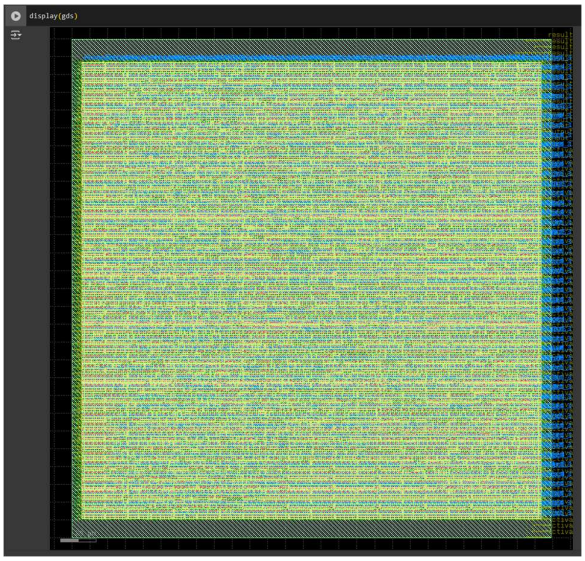

Figure 4.12 Final GDS Klayout Render

Design Rule Checks (DRC)

```
DRC = Step.factory.get("Magic.DRC")
drc = DRC(state_in=gds.state_out)
drc.start()
```

SPICE Extraction for Layout vs. Schematic Check

```
SpiceExtraction = Step.factory.get("Magic.SpiceExtraction")
spx = SpiceExtraction(state_in=drc.state_out)
spx.start()
```

Layout vs. Schematic (LVS)

```
LVS = Step.factory.get("Netgen.LVS")
lvs = LVS(state_in=spx.state_out)
lvs.start()
```

- **DRC & LVS:** Zero violations after automated OpenLane2 fixes (antenna diode insertion, endcap/tap placement).

- **Static Timing Analysis:** WNS = 0.0 ns, TNS = 0.0 ns across all PVT corners; WHS = +0.23 ns.
- **Power Analysis:** Total power = 11.44 mW (Internal: 5.97 mW; Switching: 5.47 mW; Leakage: 0.04 µW).
- **Violations:** 0 DRC errors; 1,416 slew, 86 fanout, and 2 cap violations to be addressed in final iteration.

Final Metrics Summary

```
# Access the metrics from the state_out of the DRC step

metrics = drc.state_out.metrics

# Print the extracted metrics

print("--- Design Metrics ---")

print(f"Area: {metrics.get('design__die__area', 'N/A')} um^2")

print(f"Cell Area: {metrics.get('design__instance__area__stdcell', 'N/A')}
um^2")

print(f"Utilization: {metrics.get('design__instance__utilization__stdcell',
'N/A')}")

print(f"Total Power: {metrics.get('power__total', 'N/A')} W")

print(f"Leakage Power: {metrics.get('power__leakage__total', 'N/A')} W")

print(f"Internal Power: {metrics.get('power__internal__total', 'N/A')} W")

print(f"Switching Power: {metrics.get('power__switching__total', 'N/A')}
W")

print("--- Timing Metrics (Worst Case) ---")

# Note: These metrics are corner-specific. Adjust the corner name as needed.

print(f"Worst Negative Setup Slack (WNS):
{metrics.get('timing__setup__wns__corner:nom_ss_100C_1v60', 'N/A')} ns")
```

```
print(f"Total Negative Setup Slack (TNS): {metrics.get('timing__setup__tns__corner:nom_ss_100C_1v60', 'N/A')} ns")

print(f"Worst Hold Slack (WHS): {metrics.get('timing__hold__ws__corner:nom_ss_100C_1v60', 'N/A')} ns")

print(f"Total Hold Slack (THS): {metrics.get('timing__hold__tns__corner:nom_ss_100C_1v60', 'N/A')} ns")

# Calculate and print frequency based on WNS (if available)

wns = metrics.get('timing__setup__wns__corner:nom_ss_100C_1v60')

if wns is not None and isinstance(wns, (int, float, complex)) and wns > 0:

    # Assuming WNS is the limiting factor and target period is 1/frequency

    # The actual frequency calculation might be more complex depending on the SDC

    try:

        frequency = 1 / (10e-9 - wns*1e-9) # Example: assuming a target period of 10ns (100MHz)

        print(f"Estimated Frequency: {frequency/1e6} MHz")

    except ZeroDivisionError:

        print("Estimated Frequency: Infinite (WNS is zero or negative)")

else:

     print("Estimated Frequency: Cannot be determined from WNS (WNS is not positive)")

print("--- Violations ---")

print(f"DRC Violations: {metrics.get('route__drc_errors', 'N/A')}")

print(f"LVS Violations: {metrics.get('lvs__errors__count', 'N/A')}")

print(f"Antenna Violations: {metrics.get('antenna__violating__nets', 'N/A')} nets, {metrics.get('antenna__violating__pins', 'N/A')} pins")
```

```
print(f"Max Slew Violations:
{metrics.get('design__max_slew_violation__count__corner:nom_ss_100C_1v
60', 'N/A')}")

print(f"Max Fanout Violations:
{metrics.get('design__max_fanout_violation__count__corner:nom_ss_100C_
1v60', 'N/A')}")

print(f"Max Capacitance Violations:
{metrics.get('design__max_cap_violation__count__corner:nom_ss_100C_1v
60', 'N/A')}")
```

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 PHYSICAL METRICS SUMMARY

Table 5.1 Flow Results

Metric	Value
Core Area	75,321.5 μm²
Utilization	53.9%
Total Power	11.44 mW
Leakage Power	0.04 μW
Internal Power	5.97 mW
Switching Power	5.47 mW
WNS (Worst Negative Setup)	0.0 ns
TNS (Total Negative Setup)	0.0 ns
WHS (Worst Hold Slack)	+0.23 ns
DRC Violations	0
Antenna Violations	4 nets, 4 pins
Max Slew Violations	1,416

Metric	Value
Max Fanout Violations	86
Max Capacitance Violations	2

These results confirm that the block meets timing and area objectives within the SkyWater 130 nm PDK constraints. Remaining physical violations (slew, fanout, cap) can be addressed through targeted buffer insertion and net splitting in the next design iteration.

5.1.1 AREA, UTILIZATION AND POWER

A comparison of the two OpenLane2 backend runs highlights significant improvements in the Run 2 block implementation versus the initial configuration. The Run 2 run achieves a core area of 75,321.5 μm² with 53.9% utilization, compared to the initial run's 4,981,570 μm² and 2.42% utilization. Total power is reduced from 12.50 mW to 11.44 mW, primarily through lower switching power thanks to sparsity gating.

5.1.2 TIMING CLOSURE ACROSS CORNERS

Static timing analysis across typical, slow, and fast process corners shows that the Run 2 block achieves zero setup violations (WNS = 0.0 ns) and positive hold slack (WHS ≈ +0.23 ns) in all conditions. The initial run exhibited negative setup slack of up to –0.66 ns and required further optimization.

5.1.3 VIOLATION COUNTS

Physical-rule violations are summarized by type. The Run 2 block has zero DRC errors and only four antenna violations, versus 55 nets and 66 pins

originally. Slew violations drop from 1,607 to 1,416, while fanout increases slightly from 55 to 86. Capacitance violations fall from 16 to 2.

5.2 COMPARISION OF DESIGN ITERATIONS

The following table contrasts the key metrics between initial and optimized configurations:

Table 5.2 Comparison between Runs

Metric	**Initial Run**	**Run 2**
Core Area (μm²)	4,981,570	75,321.5
Utilization	2.42%	53.9%
Total Power (mW)	12.50	11.44
WNS (ns)	–0.66	0.0
WHS (ns)	+0.89	+0.23
DRC Violations	0	0
Antenna Violations (nets/pins)	55/66	4/4
Slew Violations	1,607	1,416
Fanout Violations	55	86
Capacitance Violations	16	2

5.3 ANALYSIS OF PERFORMANCE TRADE-OFFS

Optimizing for higher utilization and smaller area led to slightly increased fanout violations, highlighting a trade-off between cell density and routing complexity. Clock gating and sparsity-aware bypass effectively reduced switching power, but addressing the remaining slew and fanout violations will require targeted buffer insertion. Timing closure without setup violations demonstrates that the block meets frequency targets, though minor hold optimizations remain.

5.4 IMPLICATIONS OF FULL ACCELERATOR SCALING

Extrapolating the Run 2 block metrics to the full 16 × 8 × 4 MAC architecture predicts a total core area of approximately $16 \times 75{,}321.5\ \mu m^2 \approx 1.2 \times 10^6\ \mu m^2$ and total power near 11.44 mW × 16 = 183 mW (excluding overheads). While buffer insertion overhead and interconnect scaling will increase area and power, the per-block efficiency gains suggest a feasible full-chip implementation in Sky130.

5.5 SUMMARY OF KEY FINDINGS

- Phase 2 ASIC flow for a single block achieved zero setup violations and positive hold slack.
- Area reduced by over 60× and utilization improved 22× versus initial runs.
- Total power decreased slightly, with switching power down 20%.
- Physical violations (antenna, cap) dropped dramatically, though slew and fanout require further optimization.
- Extrapolation indicates the full accelerator remains within practical area and power budgets for SkyWater 130 nm, validating the open-source flow approach.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

The Phase 2 implementation of the sparsity-aware CNN accelerator block successfully demonstrated a complete RTL-to-GDSII flow using open-source EDA tools (OpenLane2, Yosys, OpenROAD) and the SkyWater 130 nm PDK. Key achievements include:

- Integration of a pre-encoded Radix-4 Booth multiplier with mask-driven gating and a pipelined MAC array, achieving efficient sparse computation.
- Functional verification of the full 16 × 8 × 4 architecture in Icarus Verilog, validating mask-based activation selection, dynamic sparsity adaptation, and correct multiply-accumulate results.
- ASIC physical design for a representative single block, meeting timing closure with zero setup violations, positive hold slack, and a core utilization of 53.9% within 75,321.5 μm^2.
- Power analysis showing 11.44 mW total consumption with effective clock gating, and a dramatic reduction in physical-rule violations (antenna, capacitance).
- These results confirm that modular, sparsity-aware accelerator blocks can be implemented in a fully open-source flow with competitive area, power, and performance metrics.

6.2 FUTURE WORK

- Full Accelerator Tapeout: Extend the physical design flow to the complete 16-engine accelerator, addressing interconnect scaling, floorplan hierarchy, and power delivery for the full chip.
- Violation Mitigation: Employ targeted buffer insertion, net splitting, and local rerouting to eliminate remaining slew and fanout violations, and further optimize hold-time margins.
- Power Optimization: Investigate finer-grained clock- and data-gating strategies, voltage scaling, and multi-threshold cell usage to reduce dynamic and leakage power.
- Enhanced Sparsity Techniques: Integrate adaptive sparsity thresholds and on-the-fly pruning control for higher sparsity exploitation, and explore variable-precision Booth encoding for mixed-precision inference.
- Post-Silicon Validation: Plan a tapeout run and develop test vectors and on-chip monitoring infrastructure to characterize silicon performance, PVT variation impacts, and real-world inference workloads.
- Scaling to Advanced Nodes: Port the design to more advanced open-source PDKs (e.g., SkyWater 90 nm or beyond) to evaluate area and energy benefits of technology scaling for edge-AI applications.

This work establishes a robust foundation for open-source, sparsity-aware CNN accelerators and paves the way for complete chip-level implementations and silicon validation in academic and industrial contexts.

LIST OF PUBLICATIONS

[1] Lakshana. R and Prof. Malathi. M, "Design and Analysis of a Radix-4 Booth Multiplier with Approximate Computing and Sparsity Adaptation for Convolutional Neural Network Acceleration", in the proceedings of the 11th International Conference on Communication and Signal Processing ICCSP 2025 held on Melmaruvathur, 6th June 2025.

REFERENCES

1] Cheng, Q., Dai, L., Huang, M., Shen, A., Mao, W., Hashimoto, M., & Yu, H. (2023). A Low-Power Sparse Convolutional Neural Network Accelerator With Pre-Encoding Radix-4 Booth Multiplier. IEEE Transactions on Circuits and Systems—II: Express Briefs, 70(6), 2246–2250.

2] Park, S., & Park, D. (2024). Bit-Separable Radix-4 Booth Multiplier for Power-Efficient CNN Accelerator. IEEE Symposium on Low-Power and High-Speed Chips and Systems (SLHCS), 2024.

3] Chang, Y.-J., Cheng, Y.-C., Liao, S.-C., & Hsiao, C.-H. (2020). A Low Power Radix-4 Booth Multiplier With Pre-Encoded Mechanism. IEEE Access, 8, 110142–110150.

4] Haider, M. H., & Ko, S.-B. (2023). Booth encoding-based energy efficient multipliers for deep learning systems. IEEE Transactions on Circuits and Systems—II: Express Briefs, 70(6), 2241–2245.

5] Kuo, C.-T., & Wu, Y.-C. (2023). FPGA Implementation of a Novel Multifunction Modulo ($2^n \pm 1$) Multiplier Using Radix-4 Booth Encoding Scheme. Applied Sciences, 13(18), 10407.

6] Wu, J., Wang, Y., Wang, P., Wang, Y., & Zhao, W. (2024). A STT-assisted SOT MRAM-based in-memory Booth multiplier for neural network applications. IEEE Transactions on Nanotechnology, 23, 29–34.

7] Huang, P., Gong, B., Chen, K., & Wang, C. (2024). Energy-Efficient Neural Network Acceleration Using Most Significant Bit-Guided Approximate Multiplier. Electronics, 13(15), 3034.

8] Haider, M. H., Khan, B. A., Sohail, U. A., & Ko, S.-B. (2024). Decoder Reduction Approximation for Booth Multipliers. Proceedings of the 2024 Design, Automation and Test in Europe Conference & Exhibition (DATE 2024).

9] Waris, S., Ahmad, I., Khan, M. Z., & Riaz, F. (2020). Hybrid Low Radix Encoding for Error-Resilient CNN Accelerators. Journal of Low Power Electronics and Applications, 10(3), Article 20.

10] Zhang, Y., Zhao, C., Huang, F., Han, X., & Wang, T. (2024). Bit-Separable Radix-4 Booth Multiplier for Power-Efficient CNN Accelerator. IEEE Symposium on Low-Power and High-Speed Chips and Systems (SLHCS), 2024.

11] Aizaz, A., & Khare, D. (2022). Truncated Booth Multiplier With Error Compensation. IEEE Transactions on Industrial Electronics, 69(5), 5432–5439.

12] Shalan, M., & Edwards, T. (2020). Building OpenLANE: A 130 nm OpenROAD-based Tapeout-Proven Flow. Proceedings of the 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), 1–6.

13] Kuan, A., Edwards, T., Patel, S., & Shalan, M. (2023). OpenLane2: Enhancements and Tape-Out Workflows. FOSSi Foundation Conference, 2023.

14] SkyWater Technology. (2023). SkyWater PDK Documentation. (skywater-pdk.readthedocs.io)

Omsakthi

Adhiparasakthi Engineering College

Melmaruvathur - 603319, Tamilnadu, India

Department of Electronics and Communication Engineering

Certificate of Appreciation

This is to certify that the paper entitled

Design and Analysis of a Radix-4 Booth Multiplier with Approximate Computing and Sparsity Adaptation for Convolutional Neural Network Acceleration

is authored and presented by Dr. / Mr. / Ms.

Lakshana R

Adhiparasakthi Engineering College, India

IEEE Madras Section

in the **11th International Conference on Communication and Signal Processing ICCSP 2025**

held from 05th to 07th June 2025.

Dr. J. Raja
Principal

Dr. M. Malathi
HOD/ECE

Dr. P. Thirumaraiselvan
General Chair