Technical Report

Matrix Multiplication Accelerator

RTL Design, Simulation, Throughput and ASIC Synthesis Results

---

Team Members

Hammad Safeer

Jan Hofmann

February 2, 2026

Contents

Abstract

With the increasing usage of artificial intelligence and especially large language models, the challenge of performing large and energy efficient matrix multiplications has become a topic of interest. ASICs are well known to allow for high-speed, energy-effiecient implementations of algorithms. Because of this, it was our task to design an IC which is capable of multiplying a 4x8 and a 8x4 matrix together, resulting in an matrix multiplication accelerator, which this report is written about.

This report gives a quick userguide, followed by an in-depth explanation of the impemented design as well as showing the results of multiple synthesis runs.

1 Usage

The accelerator has three basic interactions: **Reset**, **Calculate Matix** and **Read Result**.

Reset

To perform a reset follow the steps given below.

- Set **rst**=1

Note that a reset also stops any running calculations, but does not delete the contents stored in the RAM.

Calculate Matrix

- Ensure that accelerator is either reset or not currently working
- Set **ram_slot** to indicate where in the RAM the result should be written to (possible values are 0 to 31)
- Set **start**=1
- During the **start**=1 cycle as well as the following 31 cycles provide the input matrix columnwise via the **in_data** port

After the input data is provided the accelerator will start calculation automatically. Once it is finished the **finish** signal will be set to 1. In the next cycle the accelerator is ready to calculate anoter Matrix.

Read Result

The result of a calculation is not returned immediately, but stored in an internal block of RAM instead. In order to read the result, perform the following steps.

- Ensure that accelerator is not currently working
- Set **ram_slot** to indicate which matrix you want to read from
- Set **start**=0
- Set **read**=1

In the cycle after **read** is set, the first half of the first value will be retured via the **out_data** port. The values of the Matrix are provided columnwise, with each value taking two cycles to return. For each value the lower (LSB) half will be provided first, followed by the upper (MSB) half in the next cycle. The total readout lasts 32 cycles.

2 Design Process

Before any verilog-code was programmed a design phase happened, during which the exact algorithm was decided upon. Several restrictions influenced the design process, such as the number of multipliers available, the width of input and output ports or the required ordering of the output matrix. Afterwards, an ASMD (see figure 2) as well schematic block diagram of the accelerator (see figure 1) was created. To verify the the functionality of the ASMD, its behaviour was "simulated" on paper, with some quick MATLAB checks for good measure.

The implementation orients itself on the schematic in figure 2, whith each block being turned into its own module (The RAM-Module was already implemented, and used as provided). Once all submodules were implemented, they were connected together in a larger top-module.

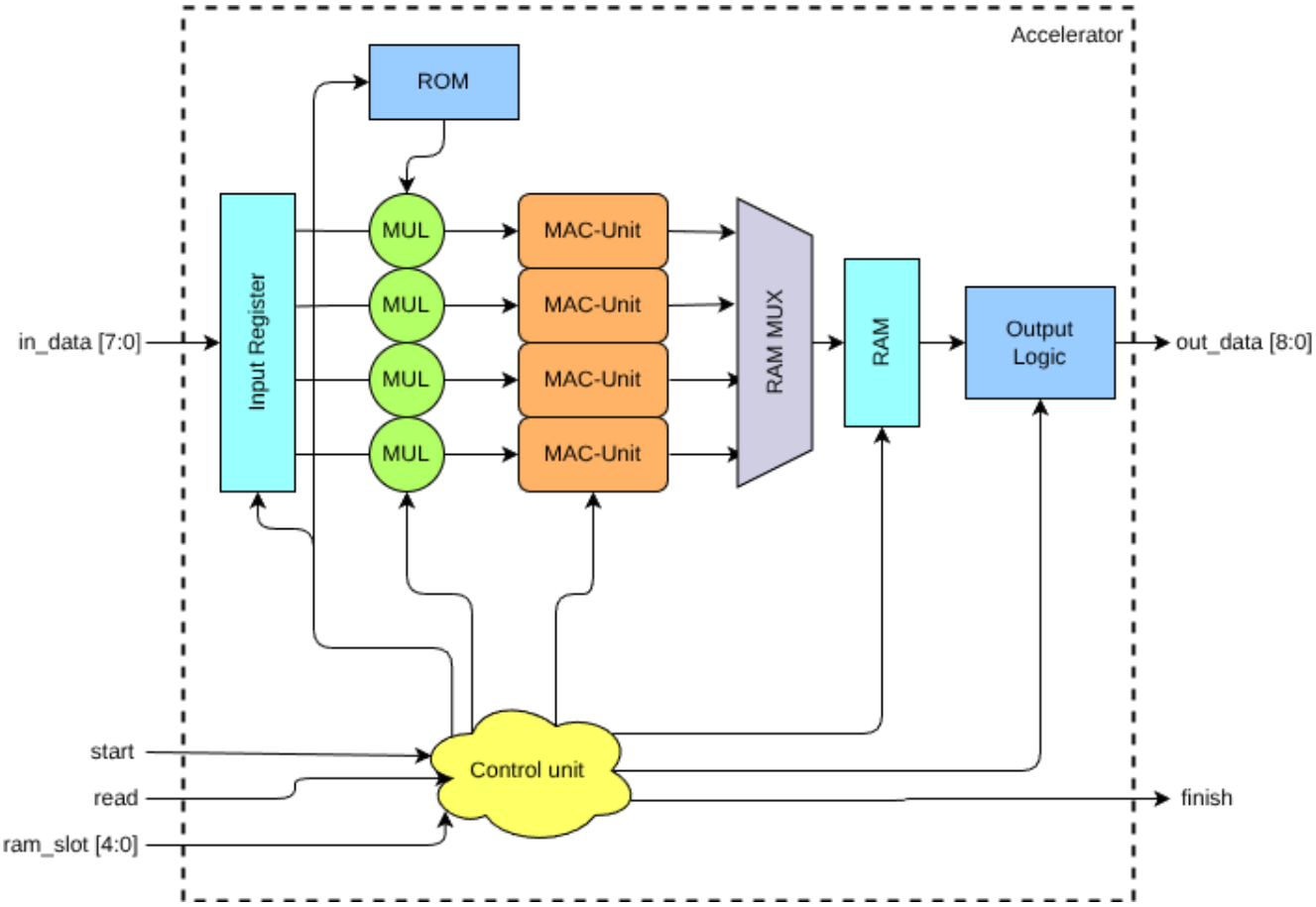

Figure 1: Architecture of the accelerator

The implemented top-module was then simulated and any occuring bugs were fixed. Simulation was done in EDA-playground as well as Xilinx Vivado. The correctness of the simulation was verified both visually, through observation of the generated waveform files, and automatically through the testbench comparing returned values with expected values provided through a MATLAB script.

3 Architecture

In order to allow for better maintainability, as well as the potential for future updates, a modular architecture was chosen, which can be seen in figure 1.

The following list contains all modules, as well as the name of the file they are located in.

- Top Module: top_file.sv
- Control Unit: calc_asmd.sv
- Input Register: ireg.sv
- Multipliers: mul.sv
- MAC-Unit: mac_unit.sv
- RAM write logic: ram_mux.sv
- RAM: RM_IHPSG13_1P_512x32_c2_bm_bist.v
- Output Logic: output_logic.sv
- ROM: rom.sv

Control Unit

The **Control Unit** implements the ASMD given in figure 2. Starting in the **idle** state, the user can either indicate a matrix-calculation via the **start** signal or read out a matrix by setting **read**. In addition to this the **ram_slot** must be provided, which is a number between 0 and 31 and indicates where in the RAM the result will be stored/is read from. After **start** is set the ASMD moves into the **READ INPUT** state, which is left once all 32 values of the input matrix are stored in the input register. Afterwards the first column will be calculated in the **CALC COLUMN** and written to RAM in the **write column**. This process repeats a total of for times (once for each column of the output matrix), before the **finish** flag is set, and the ASMD returns to **IDLE**. If **read** is set the ASMD starts to toggle between the **READ FIRST** and **READ SECOND** state, which return the lower (LSB) and upper (MSB) half of one output value. This switching repeats 16 times, once for each value of the output matrix.

Input Register

Holds the input matrix. In total the register file consists of 32 8-Bit registers, with one write and 4 read ports (one for each multiplier)

Multipliers and Adders

The name MAC-Unit is technically wrong, it only performs the accumulation part of "multiply and accumulate". The reason for this is that having external multipliers allows for easier exchanging the multipliers if a faster or cheaper multiplier is used. In addition it allows for inserting a pipeline stage in the future, which splits the long critical path from input register to multipliers to adder to internal buffer register, and thereby allows for higher clockspeeds.

RAM Write Logic

Since the RAM has only one port, the four results of the MAC-Units must be multiplexed. This happens during the **WRITE COLUMN** state.

Output Logic

This module is needed during the readout of a matrix. To safe on the number of pins, the **out_data** port is only 9 Bits wide instead of the 18 Bits needed to return a full value of the output matrix. It is this modules job to split the values before returning them one after the other.

RAM and ROM

The RAM is used to store results calculated with the help of the coefficients stored in the ROM. The RAM consists of 512 32-Bit lines. With one line per value being used, this means that the RAM can store up to 32 matrices. It is the users job to tell the accelerator which part of the RAM should be used for storage/readout via the **ram_slot** port.

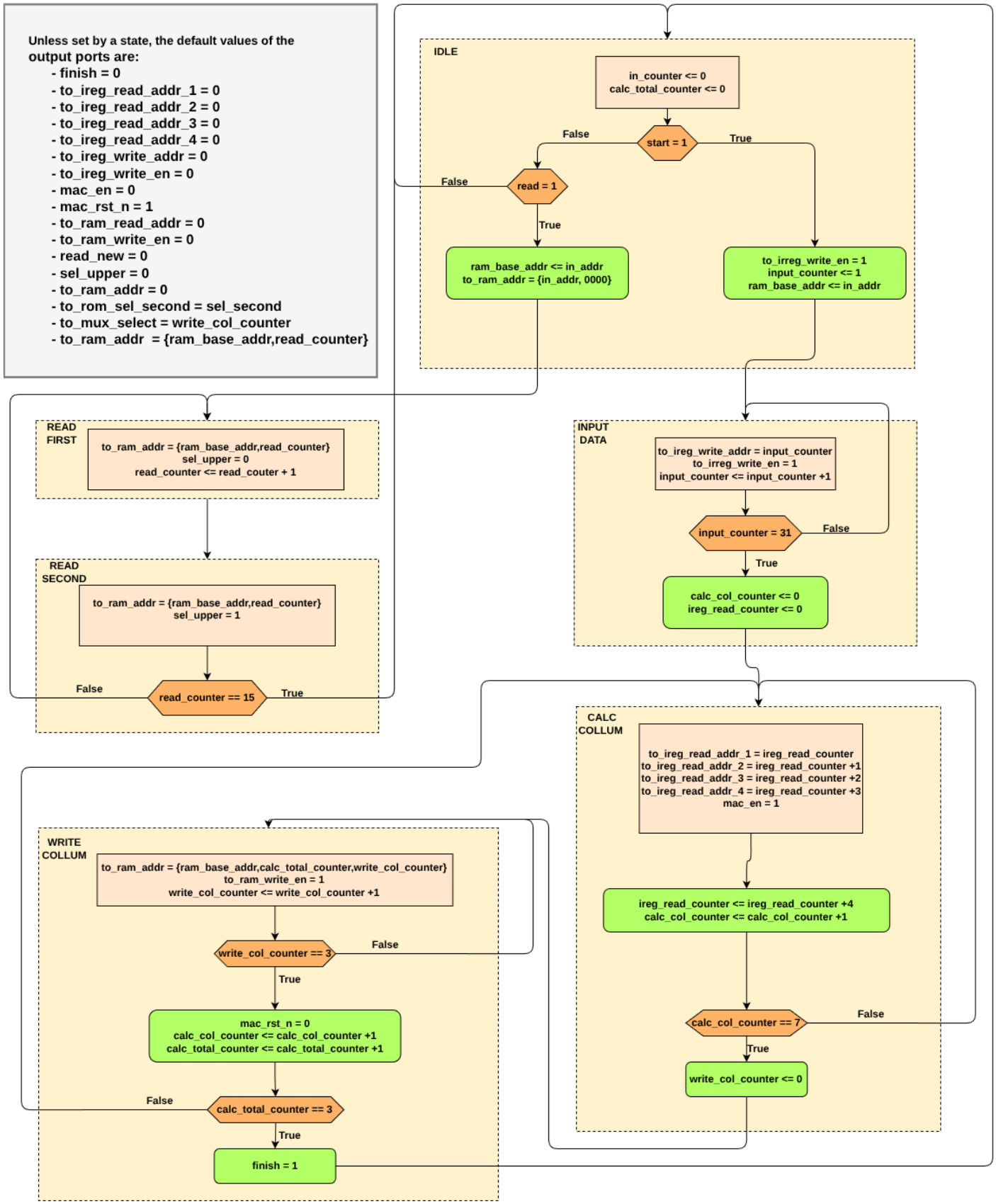

Figure 2: ASMD that controls the accelerator

4 Throughput and Critical Path

Throughput Step	**Description**	**Cycles**
Load Data	32 cycles are required to load 32 input values into the accelerator.	32
Calculate Multiplications and Accumulate	It takes 8 clock cycles to compute 4 output values in parallel. Therefore, producing all 16 output elements requires 4 iterations, i.e., **32 clock cycles in total**.	32
Multiplexing Write to RAM	4 cycles are required per write operation, repeated 4 times, resulting in a total of 16 cycles.	16
RAM Read Output	Each value takes 2 cycles to be returned. With 16 output elements, a total of 32 cycles is needed for output readout.	32
Total Cycles without Read	$32 + 32 + 16 = 80$	**80**
Total Cycles with Read	$32 + 32 + 16 + 32 = 112$	**112**

Table 1: Throughput breakdown and total cycle count for matrix computation and readout.

5 Synthesis Results

During ASIC synthesis, we evaluated different optimization objectives, including **speed**, **balanced**, and **area** modes, to see whether the tool could significantly improve area or utilization.The overall synthesized design structure, as generated by the Yosys `show` command, is illustrated in Figure 3. The results remained nearly identical across all modes when using the same standard cell library. Both total area and sequential element count showed only negligible variation, indicating limited optimization potential through logic restructuring.

This is mainly because the design is strongly **memory dominated**. The SRAM macro occupies about 79,700 μm^2, contributing more than 76% of the total chip area, while standard-cell logic accounts for only around 24%. Therefore, synthesis-level logic optimizations have little influence on the overall result.

To achieve more noticeable differences, we synthesized the design with different standard cell libraries for **slow**, **typical**, and **fast** process corners. Unlike optimization flags, changing the library affected delay and area trade-offs, leading to more meaningful variations in synthesis outcomes. An area comparison across the three optimization modes using different standard cell libraries is shown in Figure 4.

Optimization Mode	**Library**	**Standard Cell File**
Speed	Std cell Lib 1	sg13g2_stdcell_slow_1p35V_125C.lib
Balanced	Std cell Lib 2	sg13g2_stdcell_typ_1p20V_25C.lib
Area	Std cell Lib 3	g13g2_stdcell_fast_1p32V_m40C.lib

Table 2: Standard cell libraries used for synthesis under different optimization objectives.

Optimization	**Std-cell area (μm^2)**	**Total area (μm^2)**	**Sequential area (μm^2)**	**Seq %**	**Total cells**
Speed	24300.52	104020.42	5437.76	5.23	2022
Balanced	24155.45	103875.35	5437.76	5.23	2011
Area	24155.45	103875.35	5437.76	5.23	2011

Table 3: ASIC synthesis results for different optimization modes.

Cell group	**Speed**	**Balanced**	**Area**
XOR + XNOR	337 (135 + 202)	337 (131 + 206)	337 (131 + 206)
NAND (all)	408	383	383
NOR (all)	196	226	226
AND (all)	169	163	169
OR (all)	116	104	116
MUX2	20	18	18
INV	86	85	86

Table 4: Arithmetic-relevant standard cell comparison across all three synthesis runs.

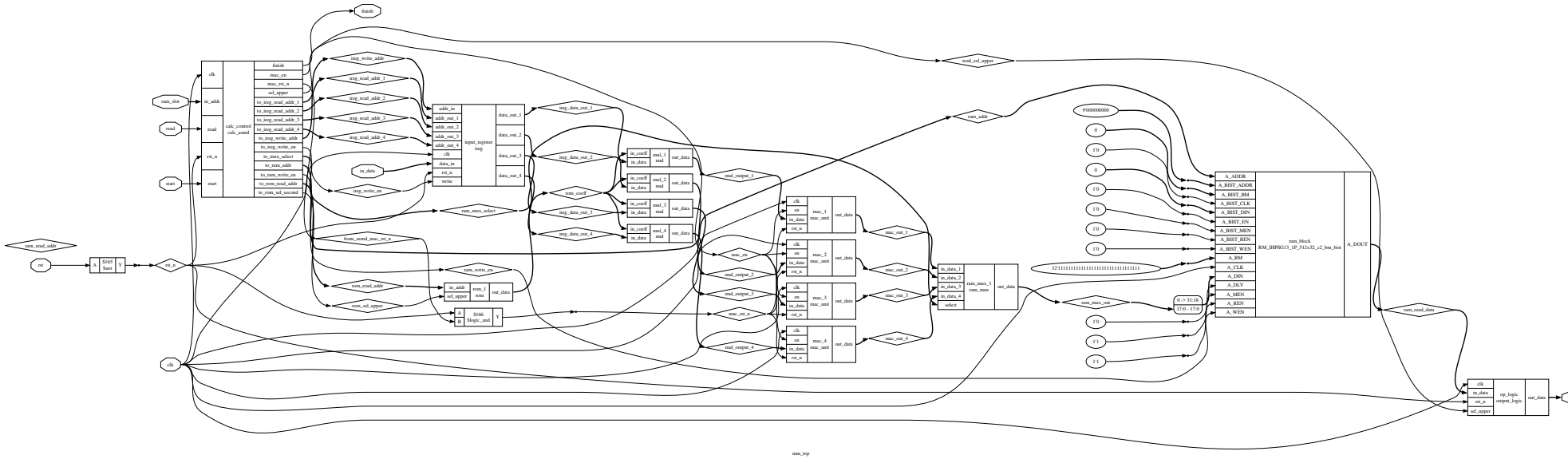

Figure 3: Synthesis resulted design.

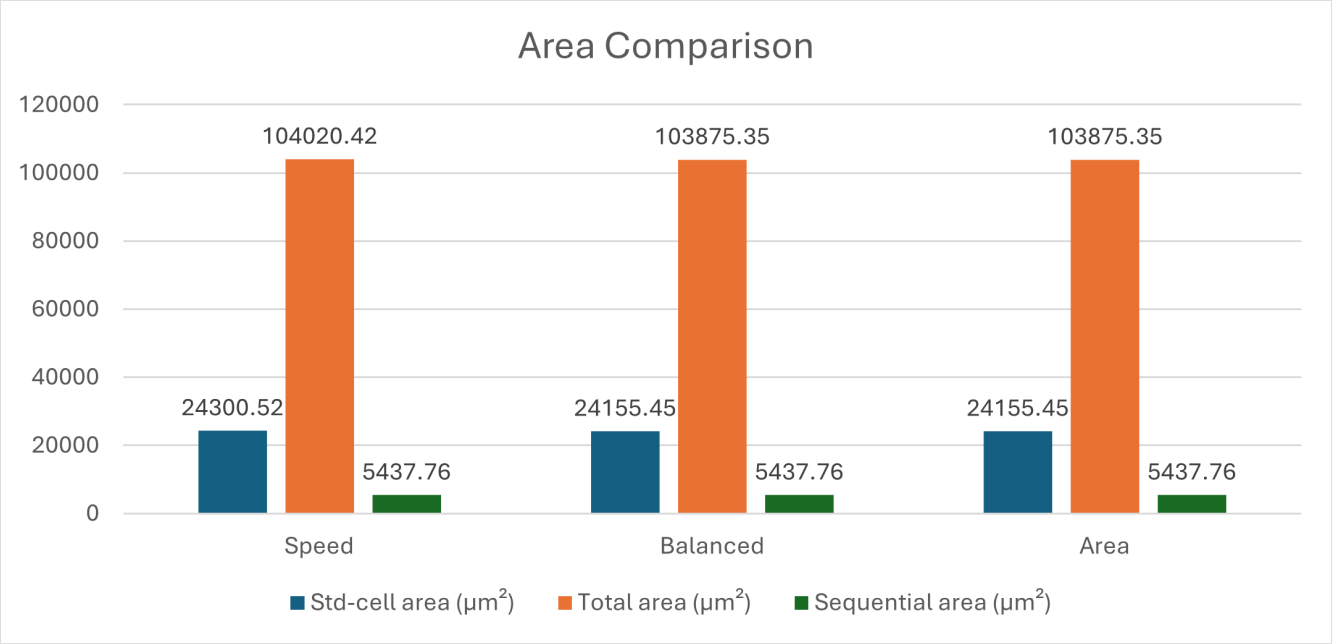

Figure 4: Area comparison across different standard cell libraries.

Figure 5 illustrates the standard-cell composition across the three synthesis runs (speed, balanced, and area). The plots highlight that arithmetic-related cells such as XOR/XNOR and NAND dominate the logic utilization, while differences between optimization modes remain small.

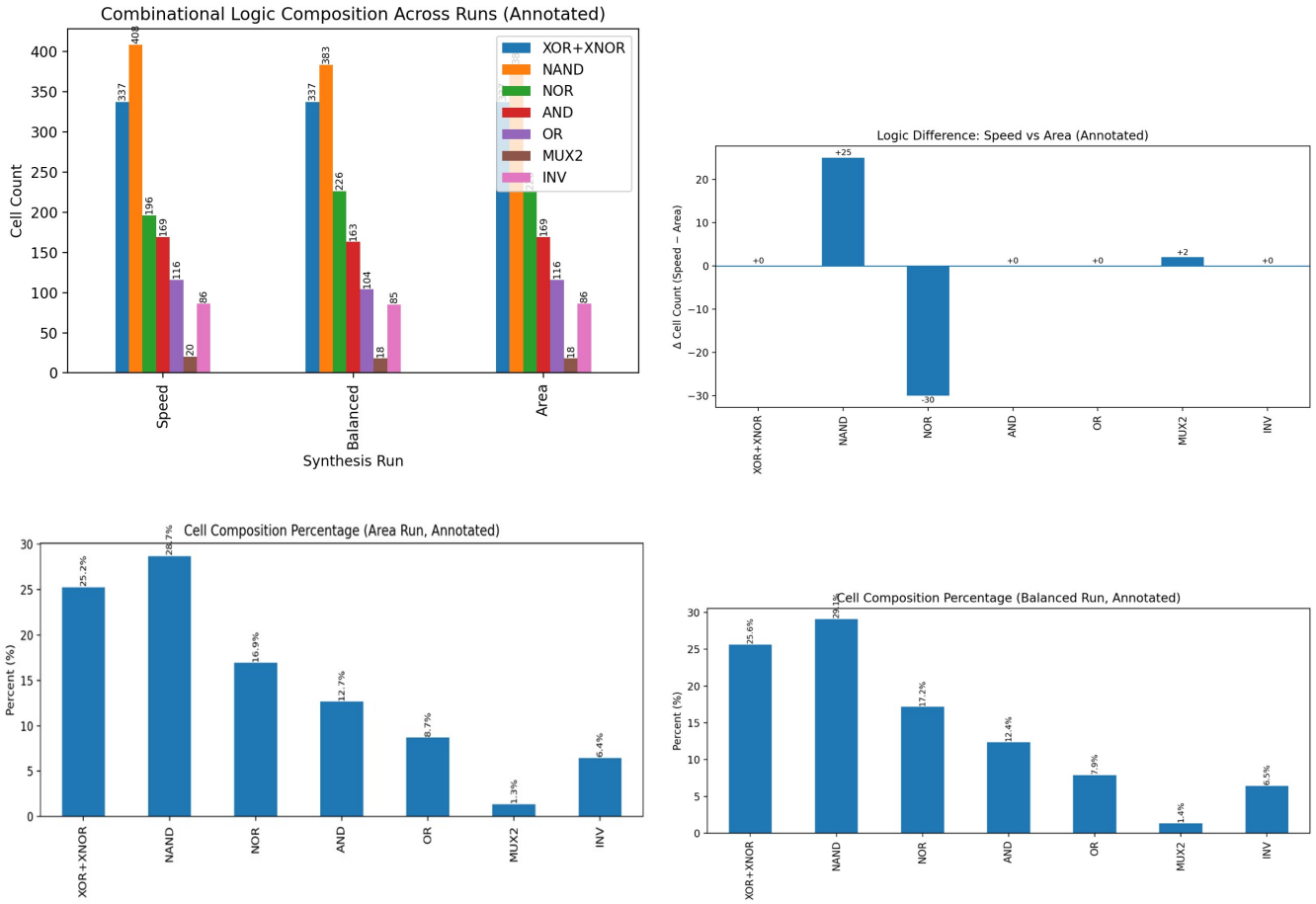

Figure 5: Synthesis result reports for different optimization modes and libraries.

A Appendix

Appendix A: Testbench

The SystemVerilog testbench used for functional verification is provided in:

- tb.sv

Appendix B: Top Module for Matrix Multiplication

The top-level module of the accelerator design is included in:

- top_file.sv

Appendix C: Other Modules for Matrix Multiplication

The remaining submodules used in the accelerator implementation are listed below:

- calc_asmd.sv
- ireg.sv
- mac_unit.sv
- mul.sv
- output_logic.sv
- ram_mux.sv
- read_asmd.sv
- rom.sv

Appendix D: RAM Modules

The two RAM Modules required for the design are as follows:

- RM_IHPSG13_1P_512x32_c2_bm_bist.v
- RM_IHPSG13_1P_core_behavioral_bm_bist.v

Appendix E: Synthesized Design Netlists

The synthesized gate-level netlists generated under different optimization objectives are:

- netlist_speed.v
- netlist_balanced.v
- netlist_area.v

Appendix F: Yosys Script

The Yosys synthesis script used for multi-run synthesis is provided in:

- multirun.ys