4×4 CMOS Array Multiplier

Krishna Karthikeya Chemudupati
Adithya Selvakumar

November 13, 2025

Contents

1 Introduction and Background

Digital multipliers are essential building blocks in modern computational systems. They appear in nearly every performance–critical arithmetic datapath, including ALUs, DSP engines, FIR filters, graphics pipelines, image–processing accelerators, and machine–learning MAC arrays. As technology scales, the system–level cost of multiplication has increased due to rising throughput demands, tighter power budgets, and shrinking timing margins. Even small improvements in the delay or energy of a single multiplier cell can accumulate into substantial architectural savings when these cells are instantiated hundreds or thousands of times in larger designs.

Within this context, small fixed–point array multipliers (such as the 4×4 structure studied here) serve as an ideal pedagogical and practical platform for understanding transistor–level behavior. Their regular, bit–sliced structure exposes fundamental CMOS effects—fanout loading, logic effort, signal alignment, glitch generation, and both dynamic and leakage power—which often remain hidden in behavioral or gate–level abstractions. By working directly at the schematic and transistor level, the designer is able to observe how real device physics shapes delay and power, making array multipliers a clean sandbox for exploring optimization techniques.

The Braun array architecture used in this project is one of the most classical and structurally uniform multiplier topologies. The array multiplier has a predictable grid of partial–product AND gates followed by rows of half and full adders. Carries flow along diagonal paths while sums propagate horizontally, making critical paths easy to trace analytically. This regularity allows for direct application of structured optimization methods such as Elmore–delay modeling and transistor sizing, with architectural effects such as glitching and partial–product races emerging naturally from the unequal arrival times of internal signals.

The purpose of this project is not merely to design a correct 4×4 multiplier, but to treat the multiplier as a vehicle for studying the full design–measure–optimize workflow used in high–performance CMOS arithmetic design. We first construct a baseline array multiplier using minimum–sized static–CMOS gates, verify its correctness with exhaustive transient simulation, and characterize three key metrics:

- worst–case propagation delay,
- active (dynamic) switching energy, and
- leakage energy.

These measurements serve as the reference against which all optimizations are compared. The optimization process then proceeds along two major axes. First, we replace the baseline "bag–of–gates" full adder with a compact complementary static–CMOS full adder that reduces internal logic depth and alleviates excessive fanout through the XOR/majority chain. This directly targets the diagonal carry–propagation staircase that dominates the critical path of the baseline array. Second, we analytically size the gates along this critical path using an Elmore–delay RC model. The analytical solution provides continuous sizing factors, revealing the ideal ratio between NAND, inverter, and XOR stages. These continuous values are then rounded to the nearest integer transistor widths and implemented in the

actual schematic, after which the optimized design is re–evaluated under the exact same testbenches as the baseline.

The broader significance of this study extends well beyond a single 4×4 multiplier. The methodology applied here—critical path identification, load–aware sizing, delay balancing, glitch–energy mitigation, and structured performance characterization—forms the backbone of modern VLSI implementation for datapath circuits. While real industrial processors operate at different scales, voltages, device technologies, and arithmetic widths, the same transistor–level principles govern their behavior. This project therefore provides a compact, controlled environment in which to explore the fundamental techniques that underlie high–performance digital integrated circuit design.

2 Baseline Design

The baseline implementation is a fully static-CMOS 4×4 Braun (array) multiplier constructed hierarchically from *minimum-sized* logic gates in the 45 nm, 1.2 V process specified in the project handout. In this baseline, every MOSFET (NMOS and PMOS) uses the minimum width and minimum channel length allowed by the PDK; no cell-level upsizing or skew tuning is performed.

At the top level, the multiplier is built from:

- sixteen 2-input AND gates that generate the partial products $X_i Y_j$,
- a grid of half adders (HAs) and full adders (FAs) that sum the partial products, and
- wiring that routes carries downward and sums leftward, as in a standard Braun array.

All HAs and FAs are themselves built from library-style XOR, NAND, NOR, AND, and inverter cells, each implemented as a minimum-size static-CMOS gate. The overall structure and the corresponding schematics are shown in the figures below.

2.1 Array-Level Architecture

Figure 1 shows the gate-level structure of the baseline 4×4 multiplier. The multiplicand bits X_0–X_3 are placed along the top of the array, and the multiplier bits Y_0–Y_3 enter from the right. Each AND gate at the top row computes a partial product $X_i Y_j$, and these partial products feed into the HA/FA array below.

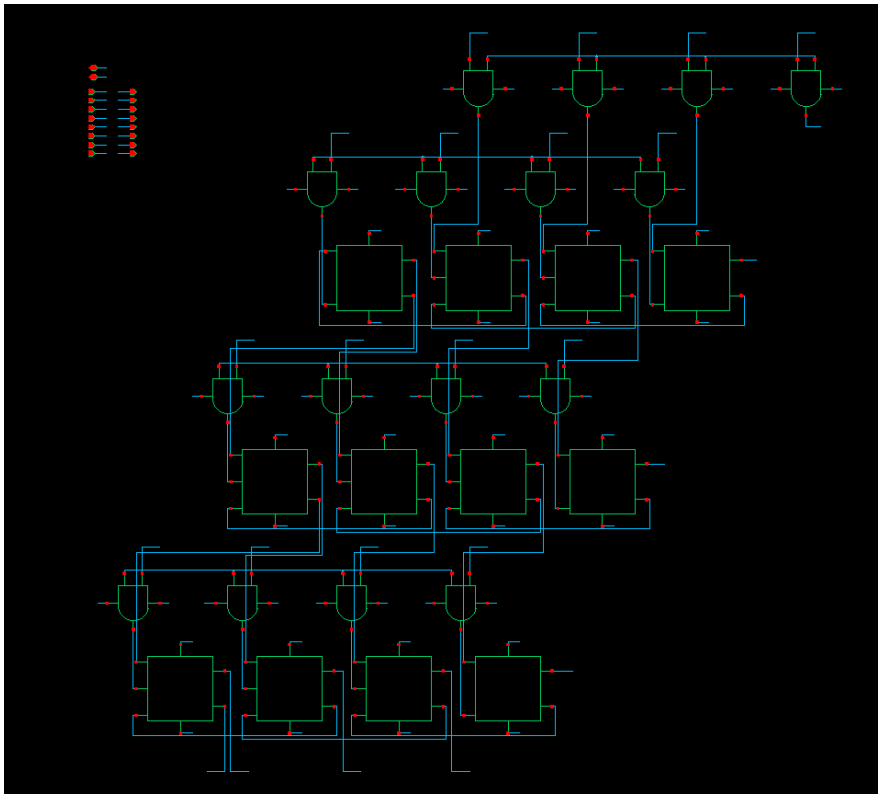


Figure 1: Gate-level structure of the baseline 4×4 Braun array multiplier. Partial products X_iY_j are generated by AND gates and reduced by a regular grid of full adders (FA) and half adders (HA) to produce the outputs Z_0–Z_7.

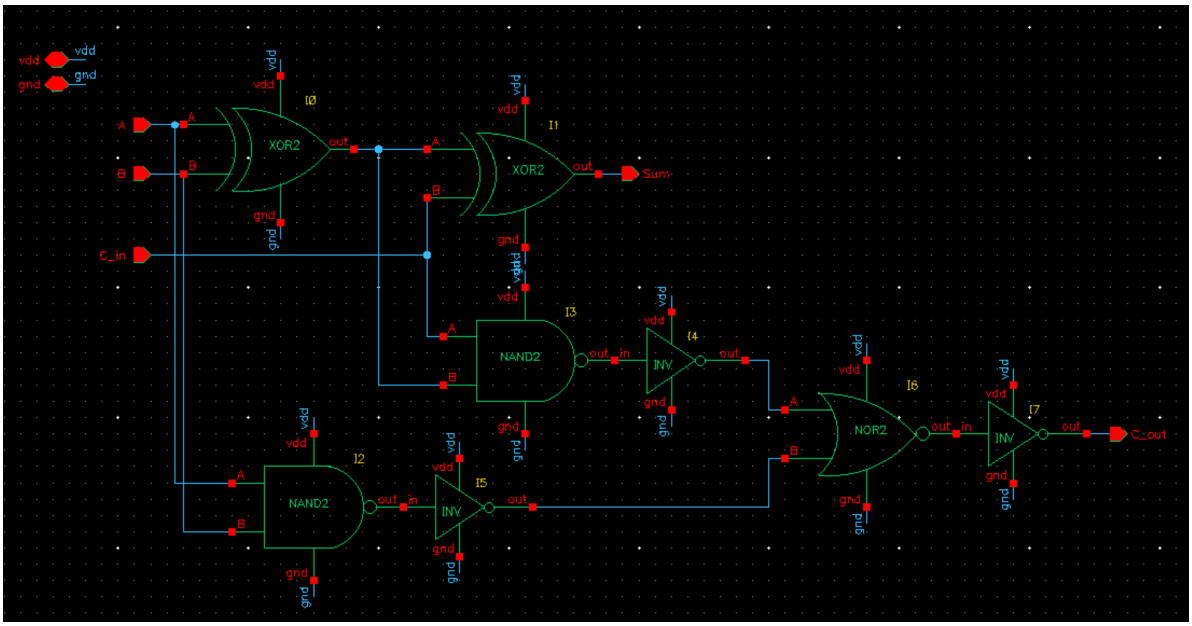
The least significant output bit Z_0 is simply the partial product X_0Y_0. Higher-order bits are formed by accumulating multiple partial products and carries:

- Column Z_1 adds X_1Y_0 and X_0Y_1 with a half adder.
- Intermediate columns (e.g., Z_2, Z_3, Z_4) combine three or more terms using one HA at the top followed by FAs underneath.
- The most significant bits Z_6 and Z_7 are formed at the bottom-left corner by the last FA and its carry-out.

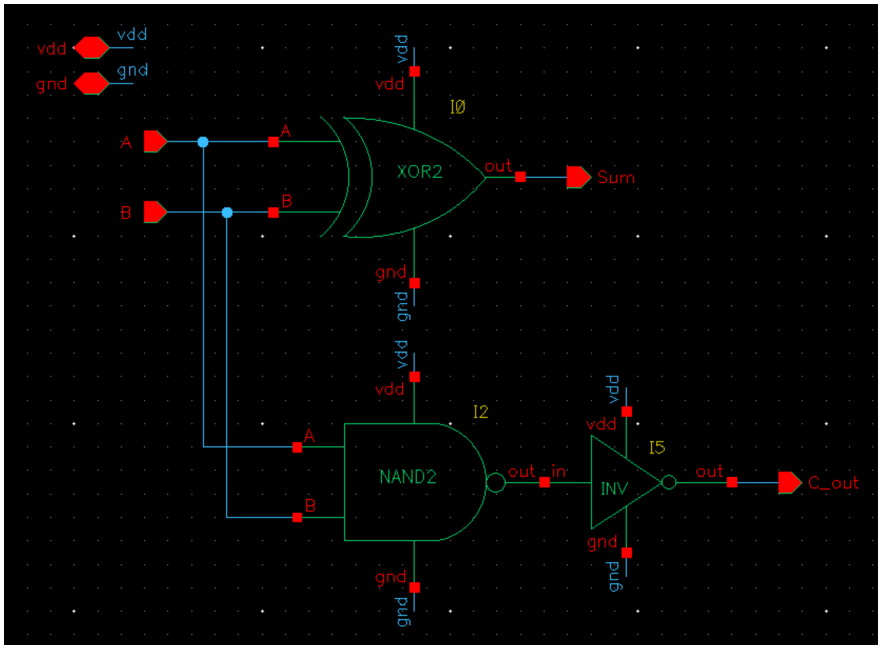
Within each column, carries propagate *downward* from one adder to the next; between columns, sums propagate *leftward*. This produces the characteristic northeast-to-southwest "staircase" critical paths (CP_1, CP_2) used in the worst-case delay analysis.

2.2 Baseline Adder Cells

The baseline array uses gate-assembled HAs and FAs. Their schematics are shown in Fig. 2. Every gate instance in these schematics is built from minimum-size transistors; there is no sizing difference between baseline and optimized implementations at this level.



(a) Baseline gate-level full adder (FA). The Sum path is implemented with two cascaded XOR2 cells; the Carry path uses NAND2, INV, and NOR2 stages followed by a final inverter.



(b) Baseline gate-level half adder (HA) built from an XOR2 (Sum) and a NAND2 followed by an inverter (Carry).

Figure 2: Adder cells used in the baseline 4×4 array multiplier.

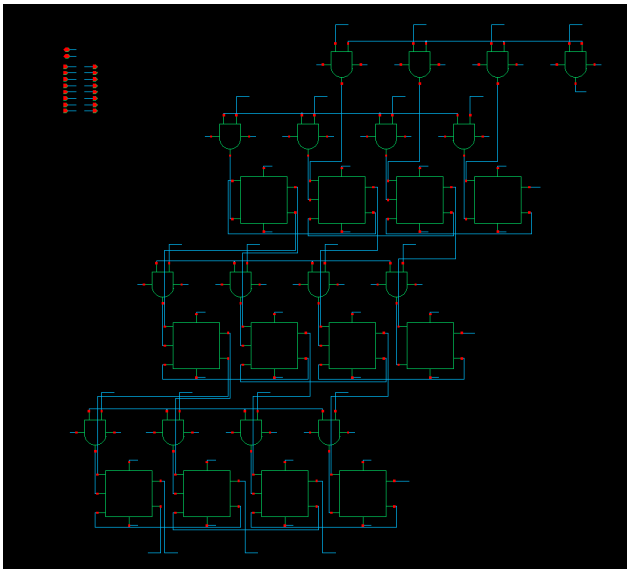


Figure 3

Full adder

The full adder implements the usual three-input addition:

$$S = A \oplus B \oplus C_{\text{in}}, \qquad C_{\text{out}} = AB + C_{\text{in}}(A \oplus B).$$

In the baseline schematic (Fig. 2a) this is realized as:

- **Sum path**: two cascaded XOR2 gates. The first XOR computes $A \oplus B$; the second XOR combines this result with C_{in} to generate S.
- **Carry path**: a small network of NAND2, INV, and NOR2 gates. A NAND2 followed by an inverter generates AB. In parallel, $A \oplus B$ and C_{in} are combined so that the final NOR2+INV sequence implements $C_{\text{out}} = AB + C_{\text{in}}(A \oplus B)$.

This "bag-of-gates" implementation is conceptually simple and easy to map from the Boolean expressions, but it introduces multiple logic stages on both the Sum and Carry paths. In particular, the Carry output passes through several cascaded gates (NAND2 $\rightarrow$ INV $\rightarrow$ NOR2 $\rightarrow$ INV), which contributes to the relatively large delay of the baseline design along the critical staircases of the array.

Half adder

The half adder computes

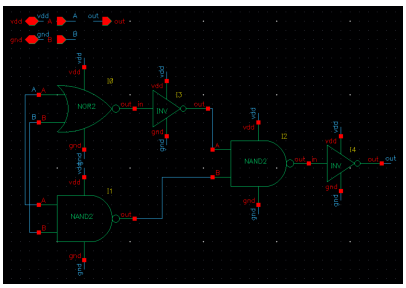
$$\text{Sum} = A \oplus B, \qquad \text{Carry} = A \cdot B.$$

As shown in Fig. 2b, the Sum output is generated directly by a single XOR2 gate, while the Carry output is realized as a NAND2 followed by an inverter (Carry $= \overline{\text{NAND}(A, B)}$).

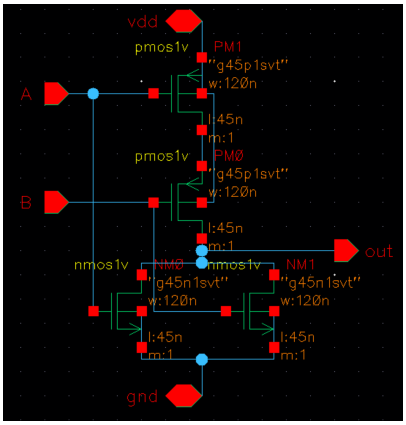
This two-gate sequence yields a static-CMOS implementation of the AND function. Compared to the full adder, the HA has shallower logic depth, which is why it is placed at the top of each column, where it only needs to add two partial products and does not receive a carry input.

2.3 Primitive CMOS Gate Implementations

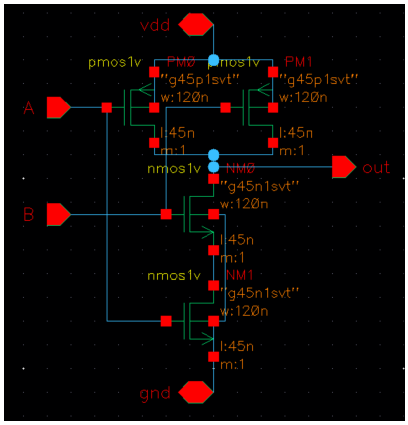
All logic gates in the baseline adder cells are implemented as transistor-level static-CMOS cells using *minimum-size* devices. Figure 4 shows the schematics of the XOR2, NOR2, NAND2, AND2, and inverter cells used throughout the design.



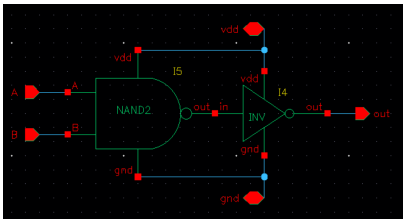
(a) XOR2



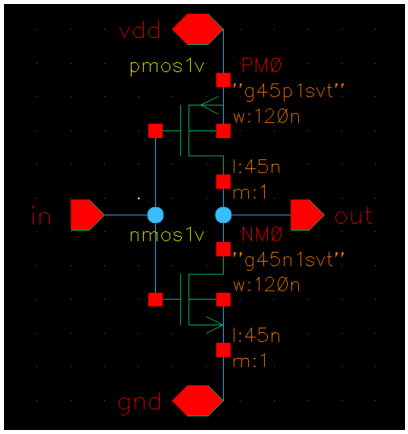
(b) NOR2



(c) NAND2



(d) AND2 (NAND2 + INV)



(e) Inverter

Figure 4: Primitive static-CMOS logic cells used in the baseline design. All transistors are minimum-sized devices (minimum width and length) in the 45 nm, 1.2 V process.

Inverter. The inverter (e) is a standard complementary pair: one PMOS pull-up to V_{DD} and one NMOS pull-down to ground, both using the minimum device size allowed by the PDK. No skew optimization is performed; this cell serves as the baseline reference driver and as the restoration stage after NAND2 to form an AND gate.

NAND2 and AND2. The 2-input NAND gate (c) uses two series NMOS devices in the pull-down network and two parallel PMOS devices in the pull-up network, all minimum size. Its output is low only when both inputs are high. The AND function is implemented as a NAND2 followed by an inverter (d), providing a full-swing static-CMOS realization of $A \cdot B$ used for the HA Carry and within the FA carry network.

NOR2. The 2-input NOR gate (b) is the dual of NAND2: two parallel NMOS devices in the pull-down and two series PMOS devices in the pull-up, again with minimum sizes. It outputs logic high only when both inputs are low and is used in the FA carry-combine network.

XOR2. The XOR2 cell (a) implements $A \oplus B$ using series/parallel transistor networks for the pull-up and pull-down, so that the output is high exactly when A and B differ. This gate is the critical building block for the Sum outputs of both the FA and HA and therefore appears frequently along timing-critical paths in the array.

2.4 Summary of Baseline Structure

In summary, the baseline design is a straightforward Braun array built from gate-level HAs and FAs, which in turn are constructed from minimum-sized static-CMOS logic cells. The architecture is regular and easy to reason about, but the gate-assembled adders lead to relatively deep logic chains, especially on the carry paths. This baseline provides a clean, fully functional reference against which we later compare the optimized full-adder topology and critical-path sizing results.

3 Baseline Functional Verification

To verify the correctness of both the baseline and optimized 4×4 multiplier implementations, we built a fully exhaustive, self-checking testbench around the transistor-level schematic. The testbench (1) applies all $2^8 = 256$ possible input combinations to the multiplier, (2) samples the eight output bits only after they have settled, and (3) automatically checks that every sampled output word matches the ideal product $Z = X \cdot Y$.

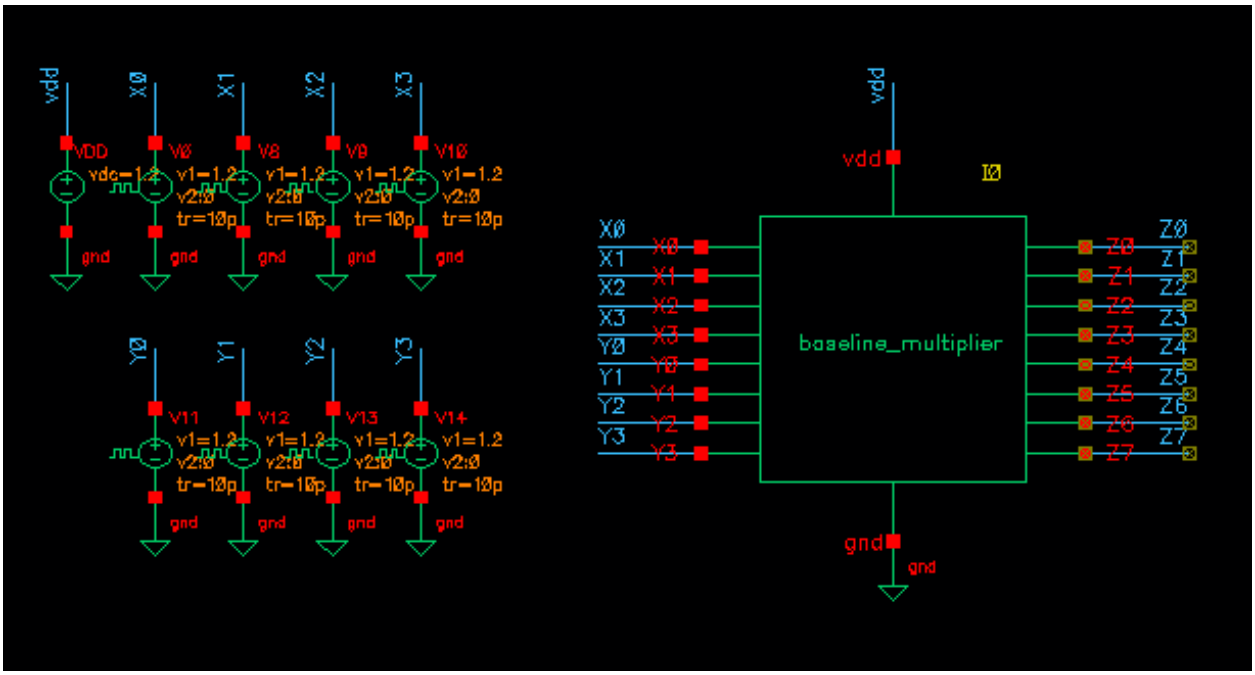


Figure 5

3.1 Exhaustive Transient Stimulus

The testbench consists of the multiplier under test driven by eight independent `vpulse` sources, one per primary input bit (X_0–X_3 and Y_0–Y_3). Each source is configured with a different period so that, taken together, the inputs realize a hardware binary counter:

$$\begin{aligned} X_0&: \text{ period } T = 10\,\text{ns}, & Y_0&: \text{ period } 16T \\ X_1&: \text{ period } 2T, & Y_1&: \text{ period } 32T \\ X_2&: \text{ period } 4T, & Y_2&: \text{ period } 64T \\ X_3&: \text{ period } 8T, & Y_3&: \text{ period } 128T \end{aligned}$$

With a 50% duty cycle on each bit, this configuration causes the eight inputs to count through all binary patterns from 0 to 255. Over a total simulation time of $1.28\,\mu\text{s} = 128T$, the testbench therefore applies every possible unsigned input pair (X, Y) with $X, Y \in [0, 15]$. The corresponding Cadence waveform capture is shown in Fig. 6, where the lower traces are the input bits and the upper traces are the multiplier outputs Z_0–Z_7.

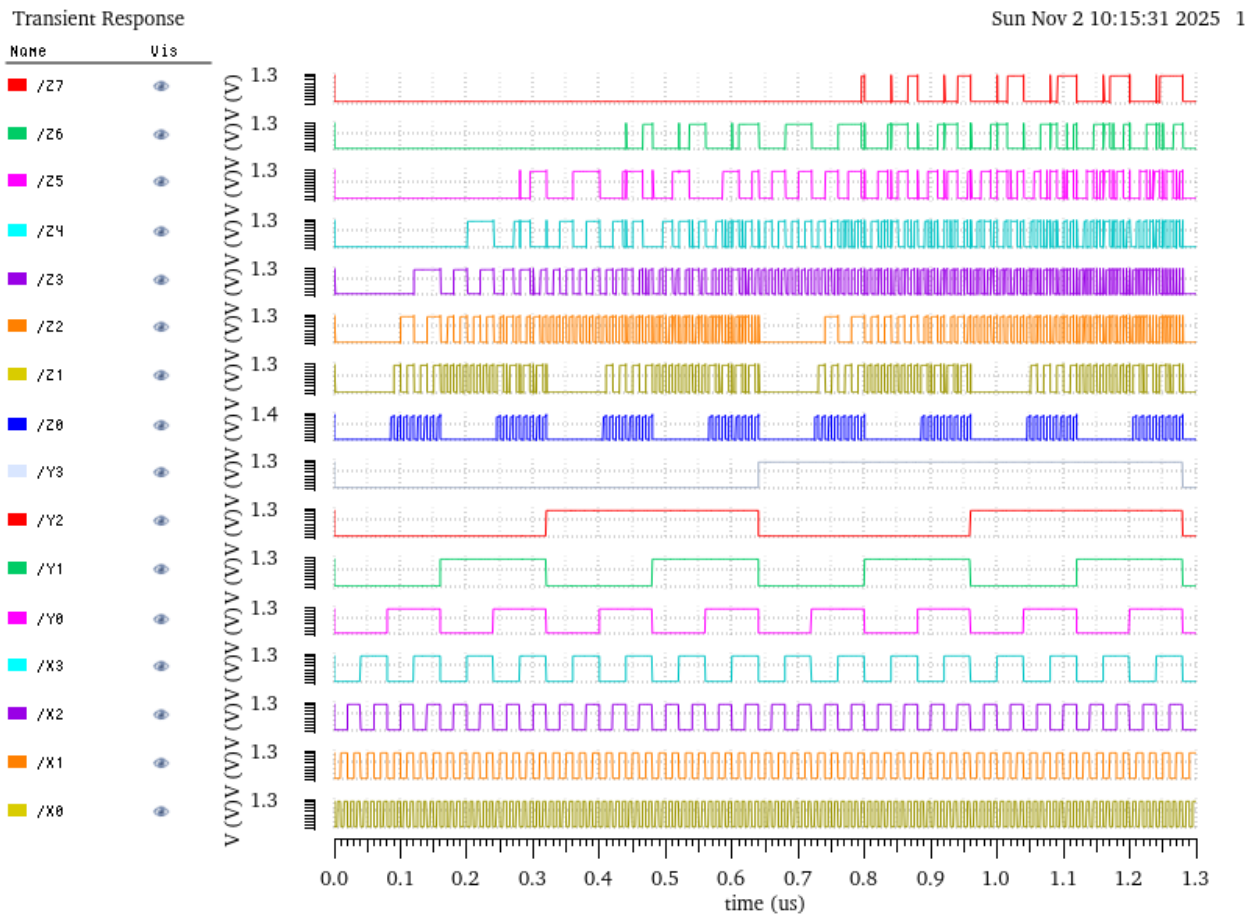


Figure 6: Transient response of the 4×4 multiplier under exhaustive binary-count stimulus. The eight input bits form a hardware counter, and the eight output bits respond to each new input vector.

Because this verification runs on the full transistor-level schematic, it naturally captures the effects of propagation delay, glitching, and finite output slew; no behavioral or idealized models are used.

3.2 Sampling and Logic-Level Conversion

The simulator output is exported to CSV and processed in Python. Rather than sampling at arbitrary times, we choose one time point in the middle of each interval where the inputs are guaranteed to be static. The shortest time between two transitions on any input is $T/2$ (the high or low half of X_0's period), so we define the kth sampling instant as

$$t_k = \frac{kT}{2} + \frac{T}{4}, \qquad k = 0, 1, \ldots, 255.$$

At t_k the counter has been stable for at least $T/4$ and the multiplier output has had time to settle to its final value.

For each t_k, the script locates the nearest time point in the exported CSV and extracts the voltages for all input and output nodes. These analog voltages are then converted to logic levels using conservative thresholds:

$$V > 1.0\,\mathrm{V} \Rightarrow \text{logic 1}, \qquad V < 0.2\,\mathrm{V} \Rightarrow \text{logic 0}.$$

Values that fall between 0.2 V and 1.0 V are marked as "unsettled" (stored as NaN) instead of being forced to a digital 0 or 1. This makes the check robust to slow transitions or ringing: if any output bit has not reached a valid logic level by the sampling time, the issue is surfaced immediately in the processed data rather than silently misclassified.

The result of this first Python stage is a new CSV file with exactly 256 rows (one per input vector) and clean digital values for all bits X_3–X_0, Y_3–Y_0, and Z_7–Z_0.

3.3 Reconstruction of Integer Operands and Products

A second Python script interprets the bit fields as integers and checks the arithmetic. A practical concern when exporting from Cadence is that the bit order in the CSV may not match the expected LSB-to-MSB ordering. To guard against this, the script first analyzes the number of transitions on each bit; the bit with the highest toggle count must be the LSB (it has the shortest period). If the fastest-toggling column is labeled X_3 instead of X_0, for example, the script concludes that the bits are reversed and flips the order in software before proceeding. The same logic is applied to the Y inputs. This automatic detection makes the verification insensitive to column-ordering mistakes in the export step.

With the bit ordering corrected, the integer values are reconstructed as

$$\begin{aligned} A &= X_0 + 2X_1 + 4X_2 + 8X_3, \\ B &= Y_0 + 2Y_1 + 4Y_2 + 8Y_3, \\ Z &= Z_0 + 2Z_1 + 4Z_2 + 8Z_3 + 16Z_4 + 32Z_5 + 64Z_6 + 128Z_7. \end{aligned}$$

These become the columns `A_val`, `B_val`, and `Z_val`. The correct reference result is simply

$$\texttt{Expected} = \texttt{A\_val} \times \texttt{B\_val}.$$

For each of the 256 vectors the script sets a Boolean flag `Pass` if `Z_val == Expected`. A short summary printed at the end of the run reports how many of the 256 cases passed and, if any failed, lists the mismatched vectors with their input pair, measured product, and expected product. For both the baseline and optimized multipliers, the script reports 256/256 passing cases.

3.4 Verification Table Visualization

To make the results easy to interpret at a glance, a final Python script converts the verified dataset into the 16×16 table shown in Fig. 7. The horizontal axis corresponds to the integer value of X, the vertical axis corresponds to Y, and each cell encodes the product for one (X, Y) pair:

- The upper-left number in each cell is the measured product Z from the simulation.

- The lower-right number is the ideal product $X \cdot Y$.
- The cell background is colored green if these two numbers agree and red otherwise.

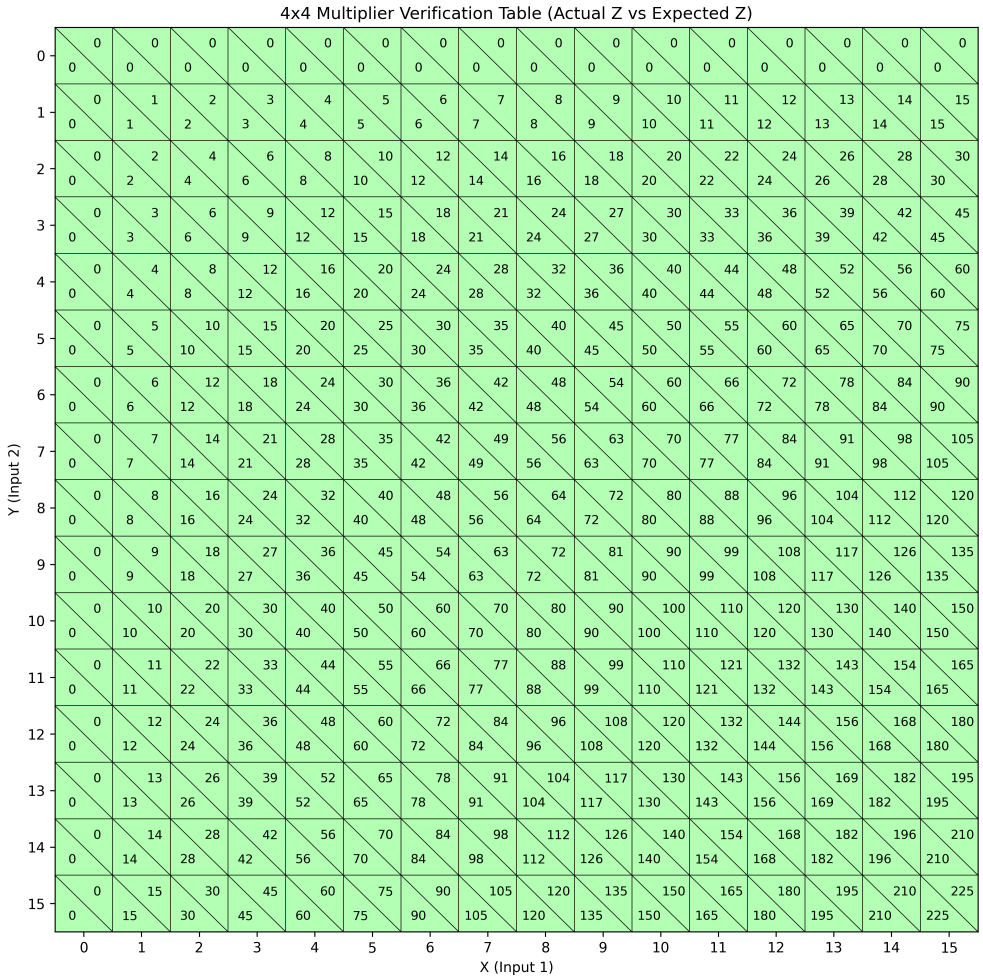


Figure 7: Visualization of functional verification for the 4×4 multiplier. For each input pair (X, Y), the upper-right number is the simulated product and the lower-left number is the ideal product $X \cdot Y$. Matching cells are shaded green; any mismatch would appear as a red cell.

Because the multiplier is correct, every cell in Fig. 7 is green and the two numbers in each cell are identical. Any systematic wiring error (for example, swapping two output bits or mis-interpreting signedness) would create a visible pattern of red cells, making this visualization a powerful sanity check in addition to the numerical pass/fail summary. The same scripts are reused for both the baseline and optimized designs, so any changes to the circuit can be re-verified quickly and objectively using identical criteria.

4 Baseline Delay Measurement

4.1 Worst–Case Delay Characterization

To determine the worst–case delay of the 4×4 array multiplier, we examine the structure of the adder array and identify the longest carry–dependent computation path. Each output bit Z_k is formed by a column of full adders (FAs) and half adders (HAs) whose inputs depend on the partial products $X_i Y_j$ and on the carry values generated by preceding adders. A delay "bottleneck" occurs whenever an adder must wait for its carry–in to settle before producing either its sum or carry–out. Thus, the critical path corresponds to the path along which the largest number of carry–dependent operations occur sequentially.

4.1.1 Critical Path Identification

Among all output bits, Z_7 is the furthest from the first carry–producing element and therefore admits the longest possible chain of dependent adders. The first carry is introduced at the Z_1 half adder, and from this point, the carry must propagate diagonally across the array through a sequence of full adders before eventually reaching the most significant output Z_7. Although multiple geometric paths exist from Z_1 to Z_7, each path contains the same total number of carry–in to carry–out transitions and the same number of carry–in to sum transitions. Because the array multiplier is topologically symmetric, all such paths have identical logical depth.

For a 4×4 array, the critical path contains:

- six FA/HAs contributing a Cin $\rightarrow$ Cout delay (horizontal movement), and
- two FA/HAs contributing a Cin $\rightarrow$ Sum delay (vertical movement).

One representative critical path is:

$$\text{AND} \rightarrow \text{HA}_{Z_1} \rightarrow \text{FA} \rightarrow \text{FA} \rightarrow \text{FA} \rightarrow \text{FA} \rightarrow \text{FA} \rightarrow \text{FA} \rightarrow \text{FA}_{Z_7},$$

where the ordering reflects successive carry propagation through the array.

4.1.2 Adder Operating States and Worst–Case Delay Conditions

An adder (FA or HA) can operate in one of three logical states depending on the values of its two data inputs (A, B):

1. **Terminate (T):** $(A, B) = (0, 0)$. In this state, the sum equals Cin, but the carry–out is always 0. This state prematurely stops the carry chain and therefore does *not* contribute to the worst–case delay.
2. **Propagate (P):** $(A, B) \in \{(0, 1), (1, 0)\}$. Both sum and carry–out depend on Cin, allowing the incoming carry to ripple through the adder. This is the state required to maximize delay.

3. **Generate (G):** $(A, B) = (1, 1)$. The carry–out is forced to logic high regardless of Cin, immediately terminating the dependency on previous stages. This state shortens the delay and therefore must be avoided on the critical path.

To excite the worst–case delay, *every adder along the identified critical path must be placed in the propagate state*. Once all adders are in propagate mode, toggling the carry–in of the first HA will cause a carry transition to ripple uninterrupted across the entire chain, from Z_1 through to Z_7.

4.1.3 Worst–Case Input Vector Selection

Let X_0 be the least significant bit of X and Y_0 the least significant bit of Y. To create a clean rising transition that initiates the carry propagation, X_0 is toggled from 0 to 1 while ensuring that the multiplier inputs enforce the propagate state at all adders along the critical path. One valid worst–case input assignment is:

Input Bit	Value	Effect
X_0	$0 \rightarrow 1$	Introduces a rising carry stimulus at HA_{Z_1}
X_1	0	Ensures propagate state at next HA
X_2	0	Propagate state for deeper FAs
X_3	1	Propagate state in upper FA chain
Y_0	1	Ensures HA input is capable of seeing Cin transition
Y_1	1	Same as above
Y_2	1	Maintains propagate condition in intermediate FAs
Y_3	1	Maintains propagate condition at upper FAs

With these inputs, all adders along the $Z_1 \rightarrow Z_7$ critical path reside in the propagate state, ensuring that a single transition at Cin traverses the full logical depth of the array. Measuring the time difference between the transition at X_0 and the corresponding transition observed at Z_7 yields the worst–case propagation delay of the baseline multiplier.

4.2 Worst-Case Delay Test Schematic

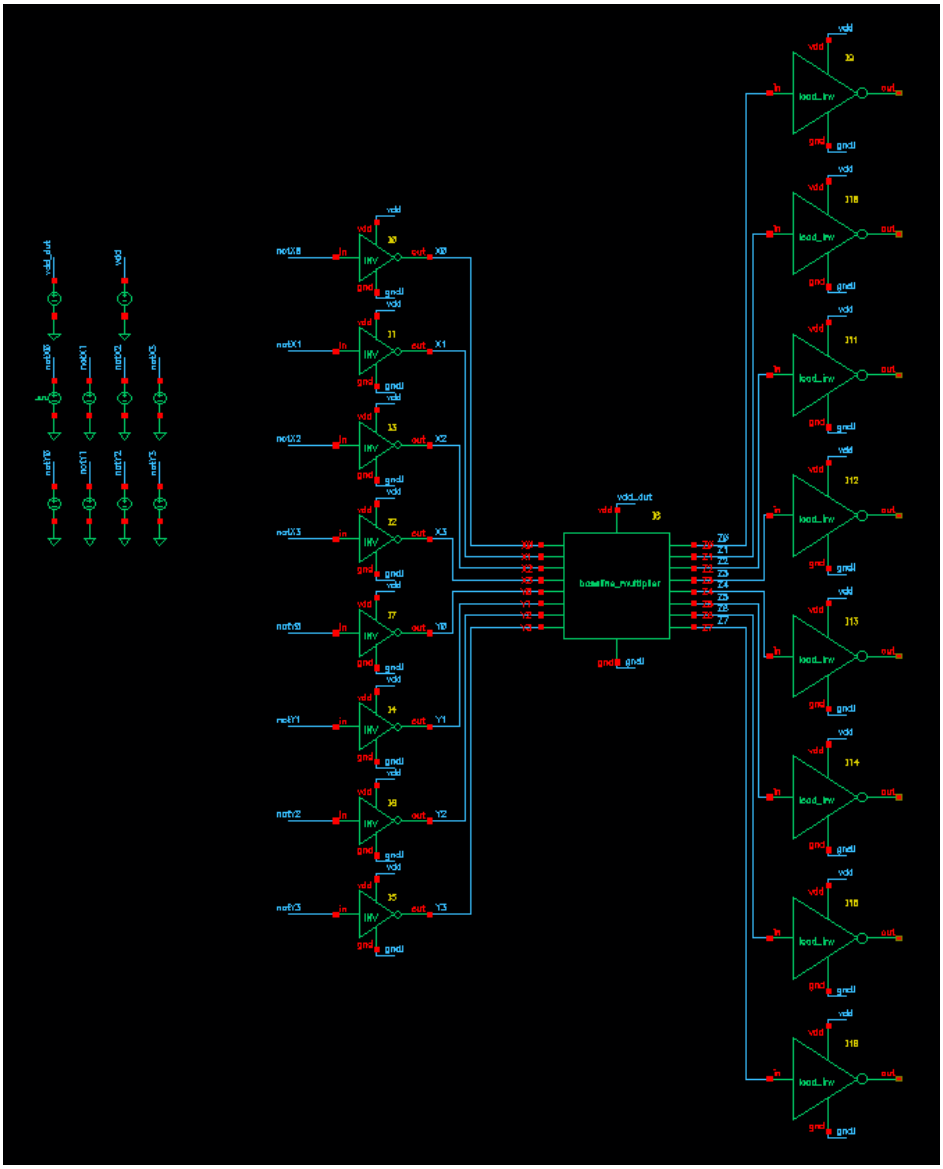


Figure 8

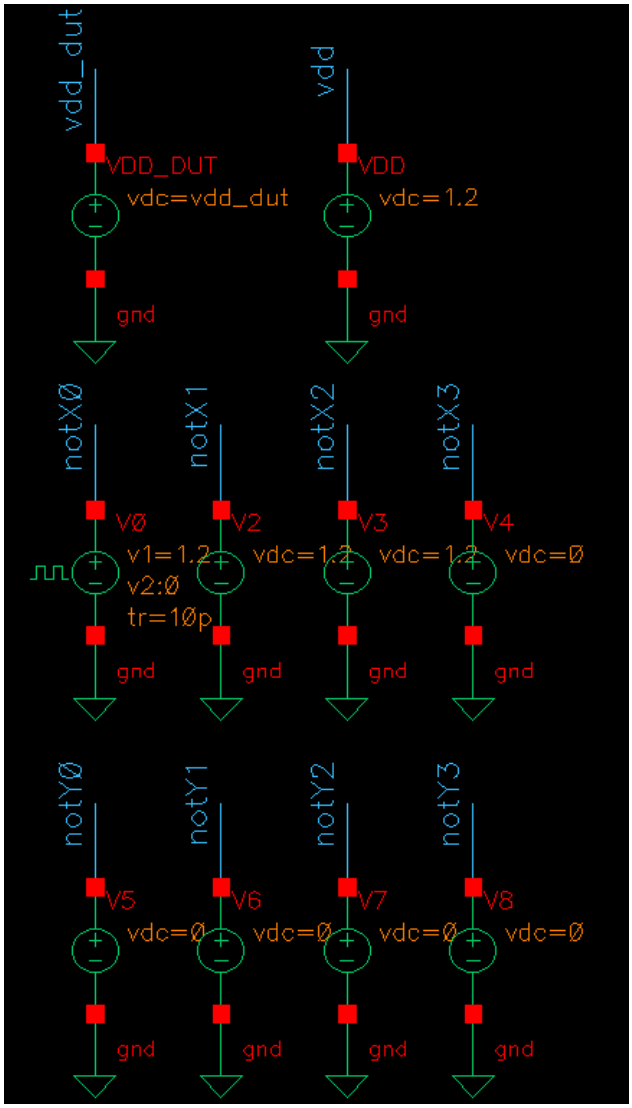

vdd_dut
vdd
VDD_DUT
vdc=vdd_dut
VDD
vdc=1.2
gnd
gnd
notX0
notX1
notX2
notX3
V0
V2
V3
V4
v1=1.2
v2:0
tr=10p
vdc=1.2
vdc=1.2
vdc=0
gnd
gnd
gnd
gnd
notY0
notY1
notY2
notY3
V5
V6
V7
V8
vdc=0
vdc=0
vdc=0
vdc=0
gnd
gnd
gnd
gnd

Figure 9

4.3 Worst-Case Delay Analysis

4.3.1 Worst Case Propagation for Switching Input 0 → 1

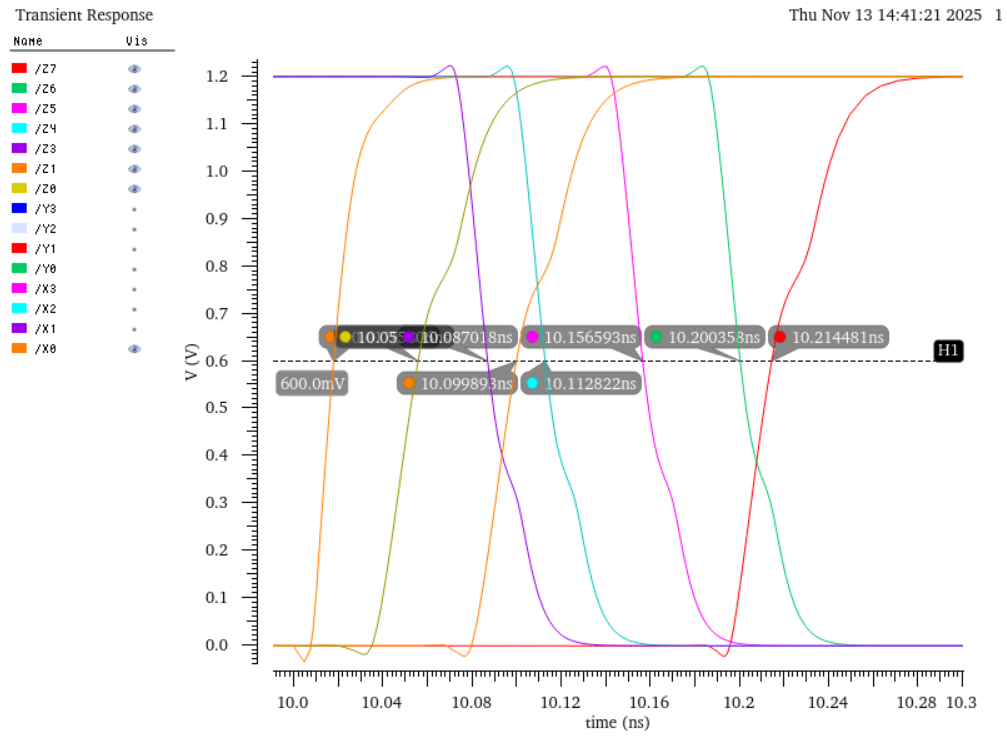


Figure 10

To extract the worst–case delay, the multiplier was excited with the input pattern described in Section 4.1.3, ensuring that all adders along the $Z_1 \rightarrow Z_7$ diagonal operate in the propagate state. A rising transition was applied to X_0, which produces the first carry at the Z_1 half adder and initiates a carry ripple through all adders on the critical path.

The propagation delay t_{PLH} was measured using the 50% voltage points of the input and output waveforms. The X_0 input crosses 0.6 V at

$$t_{\mathrm{in},50\%} = 10.017844 \text{ ns},$$

and the most significant output bit Z_7 crosses 0.6 V at

$$t_{\mathrm{out},50\%} = 10.214481 \text{ ns}.$$

The low–to–high propagation delay is therefore

$$t_{\mathrm{PLH}} = t_{\mathrm{out},50\%} - t_{\mathrm{in},50\%} = 10.214481 \text{ ns} - 10.017844 \text{ ns} = 0.196637 \text{ ns}.$$

Thus, the worst–case delay of the baseline multiplier is

$$\boxed{t_{\mathrm{PLH}} = 196.637 \text{ ps}}$$

4.3.2 Worst Case Propagation for Switching Input 1 → 0

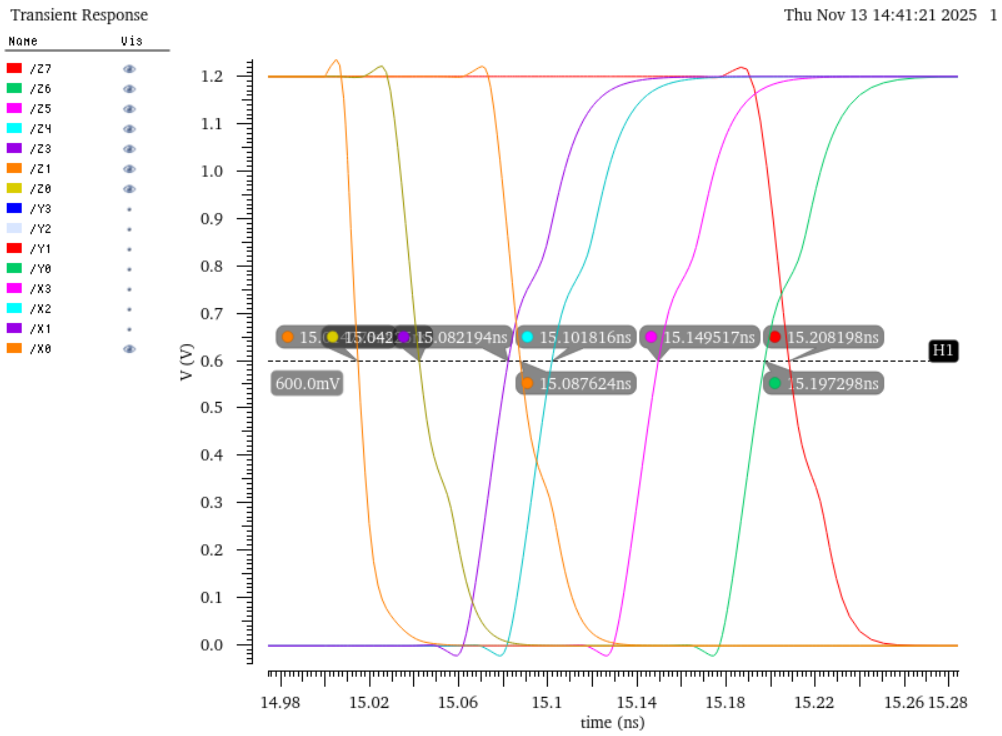


Figure 11

The high–to–low worst–case delay was obtained using the same propagate–state input vector used for the low–to–high measurement. In this case, X_0 transitions from logic high to logic low, and the resulting carry discharge travels along the $Z_1 \rightarrow Z_7$ critical path.

The 50% crossing of the input X_0 waveform occurs at

$$t_{\text{in},50\%} = 15.014775 \text{ ns},$$

while the 50% crossing of the output Z_7 waveform occurs at

$$t_{\text{out},50\%} = 15.208198 \text{ ns}.$$

Thus, the high–to–low propagation delay is

$$t_{\text{PHL}} = t_{\text{out},50\%} - t_{\text{in},50\%} = 15.208198 \text{ ns} - 15.014775 \text{ ns} = 0.193423 \text{ ns}.$$

Therefore, the worst–case t_{PHL} of the baseline multiplier is

$$\boxed{t_{\text{PHL}} = 193.423 \text{ ps}}$$

5 Baseline Active Energy Measurement

5.1 Active Energy Test Schematic

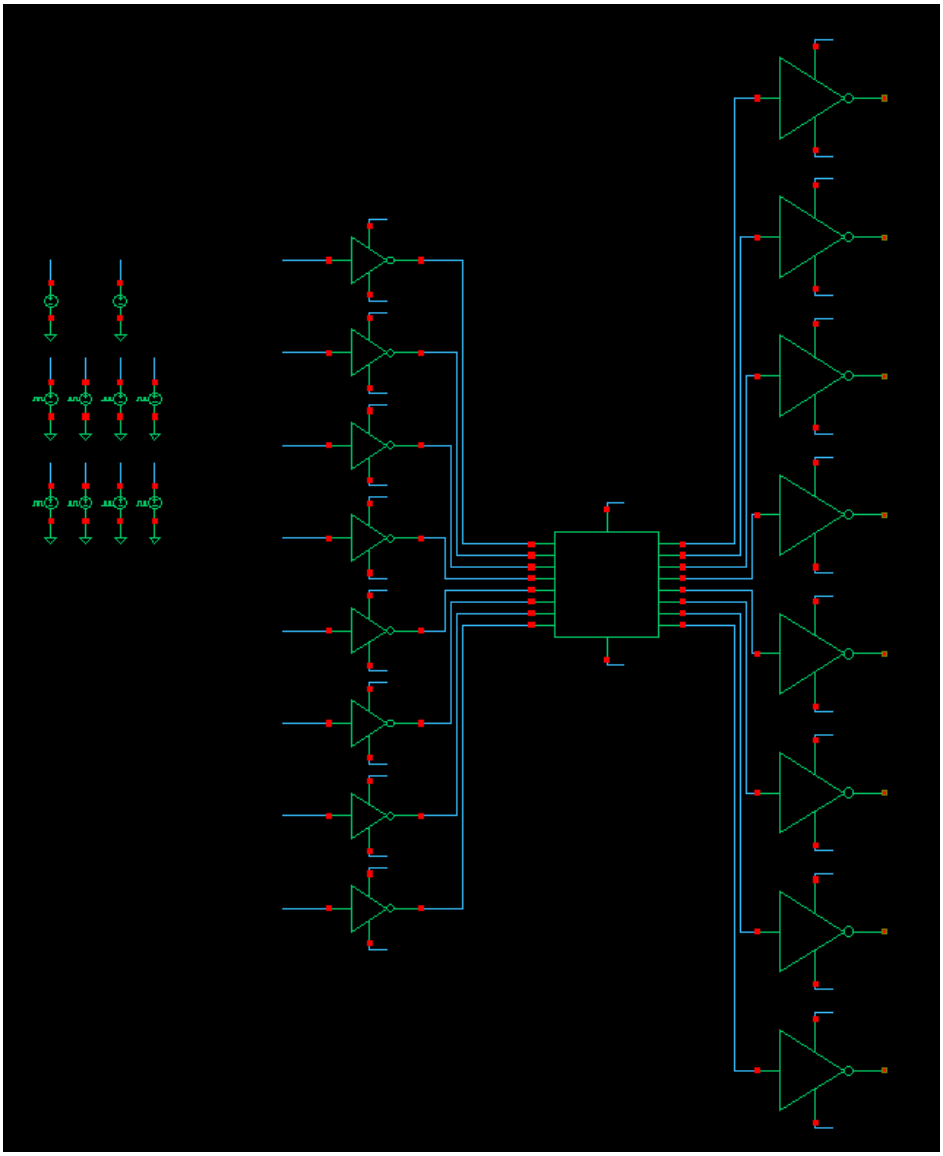


Figure 12

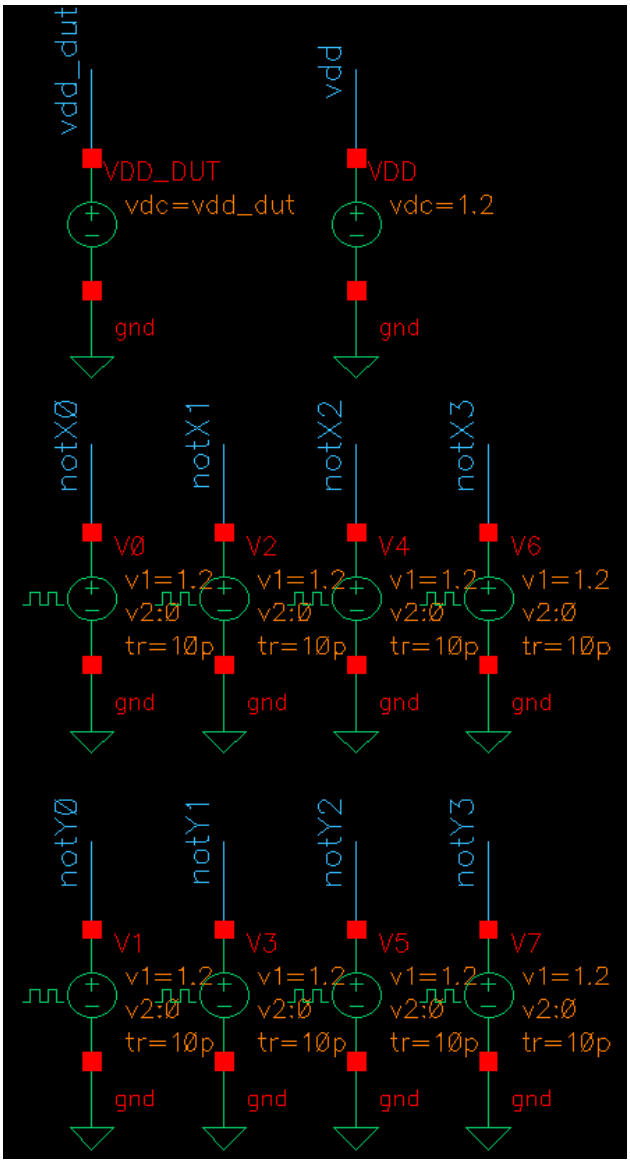


Figure 13: Maximum active energy inputs

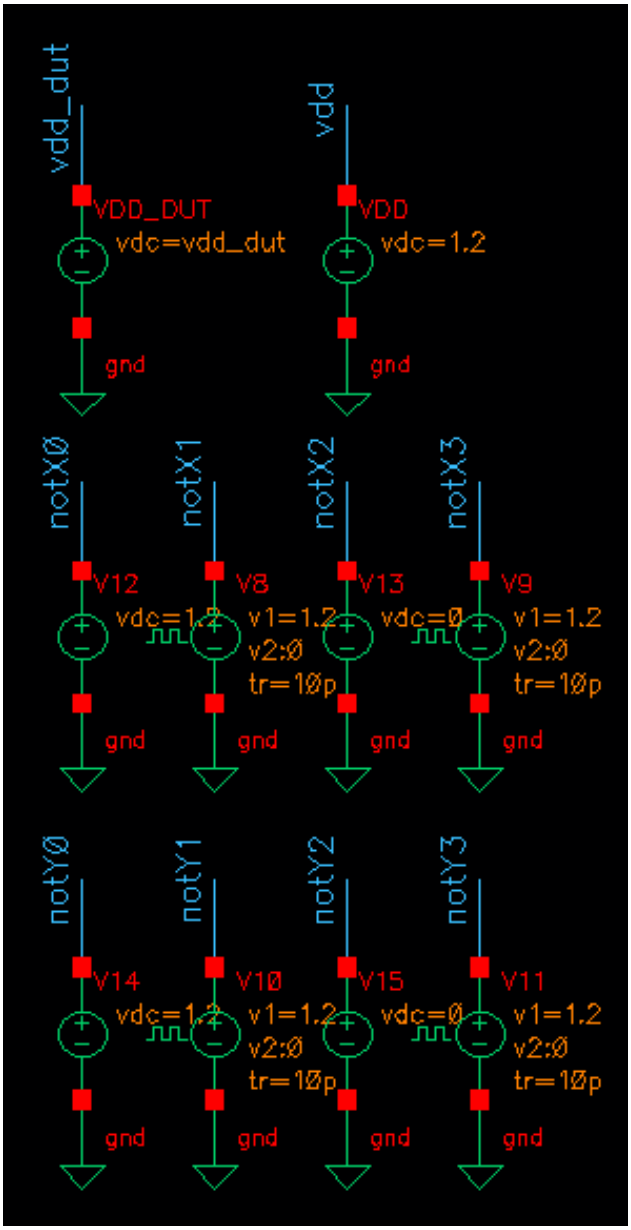


Figure 14: Average active energy inputs

5.2 Active Energy Analysis

To characterize the dynamic power consumption of the baseline 4×4 array multiplier, active switching energy was measured under two representative transitions: (i) a maximum-switching case where every input bit toggles from $0 \to 1$, and (ii) an average-switching case where only half of the input bits toggle. In both tests, the multiplier's VDD rail was isolated on a dedicated supply pin, allowing the instantaneous supply current to be integrated directly using ADE's `integ()` operator:

$$E_{\text{active}} = \int_{t_0}^{t_1} i_{\text{VDD}}(t)\, V_{\text{DD}}\, dt.$$

5.2.1 Maximum Switching Case

In the maximum-switching condition, all bits of X and Y transition simultaneously from 0 to 1. This forces nearly all internal nodes to switch, producing the highest toggle activity factor and therefore the highest dynamic energy.

Figure 15 shows the transient switching activity across the multiplier during this input event. The ADE current integration result in Fig. 16 reports a total energy consumption of:

$$E_{\text{max}} = 122.2 \text{ fJ}.$$

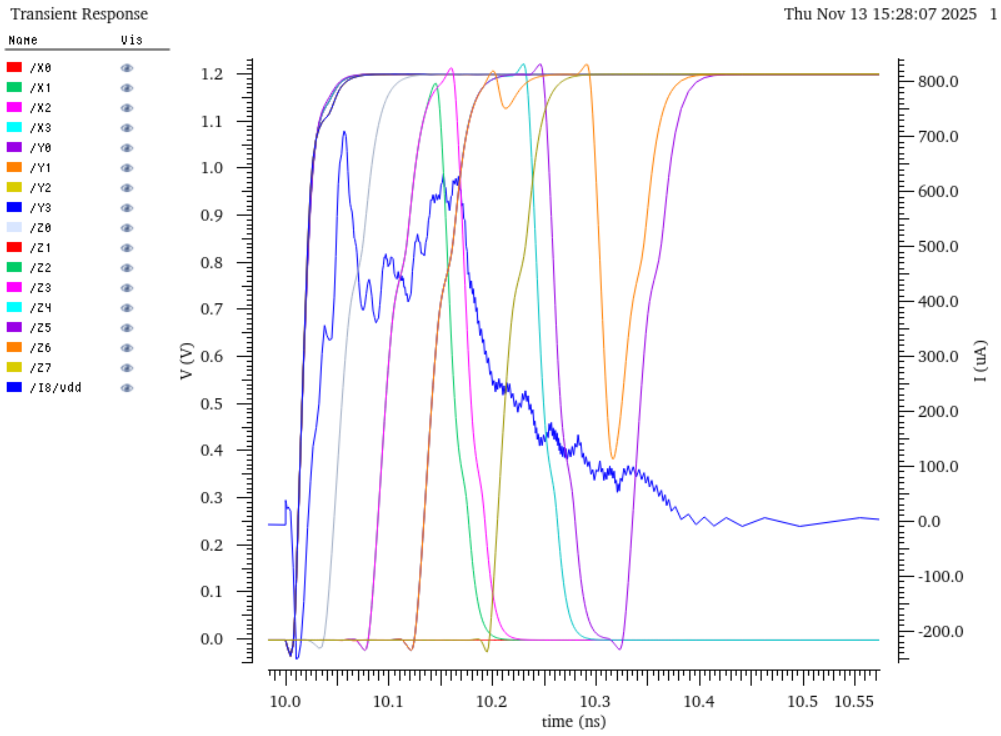


Figure 15: Transient output and internal switching behavior under the maximum-switching input case ($X, Y : 0 \to 1$).

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 10n 10.462837n " ")*1.2	122.2E-15

Figure 16: Integrated supply current for the maximum-switching case, yielding $E_{\max} = 122.2$ fJ.

5.2.2 Average Switching Case

The average-switching case represents a more typical energy profile, where roughly half of the bits in each operand toggle from $0 \rightarrow 1$. This reduces the number of partial-product transitions and internal adder activations.

The transient behavior for this case is shown in Fig. 17, and the corresponding ADE integration result in Fig. 18 yields:

$$E_{\text{avg}} = 89.83 \text{ fJ}.$$

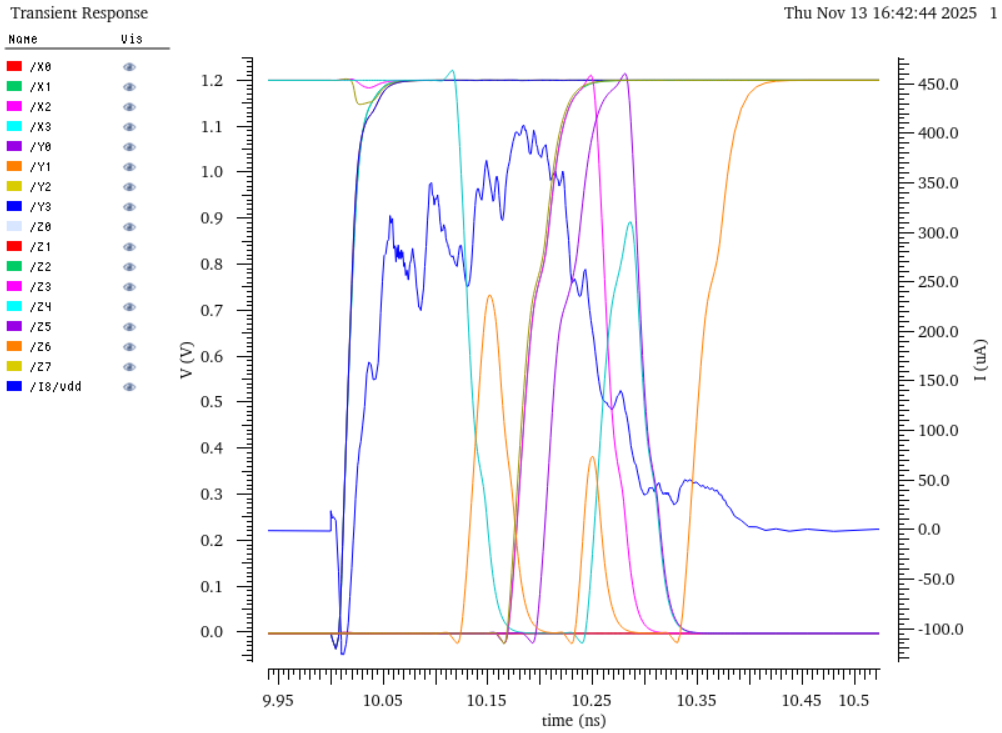


Figure 17: Transient output and internal switching behavior under the average-switching input case.

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 10n 10.4801577n " ")*1.2	89.83E-15

Figure 18: Integrated supply current for the average-switching case, yielding $E_{\text{avg}} = 89.83$ fJ.

6 Baseline Leakage Energy Measurement

6.1 Leakage Energy Test Schematic

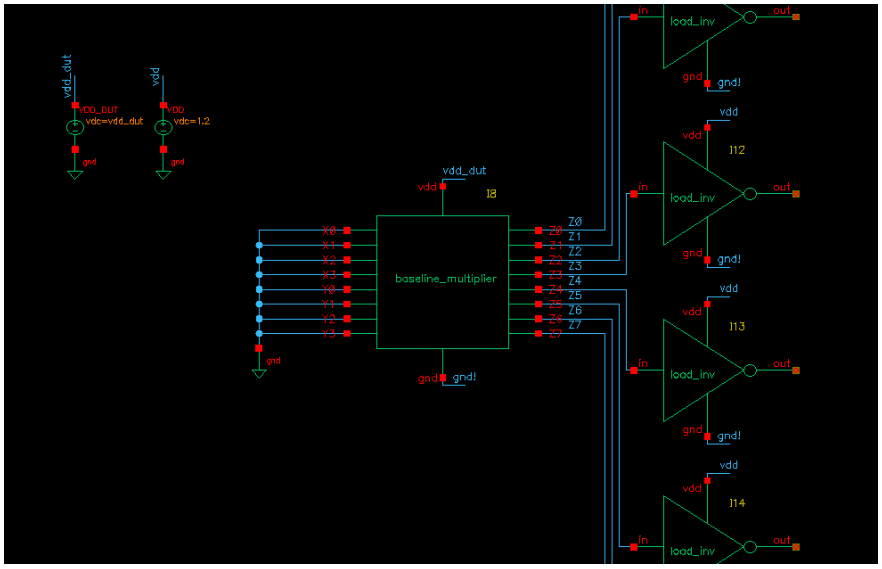


Figure 19: Maximum leakage energy testbench

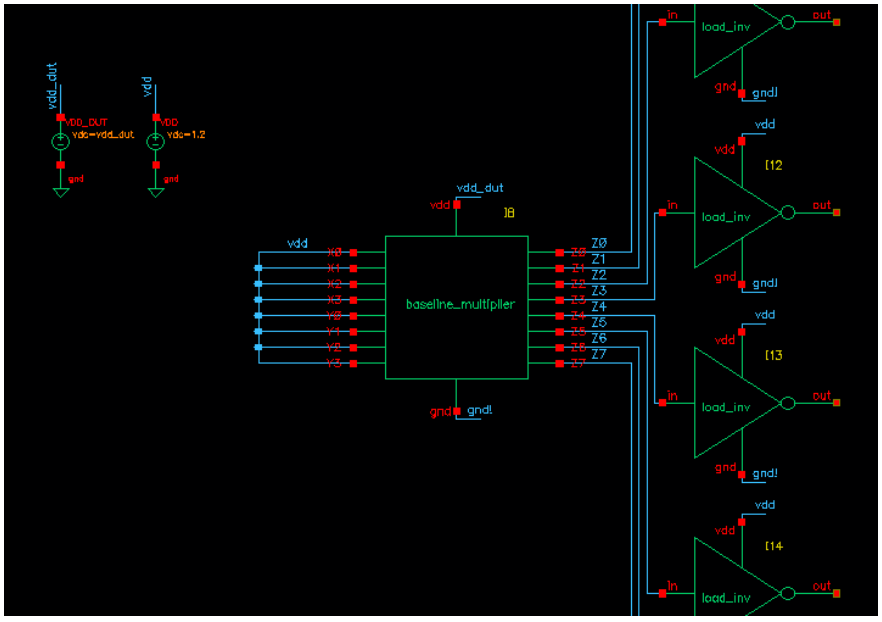


Figure 20: Minimum leakage energy testbench

6.2 Leakage Energy Analysis

Leakage energy in the 4×4 array multiplier is dominated by the static leakage of the full adders (FAs), which form the majority of the internal logic. Because each FA contributes similarly to the total static energy, the maximum and minimum leakage cases for the entire multiplier can be inferred directly by characterizing a single FA. Once the FA's highest– and lowest–leakage input states are identified, the multiplier-level input vectors follow naturally by applying the same bit patterns to all FA instances.

6.2.1 Leakage Energy Model

The leakage energy over one delay period T is defined as

$$U_{\text{leak}} = \int_0^T V_{\text{DD}}\, i(t)\, dt.$$

Since the leakage current varies only slightly during static input conditions, we approximate it as constant:

$$U_{\text{leak}} \approx V_{\text{DD}} \cdot i_{\text{leak}} \cdot T.$$

With $V_{\text{DD}} = 1.2\,\text{V}$ and a delay period of $T = 196.6\,\text{ps}$ (the worst-case propagation delay measured for the baseline multiplier), all leakage energies are normalized to this interval.

6.3 Full-Adder Leakage Characterization

To characterize FA leakage, we exhaustively sweep all static input combinations:

$$(\text{In1}, \text{In2}, C_{\text{in}}) \in \{0, 1\}^3.$$

For each input triplet:

- Inputs are tied to constant VDD or GND.
- Outputs are loaded with the standard inverter load ($8C_g$).
- A 4 ns transient simulation is run in Spectre.
- Leakage energy is extracted using:

$$U_{\text{leak}} = \text{integ}\big(\text{abs}(i(\text{“/VDD/PLUS”}))\big)\Big|_{t_0}^{t_0+T} \cdot 1.2.$$

A representative FA leakage waveform for the $(0, 0, 0)$ input case is shown in Figure 21.

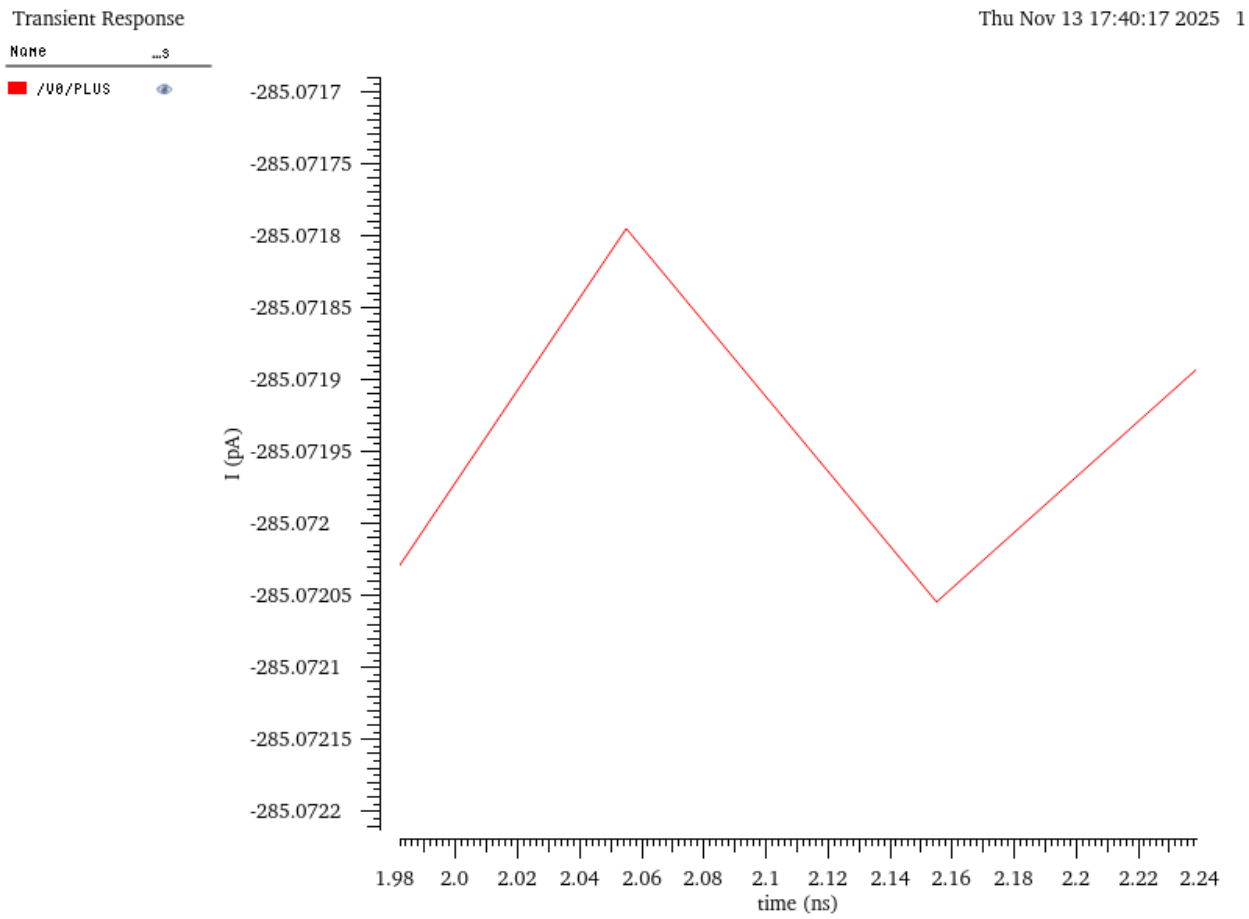


Figure 21: Example FA leakage current waveform for In1 = In2 = $C_{\text{in}} = 0$.

	Expression	Value
1	integ(i("/V0/PLUS" ?result "tran") 2n 2.196637n " ")*1.2...	67.27E-21

Figure 22: Example leakage energy for the $(0, 0, 0)$ input case. The remaining input cases were tested and extracted values are shown below in Table 1

In1	**In2**	**C_{in}**	**Leakage Energy (zJ)**
0	0	0	67.27
1	0	0	61.28
0	1	0	62.88
1	1	0	55.29
0	0	1	61.57
1	0	1	57.41
0	1	1	59.83
1	1	1	52.07

Table 1: Full-adder leakage energy for all static input combinations (integrated over one delay period).

From Table 1, the maximum leakage occurs for

$$\text{In1} = \text{In2} = C_{\text{in}} = 0,$$

and the minimum leakage for

$$\text{In1} = \text{In2} = C_{\text{in}} = 1.$$

The all-zero case creates several partially-conducting leakage paths through the pull-down devices, while the all-one case produces stronger OFF states, reducing leakage.

6.4 Multiplier Leakage Measurement

Using the FA-level results, we apply the corresponding input patterns to the entire multiplier. This ensures every FA experiences the same static state.

$$\textbf{Maximum leakage case:} \quad X_3X_2X_1X_0 = 0000, \quad Y_3Y_2Y_1Y_0 = 0000,$$
$$\textbf{Minimum leakage case:} \quad X_3X_2X_1X_0 = 1111, \quad Y_3Y_2Y_1Y_0 = 1111.$$

The same methodology is used: static inputs, a 4 ns transient run, and integration over the window $T = 196.6\,\text{ps}$ on the supply current.

The multiplier-level leakage energies are summarized in Table 2.

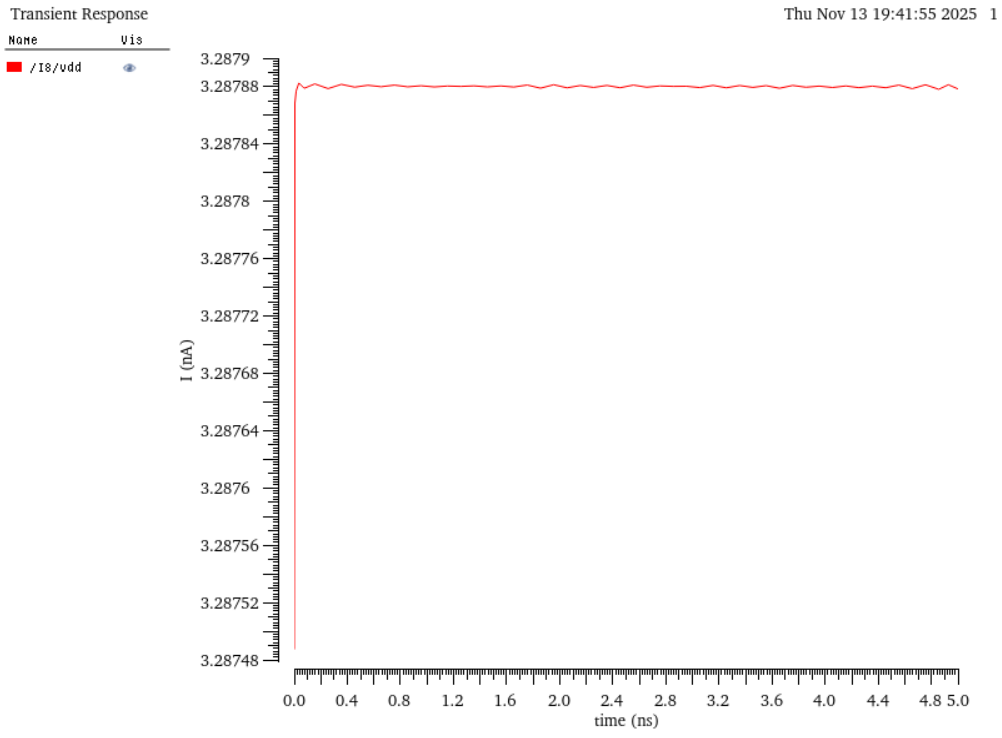


Figure 23: Minimum leakage signal

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 2n 2.196637n " ")*1.2	775.8E-21

Figure 24: Minimum leakage value

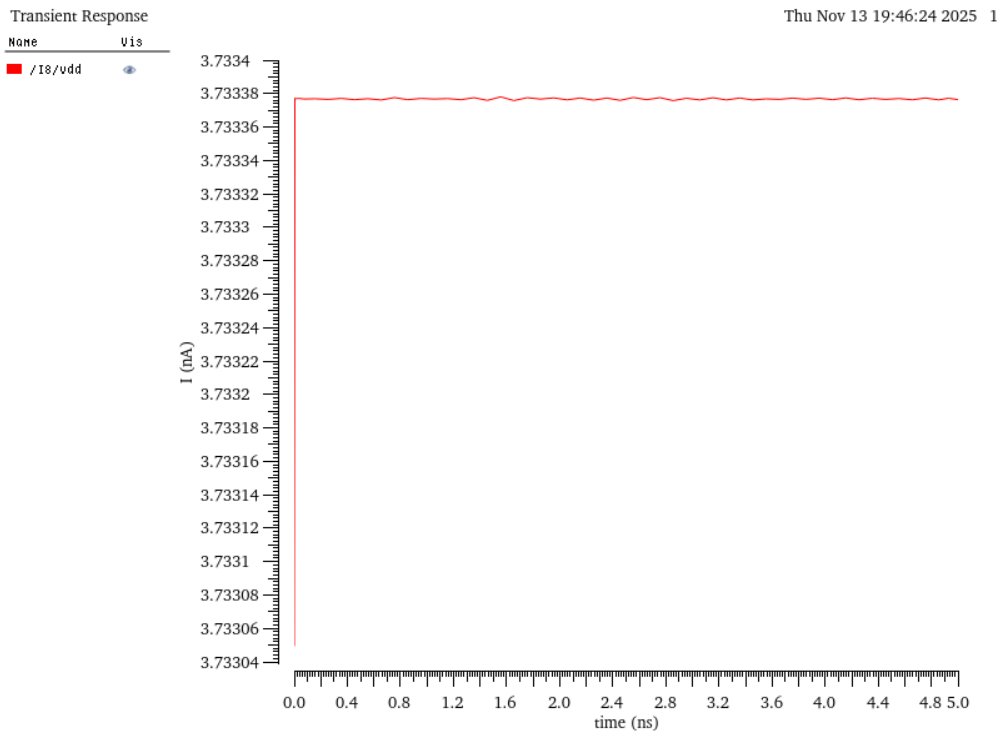


Figure 25: Maximum leakage signal

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 2n 2.196637n " ")*1.2	880.9E-21

Figure 26: Maximum leakage value

X Inputs	**Y Inputs**	**Leakage Energy (zJ)**
0000	0000	880.9
1111	1111	775.8

Table 2: Multiplier leakage energy for maximum and minimum leakage input vectors.

These results scale consistently with the FA characterization: the multiplier consumes the highest leakage energy when all FAs see the $(0, 0, 0)$ state, and the lowest leakage when all see $(1, 1, 1)$. The nearly linear scaling confirms that leakage is dominated by per-FA device leakage rather than interconnect or loading effects.

7 Baseline Design Summary Table

Table 3 summarizes the key performance metrics of the baseline 4×4 array multiplier, including worst-case delay, dynamic (active) energy, and static (leakage) energy characteristics.

Metric	**Value**
Low-to-high delay t_{PLH}	196.637 ps
High-to-low delay t_{PHL}	193.423 ps
Worst-case delay $t_{\text{wc}} = \max(t_{PLH}, t_{PHL})$	196.637 ps
Maximum switching active energy E_{max}	122.2 fJ
Average switching active energy E_{avg}	89.83 fJ
Maximum leakage energy ($X = 0000$, $Y = 0000$)	880.9 zJ
Minimum leakage energy ($X = 1111$, $Y = 1111$)	775.8 zJ

Table 3: Summary of baseline multiplier delay and energy metrics.

8 Summary of Optimization Process

8.1 Worst Case Delay

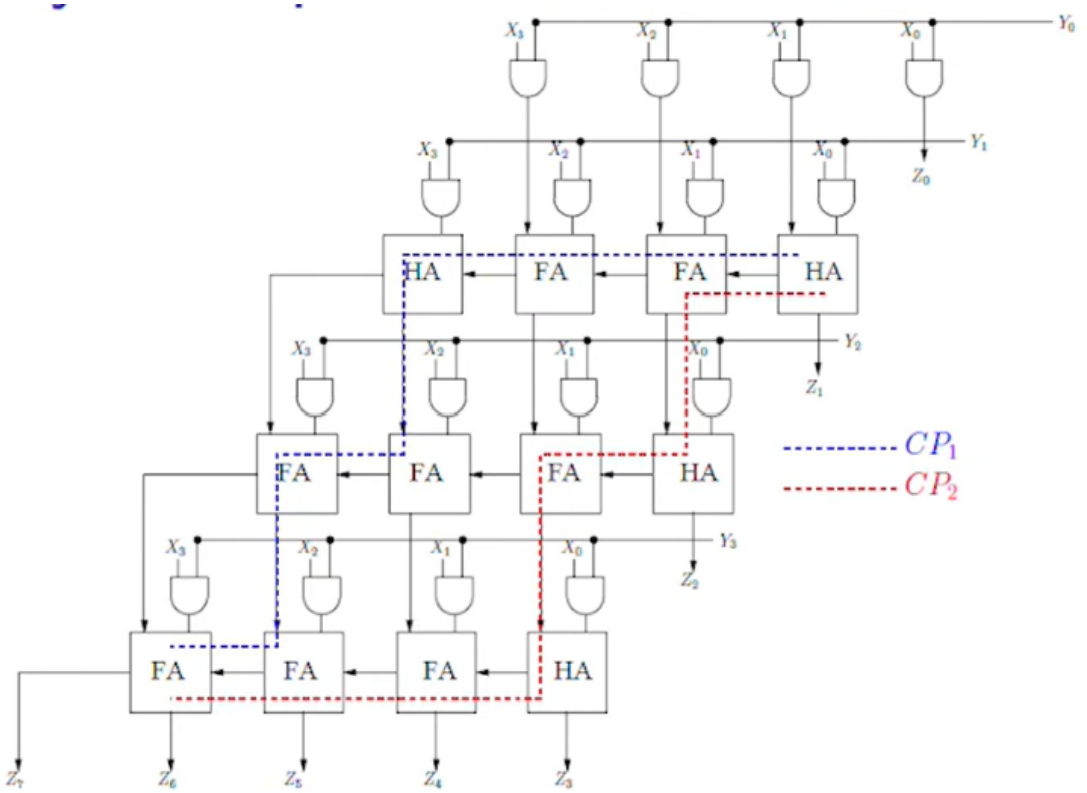


Figure 27: Critical-path staircases in a 4×4 Braun array multiplier. X_0–X_3 are the N bits (multiplicand) and Y_0–Y_3 are the M bits (multiplier). Figure adapted from materials by Prof. Janakiraman Viraraghavan, IIT Madras.

We analyze the 4×4 Braun (array) multiplier shown above. The multiplicand bits are X_0–X_3 (the N bits) and the multiplier bits are Y_0–Y_3 (the M bits). Each partial product $x_i y_j$ is generated by a static-CMOS AND gate, and these partial products are reduced by a regular grid of half adders (HAs) and full adders (FAs). Within a column, carries propagate downward; between columns, sums propagate leftward. The slow paths are the northeast-to-southwest staircases that begin at a right-edge partial product and alternate vertical carry hops with horizontal sum hops until they reach the lower-left outputs.

Delay primitives and convention We define three cell delays under the loading and drive conditions that match the tiled array:

t_{and} for the partial-product AND, t_{carry} for an HA/FA carry-out, t_{sum} for an HA/FA sum-out.

In static-CMOS adders the sum network is built from XOR/XNOR structures that have larger logical effort and parasitic capacitance than the carry (majority / AND-OR) network,

so typically

$$t_{\text{sum}} > t_{\text{carry}}.$$

Exciting the limiting staircase The worst transition is not the input vector that flips the most bits. It is the transition that (i) toggles a right-edge partial product and (ii) places each encountered adder in propagate mode ($a \oplus b = 1$ and $ab = 0$), forcing a newly injected carry to ripple through all stages along a staircase to the lower-left.

Counting hops on an M×N array For any such staircase in an M×N array, the stage counts are:

$$\text{carry hops} = (M-1) + (N-2),$$
$$\text{sum hops} = (M-1),$$

plus the initial t_{and} that launches the path. Therefore the generic array-multiplier timing is

$$t_{\text{array}}(M,N) \;=\; \big[(M-1)+(N-2)\big]\, t_{\text{carry}} \;+\; (M-1)\, t_{\text{sum}} \;+\; t_{\text{and}}.$$

Specialization to the 4×4 case With $M = N = 4$ we obtain five carry hops and three sum hops:

$$t_{\text{path}} \;=\; 5\, t_{\text{carry}} + 3\, t_{\text{sum}} + t_{\text{and}}.$$

In the 4×4 topology the bottom-left FA drives two neighboring outputs: its carry is Z_7 and its sum is Z_6. The staircase counted above terminates at the sum of this FA, so

$$\boxed{t_{Z_6} = 5\, t_{\text{carry}} + 3\, t_{\text{sum}} + t_{\text{and}}}.$$

Because $t_{\text{sum}} > t_{\text{carry}}$ for static-CMOS FAs, Z_6 is slower than the otherwise identical staircase that would end at the carry output Z_7. Hence the worst-case delay of the 4×4 array ends at Z_6, not Z_7.

Implications Multiple staircases (such as CP_1 and CP_2 in the figure) have the same hop counts and are nearly iso-delay, so cell sizing must balance stage effort across the grid. Nevertheless, the decisive last stage is the sum network of the bottom-left FA that produces Z_6, and this stage should be weighted accordingly when sizing to minimize the overall worst-case delay.

8.2 Schematics in Static CMOS

8.2.1 Full Adder (FA)

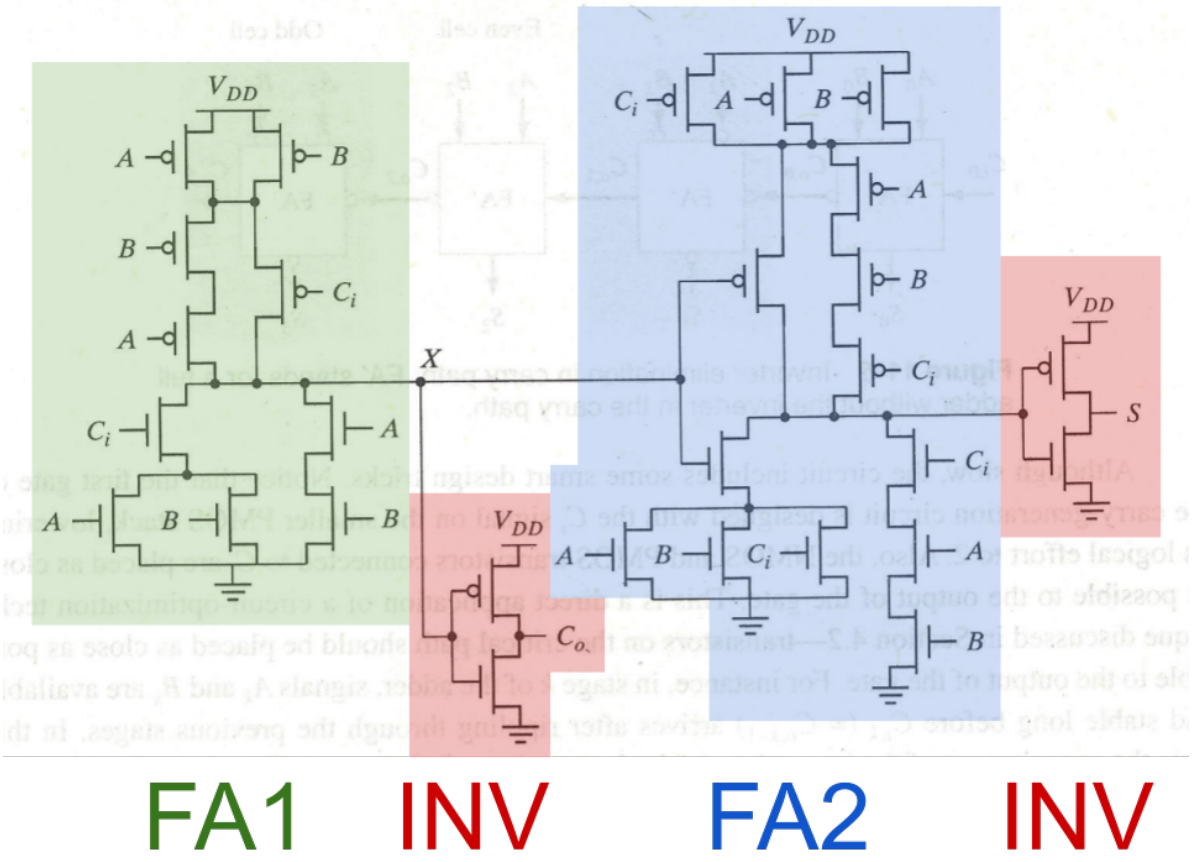


Figure 28: Complementary static-CMOS full adder. Figure adapted from Rabaey.

The full adder computes the carry-out C_o and sum S from inputs A, B, and C_i. We employ a complementary static-CMOS realization that shares intermediate logic between the carry and sum paths. The logic is organized as

$$C_o = AB + BC_i + AC_i \quad \text{(3-input majority)}, \qquad S = ABC_i + C_o\,(A + B + C_i).$$

The schematic implements these expressions with series–parallel p- and n-networks so that each output is produced by static pull-up and pull-down paths, guaranteeing full-swing, monotonic transitions and large noise margins.

Signal flow and stage counts The circuit is partitioned around an internal node X that realizes the reorganized carry logic.

- **Path to C_o (two static stages).** A complex CMOS network first forms X, which captures the majority function. An output inverter then buffers X to produce C_o.

- **Path to S (three static stages).** The same X is reused by a second static network that combines X with A, B, and C_i to generate a pre-sum node. A final inverter buffers this node to produce S.

Thus, C_o emerges after two static stages and S after three. While not "single-stage" at the pins, these are compact paths with low internal fanout and limited parasitic loading.

Why static CMOS over a bag of gates A gate-assembled FA (e.g., library XOR/AND/OR composition) typically yields a deeper chain: C_o commonly requires two XOR-class stages, and S often traverses a sequence such as XOR $\rightarrow$ NAND $\rightarrow$ NOT $\rightarrow$ NOR $\rightarrow$ NOT. Each additional logic boundary adds logical effort and parasitic capacitance, inflating end-to-end delay and degrading slews. The static-CMOS realization reduces the number of boundaries, shares X rather than duplicating heavy networks, and eliminates large internal fanouts between discrete gates. Because both outputs are produced by complementary static networks and then buffered, signals remain rail-to-rail and monotonic across many cascaded cells.

Fit for the array multiplier In the 4×4 Braun array, the slow paths are the northeast-to-southwest staircases of adders, and the measured worst case terminates at the sum output of the bottom-left FA (Z_6). Replacing multi-gate chains with the compact static implementation shortens the effective stage depth on these staircases and reduces internal parasitics, lowering the overall delay.
At the same time, static-CMOS robustness (no ratioing, no dynamic storage) ensures predictable behavior when many FA cells are tiled: outputs do not droop through the grid, hazards are suppressed by the static topologies and output inverters, and uniform polarity avoids the need for multiple FA variants or extra inverters during integration.
In summary, the Rabaey complementary static-CMOS FA provides a favorable speed–robustness tradeoff for the multiplier: fewer and better stages than a bag-of-gates implementation, shared intermediate logic that limits internal loading, and full-swing stability that holds when cascading many cells.

8.2.2 Half Adder (HA)

The half adder takes one-bit inputs A and B and produces outputs *Sum* and *Carry*. From the truth table,

$$\text{Sum} = A \oplus B, \qquad \text{Carry} = A \cdot B = \overline{\text{NAND}(A, B)}.$$

Implementation in complementary static CMOS is therefore direct:

- **Sum path.** Realize $A \oplus B$ with a static-CMOS XOR gate. This is a single logic stage at the Sum pin with full-swing, monotonic behavior.

- **Carry path.** Realize $A \cdot B$ as a static-CMOS NAND followed by a static inverter: $\text{Carry} = \overline{\text{NAND}(A, B)}$. This is two logic stages at the Carry pin (NAND then inverter), also full swing and monotonic.

Unlike the full adder, there is no third input to exploit for algebraic sharing or logic reorganization; the minimal Boolean forms are already $A \oplus B$ and $A \cdot B$. Consequently, the

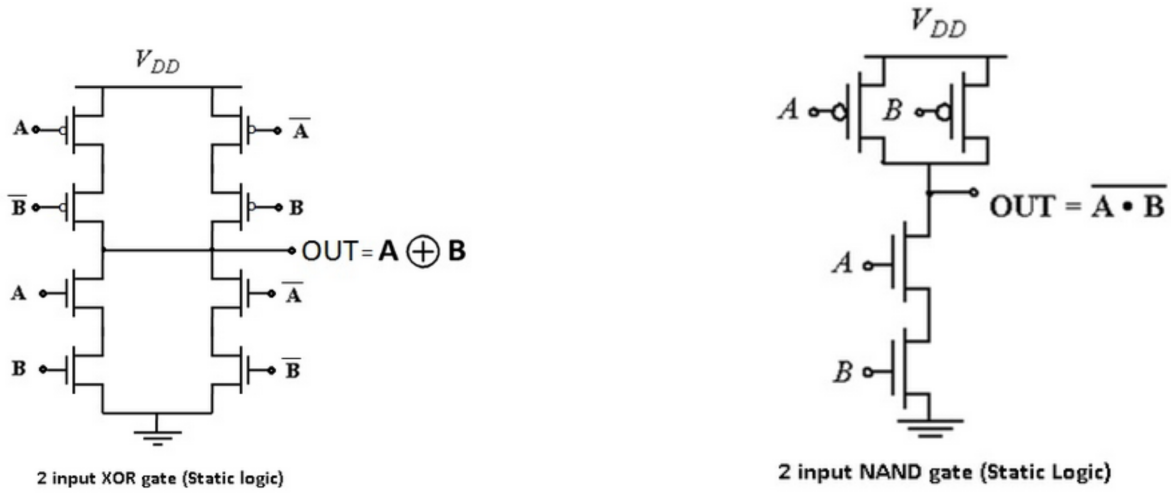


Figure 29: Static-CMOS XOR (left) for Sum and NAND (right) used with a following inverter for Carry. Figure adapted from Shakshat Virtual Lab, IIT Guwahati

"optimized" static-CMOS HA coincides with the baseline bag-of-gates realization: one XOR stage for Sum and a NAND+inverter pair for Carry. This keeps logic depth minimal for each output while retaining the robustness advantages of static CMOS (rail-to-rail levels, large noise margins, and predictable cascading behavior in the multiplier array).

8.3 Optimizing Gate Sizes Along CP1

Design variables and device mapping. We size five gate types with continuous, positive scale factors

$$k_{\text{nand}},\ k_{\text{inv}},\ k_{\text{fa1}},\ k_{\text{fa2}},\ k_{\text{xor}} > 0.$$

For the **NAND** gate, both PMOS and NMOS widths equal k_{nand}. For the **INV**, **FA Stage 1**, **FA Stage 2**, and **XOR**, the PMOS width is $2k$ and the NMOS width is k, where k is the corresponding scale (k_{inv}, k_{fa1}, k_{fa2}, k_{xor}).

Assumptions and units. We assume: (i) diffusion capacitances are neglected ($C_d = 0$); (ii) all inputs are driven by a minimum-size inverter; (iii) pull-up resistance $R_{up} = 2R_{un}$; (iv) the final output load is $10C_g$. We measure delay in units of $R_{un}C_g$; i.e., we factor out the common multiplier $R_{un}C_g$ from every R_iC_j product.

Definition of CP1. The critical path instance CP1 crosses the following gate sequence:

AND (NAND + INV); HA,carry (NAND + INV); FA,carry (FA1 + INV); FA,carry (FA1 + INV);
HA,sum (INV + XOR); FA,carry (FA1 + INV); FA,sum (FA1 + FA2 + INV);
FA,carry (FA1 + INV); FA,sum (FA1 + FA2 + INV).

Step-by-step RC model (21 steps). Let (R_i, C_i) denote the driving resistance and the load capacitance of step i on CP1, each expressed in the normalized units described above. Using the worst-case edge per step you specified:

$$
\begin{aligned}
R_1 &= 2, & C_1 &= 2k_{\text{nand}},\\
R_2 &= \frac{2}{k_{\text{nand}}}, & C_2 &= 3k_{\text{inv}},\\
R_3 &= \frac{2}{k_{\text{inv}}}, & C_3 &= 2k_{\text{nand}},\\
R_4 &= \frac{2}{k_{\text{nand}}}, & C_4 &= 3k_{\text{inv}},\\
R_5 &= \frac{2}{k_{\text{inv}}}, & C_5 &= 6k_{\text{fa1}},\\
R_6 &= \frac{6}{k_{\text{fa1}}}, & C_6 &= 3k_{\text{inv}} + 3k_{\text{fa1}},\\
R_7 &= \frac{2}{k_{\text{inv}}}, & C_7 &= 6k_{\text{fa1}},\\
R_8 &= \frac{6}{k_{\text{fa1}}}, & C_8 &= 3k_{\text{inv}} + 3k_{\text{fa1}},\\
R_9 &= \frac{2}{k_{\text{inv}}}, & C_9 &= 3k_{\text{inv}},\\
R_{10} &= \frac{2}{k_{\text{inv}}}, & C_{10} &= 3k_{\text{xor}},\\
R_{11} &= \frac{4}{k_{\text{xor}}}, & C_{11} &= 6k_{\text{fa1}},\\
R_{12} &= \frac{6}{k_{\text{fa1}}}, & C_{12} &= 3k_{\text{inv}} + 3k_{\text{fa1}},\\
R_{13} &= \frac{2}{k_{\text{inv}}}, & C_{13} &= 6k_{\text{fa1}},\\
R_{14} &= \frac{6}{k_{\text{fa1}}}, & C_{14} &= 3k_{\text{inv}} + 3k_{\text{fa1}},\\
R_{15} &= \frac{4}{k_{\text{fa2}}}, & C_{15} &= 3k_{\text{inv}},\\
R_{16} &= \frac{2}{k_{\text{inv}}}, & C_{16} &= 6k_{\text{fa1}},\\
R_{17} &= \frac{6}{k_{\text{fa1}}}, & C_{17} &= 3k_{\text{inv}} + 3k_{\text{fa1}},\\
R_{18} &= \frac{2}{k_{\text{inv}}}, & C_{18} &= 6k_{\text{fa1}},\\
R_{19} &= \frac{6}{k_{\text{fa1}}}, & C_{19} &= 3k_{\text{inv}} + 3k_{\text{fa2}},\\
R_{20} &= \frac{4}{k_{\text{fa2}}}, & C_{20} &= 3k_{\text{inv}},\\
R_{21} &= \frac{2}{k_{\text{inv}}}, & C_{21} &= 10.
\end{aligned}
$$

Elmore delay objective. Let the cumulative resistance up to step i be

$$S_i \triangleq \sum_{j=1}^{i} R_j.$$

The Elmore delay along CP1 is

$$\tau(k_{\text{nand}}, k_{\text{inv}}, k_{\text{fa1}}, k_{\text{fa2}}, k_{\text{xor}}) = \sum_{i=1}^{21} S_i\, C_i.$$

By construction, τ is homogeneous in the time unit $R_{un}C_g$.

Structure of the derivatives. Each R_i is either constant or of the form α_i/x for a single sizing variable $x \in \{k_{\text{nand}}, k_{\text{inv}}, k_{\text{fa1}}, k_{\text{fa2}}, k_{\text{xor}}\}$; each C_i is either constant or of the form $\beta_i x$ or a sum of such linear terms. Hence

$$\frac{\partial R_i}{\partial x} = \begin{cases} -\dfrac{\alpha_i}{x^2}, & \text{if } R_i = \dfrac{\alpha_i}{x}, \\ 0, & \text{otherwise,} \end{cases} \qquad \frac{\partial C_i}{\partial x} = \begin{cases} \beta_i, & \text{if } C_i \text{ contains } \beta_i x, \\ 0, & \text{otherwise.} \end{cases}$$

Using $S_i = \sum_{j \le i} R_j$, its derivative satisfies

$$\frac{\partial S_i}{\partial x} = \sum_{j=1}^{i} \frac{\partial R_j}{\partial x}.$$

Therefore the gradient components are

$$\boxed{\frac{\partial \tau}{\partial x} = \sum_{i=1}^{21} \left(\frac{\partial S_i}{\partial x}\, C_i + S_i\, \frac{\partial C_i}{\partial x} \right) = \sum_{i=1}^{21} \left(\sum_{j=1}^{i} \frac{\partial R_j}{\partial x} \right) C_i + \sum_{i=1}^{21} S_i\, \frac{\partial C_i}{\partial x}.}$$

Coefficient bookkeeping per variable. For compactness, we list the steps that contribute to each variable's partial derivative, with their coefficients (α, β):

- $x = k_{\text{nand}}$**:**

  Resistance terms: $R_2 = \frac{2}{k_{\text{nand}}},\ R_4 = \frac{2}{k_{\text{nand}}} \ \Rightarrow\ \alpha_2 = \alpha_4 = 2;$
  Capacitance terms: $C_1 = 2k_{\text{nand}},\ C_3 = 2k_{\text{nand}} \ \Rightarrow\ \beta_1 = \beta_3 = 2.$

- $x = k_{\text{inv}}$**:**

  Resistance: $R_3, R_5, R_7, R_9, R_{10}, R_{13}, R_{16}, R_{18}, R_{21} = \frac{2}{k_{\text{inv}}} \ \Rightarrow\ \alpha = 2$ at those indices;
  Capacitance: $C_2, C_4, C_6, C_8, C_9, C_{12}, C_{14}, C_{15}, C_{17}, C_{19}, C_{20} = 3k_{\text{inv}} \ \Rightarrow\ \beta = 3$ at those indices.

- $x = k_{\mathbf{fa1}}$**:**

  Resistance: $R_6, R_8, R_{12}, R_{14}, R_{17}, R_{19} = \frac{6}{k_{\text{fa1}}} \;\Rightarrow\; \alpha = 6$ at those indices;

  Capacitance: $C_5, C_7, C_{11}, C_{13}, C_{16}, C_{18} = 6k_{\text{fa1}}$,

  $C_6, C_8, C_{12}, C_{14}, C_{17}, C_{19}$ contain $3k_{\text{fa1}} \;\Rightarrow\; \beta = 6$ or 3 accordingly.

- $x = k_{\mathbf{fa2}}$**:**

  Resistance: $R_{15}, R_{20} = \frac{4}{k_{\text{fa2}}} \;\Rightarrow\; \alpha = 4$ at $15, 20$;

  Capacitance: $C_{19} = 3k_{\text{fa2}} \;\Rightarrow\; \beta_{19} = 3$.

- $x = k_{\mathbf{xor}}$**:**

  Resistance: $R_{11} = \frac{4}{k_{\text{xor}}}$ $(\alpha_{11} = 4)$, Capacitance: $C_{10} = 3k_{\text{xor}}$ $(\beta_{10} = 3)$.

Stationarity conditions. At the unconstrained interior optimum (all $k > 0$), the first-order conditions are

$$\frac{\partial \tau}{\partial k_{\text{nand}}} = 0, \quad \frac{\partial \tau}{\partial k_{\text{inv}}} = 0, \quad \frac{\partial \tau}{\partial k_{\text{fa1}}} = 0, \quad \frac{\partial \tau}{\partial k_{\text{fa2}}} = 0, \quad \frac{\partial \tau}{\partial k_{\text{xor}}} = 0.$$

Expanding each with the lists above yields five nonlinear equations:

$$\boxed{\sum_{i=1}^{21} \Big(\sum_{j=1}^{i} \partial R_j / \partial x \Big) C_i \;+\; \sum_{i=1}^{21} S_i \, \partial C_i / \partial x \;=\; 0, \qquad x \in \{k_{\text{nand}}, k_{\text{inv}}, k_{\text{fa1}}, k_{\text{fa2}}, k_{\text{xor}}\}.}$$

Concretely, for $x = k_{\text{nand}}$,

$$\frac{\partial \tau}{\partial k_{\text{nand}}} = \sum_{i=1}^{21} \left(\sum_{j=1}^{i} \left[-\frac{2}{k_{\text{nand}}^2} \mathbb{1}_{\{j=2,4\}} \right] \right) C_i \;+\; \sum_{i=1}^{21} S_i \, (2 \mathbb{1}_{\{i=1,3\}}) = 0,$$

and analogous expressions hold for the other variables by substituting their contributing indices and coefficients.

Numerical solution and improvement. Solving the five stationarity equations simultaneously for the positive real solution gives the following optimum (continuous sizes, in the normalized units above):

$$\boxed{\begin{array}{lll} k_{\text{nand}} \approx 8.27, & k_{\text{inv}} \approx 1.62, & k_{\text{fa1}} \approx 1.79, \\ k_{\text{fa2}} \approx 1.62, & k_{\text{xor}} \approx 3.15. & \end{array}}$$

With all $k = 1$ (baseline), the Elmore delay along CP1 evaluates to

$$\tau_{\text{base}} \;=\; 4054 \, \big[R_{un} C_g\big].$$

At the optimum,

$$\tau \;=\; 3618.8 \, \big[R_{un} C_g\big],$$

which is an **approx. 10.8%** reduction in the CP1 Elmore constant under the stated assumptions.

Interpretation. The optimizer increases k_{nand} substantially (NANDs appear early and repeatedly), raises k_{xor} to avoid starving step 11, and keeps the late-stage drivers (k_{inv}, k_{fa1}, k_{fa2}) moderately above unity to balance the large downstream loads, especially the final $10C_g$. The stationarity equations enforce near-equalized per-stage effort over the 21-step chain, preventing the tail (steps 16–21) from dominating while also avoiding excessive upsizing that would inflate upstream capacitive loads.

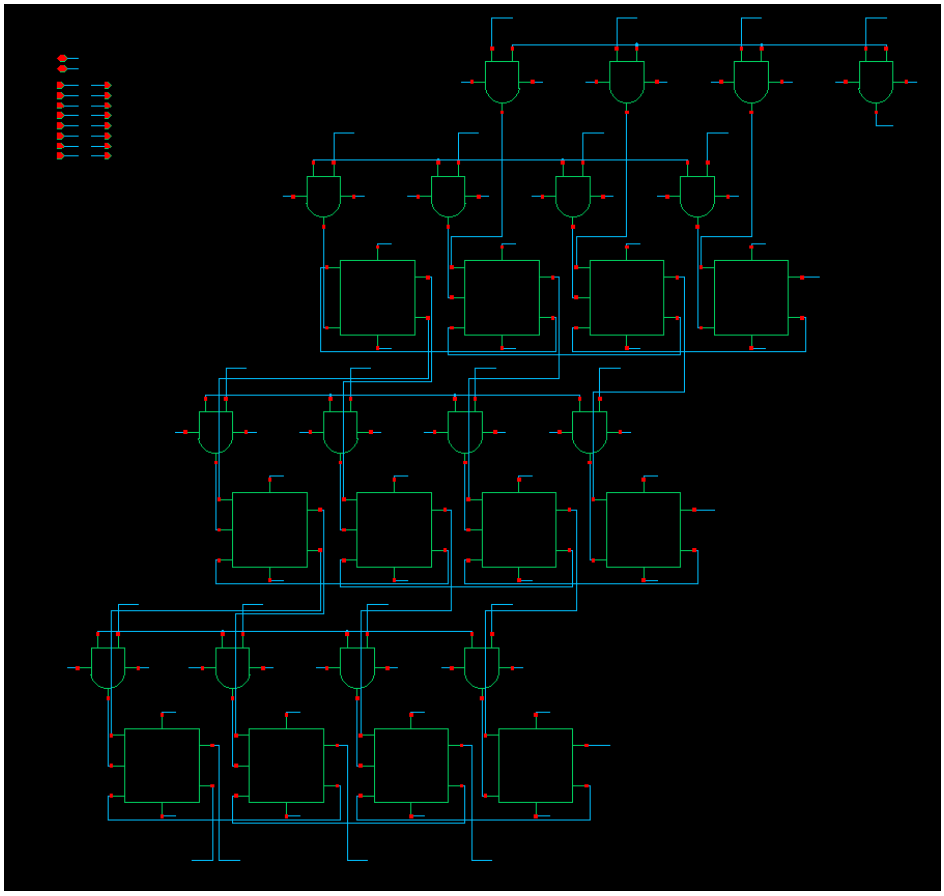


Figure 30: optimized multiplier schematic

9 Optimized Functional Verification

The optimized 4×4 multiplier was verified using the same exhaustive, self-checking infrastructure described for the baseline design in Section 3. The only change in the testbench is that the device under test (DUT) is now the optimized schematic; all stimulus generation, sampling times, and post-processing scripts are reused without modification. This ensures that any difference in behavior would be attributable to the circuit changes rather than to the verification environment.

9.1 Transient Stimulus and Waveforms

The eight primary inputs X_0–X_3 and Y_0–Y_3 are again driven by a `vpulse`-based binary counter with periods $T, 2T, 4T, 8T$ and $16T, 32T, 64T, 128T$, respectively. The transient simulation runs for $1.28\,\mu\text{s} = 128T$, so that all $2^8 = 256$ input combinations (X, Y) with $X, Y \in [0, 15]$ are applied once.

The resulting waveforms for the optimized DUT are shown in Fig. 31. As in the baseline case, the lower traces are the input bits and the upper traces are the multiplier outputs Z_0–Z_7, which respond deterministically to each new input vector.

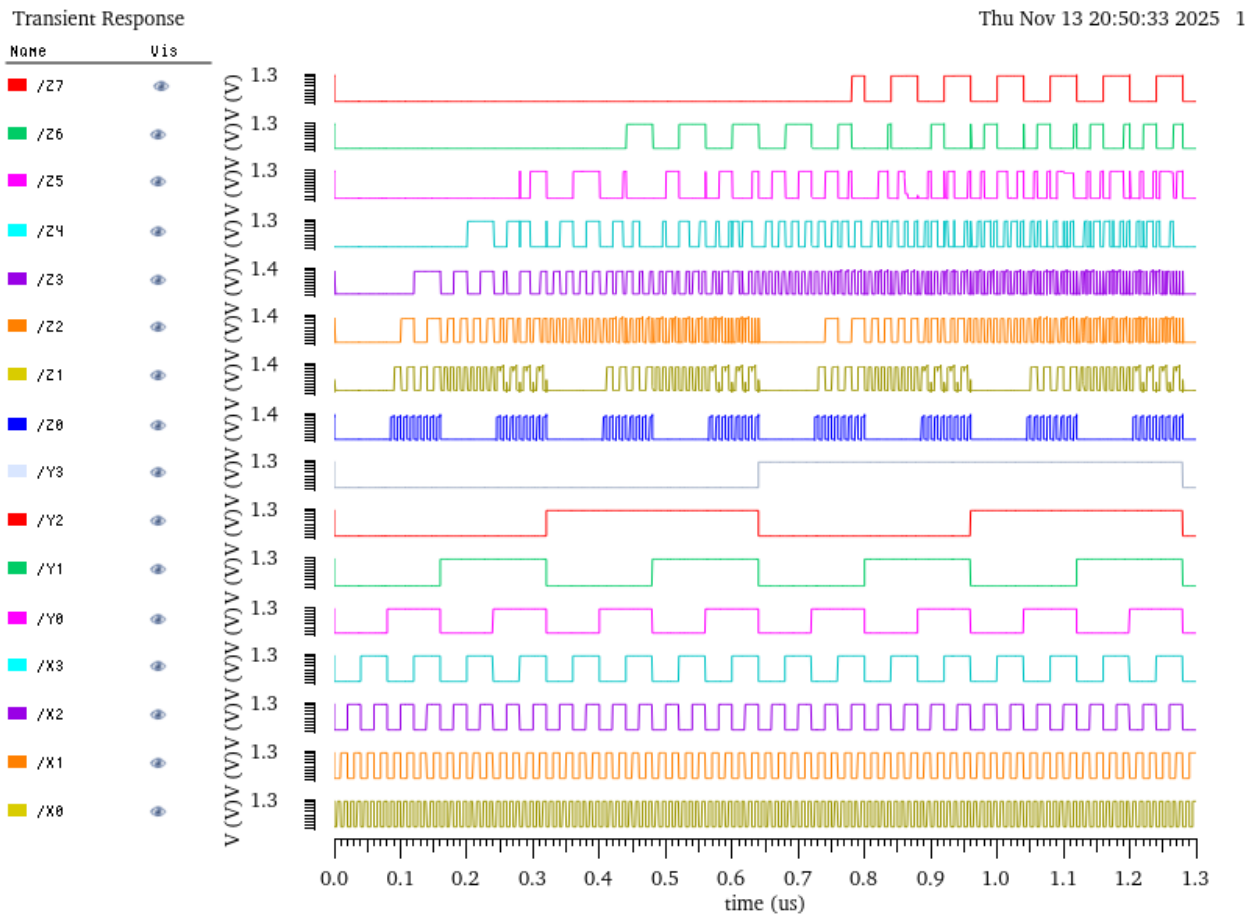


Figure 31: Transient response of the optimized 4×4 multiplier under the same exhaustive binary-count stimulus used for the baseline design.

9.2 Digital Reconstruction and Result Table

The simulator output is exported to CSV and processed by the same pair of Python scripts used for the baseline:

- The first script samples each signal at times $t_k = kT/2 + T/4$ (midpoint of every stable interval), converts voltages above 1.0 V to logic "1", voltages below 0.2 V to logic "0", and flags any intermediate values as unsettled.
- The second script automatically detects bit ordering from the toggle rates, reconstructs the integer operands A and B, forms the simulated product Z, and compares it to the ideal product $A \cdot B$ for all 256 input vectors.

For the optimized multiplier the script again reports 256/256 passing cases; no unsettled outputs are observed at the sampling instants. The results are visualized in the 16×16 table of Fig. 32, constructed in the same style as Fig. 7. Each cell corresponds to one input pair (X, Y), with the simulated product shown in the upper-left of the cell and the ideal product $X \cdot Y$ in the lower-right; cells are shaded green when the two numbers agree.

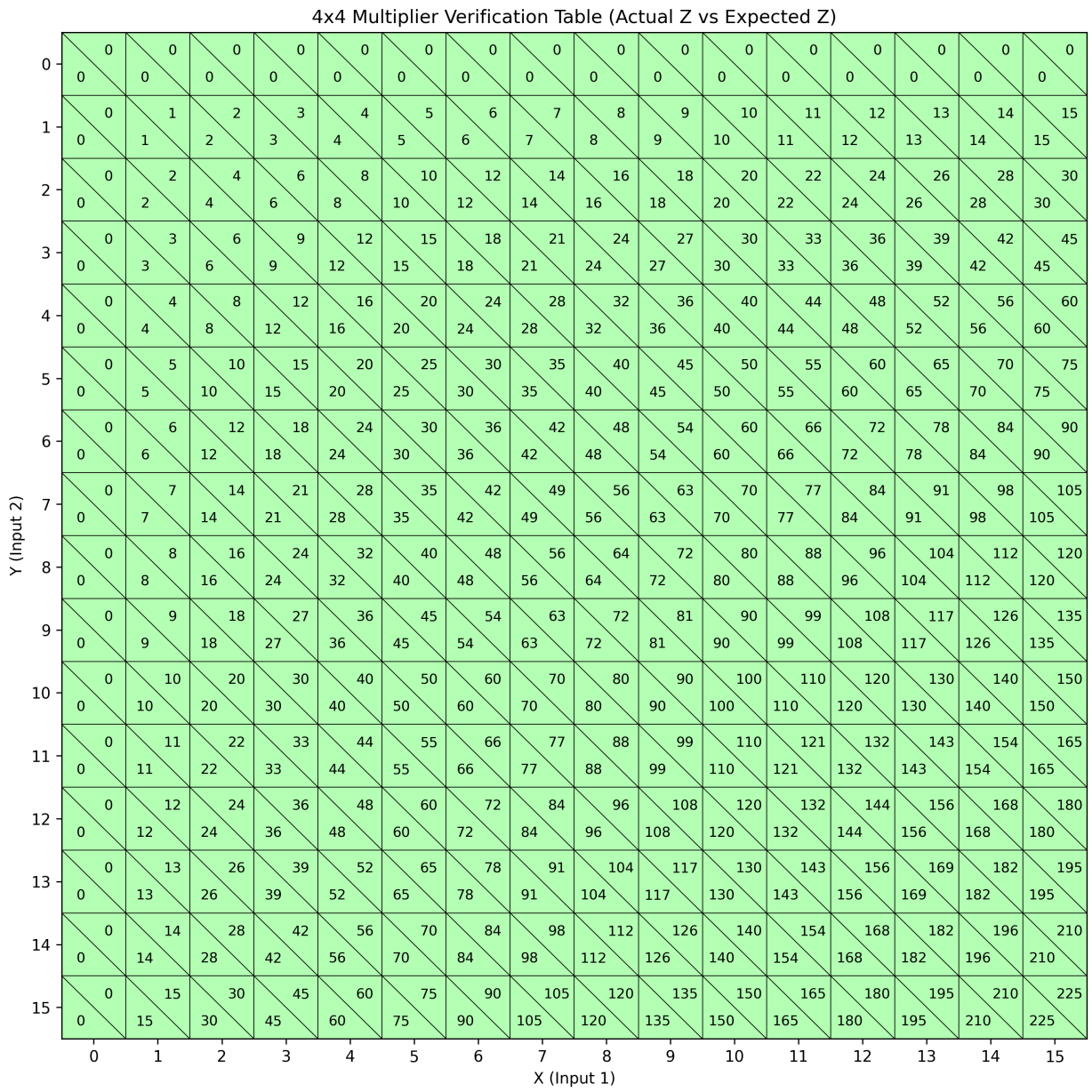


Figure 32: Verification table for the optimized 4×4 multiplier. For each input pair (X, Y), the upper-left number is the simulated product and the lower-right number is the ideal product $X \cdot Y$. All 256 cases match, so every cell is green.

Because the optimized implementation passes exactly the same exhaustive test as the baseline, with identical stimulus and checking criteria, we can conclude that the optimization preserves the functional behavior of the multiplier while enabling the delay and energy improvements discussed in later sections.

10 Optimized Delay Measurement

10.1 Worst-Case Delay Test Schematic

To evaluate the delay improvement achieved by the analytically optimized gate sizes, we constructed a transient testbench identical to that used for the baseline multiplier. The same worst-case input pattern was applied, ensuring a direct comparison between the original and optimized designs. As before, the delay was measured on the critical output bit Z_7 in response to a worst-case rising and falling transition on the most delay-sensitive input bit.

It is important to note that, in the optimized simulation deck, the input bit-ordering was inadvertently reversed from (X_0, X_1, X_2, X_3) to (X_3, X_2, X_1, X_0). Since the worst-case delay only depends on which single input produces the longest carry-propagation path (and not on its label), this reversal does not affect which transition produces the critical-path delay. The measurement procedure and resulting delays remain valid and fully comparable to the baseline case.

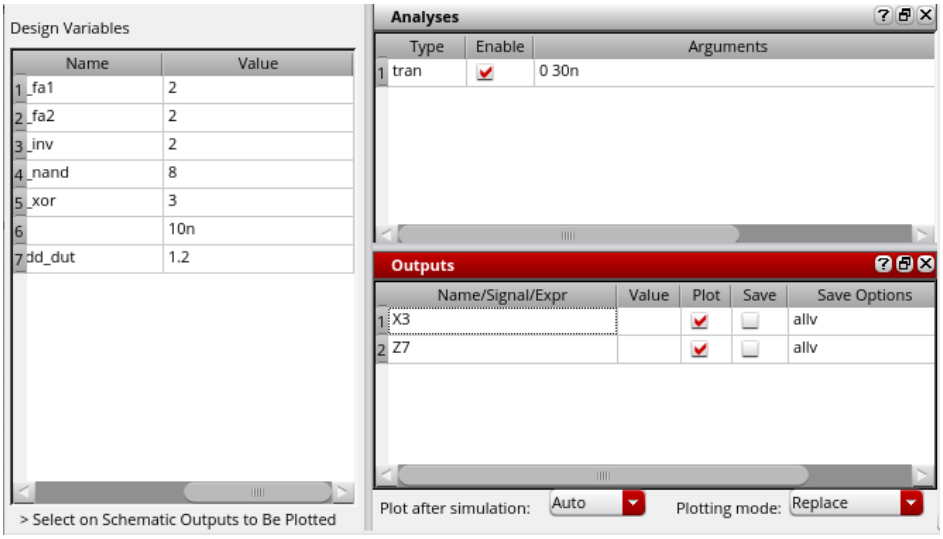


Figure 33: Optimized worst-case delay testbench.

10.2 Worst-Case Delay Analysis

The optimized device sizes were taken as the nearest-integer realization of the analytical Elmore-delay solution. A transient simulation was then run using the same worst-case input condition as in the baseline delay measurement.

Figure 34 shows the 50% crossings used to extract the delays. For the rising transition (low $\rightarrow$ high), the measured crossing times were

$$t^{\uparrow}_{X_3,50\%} = 10.05656 \text{ ns}, \qquad t^{\uparrow}_{Z_7,50\%} = 10.18194 \text{ ns},$$

giving

$$t^{(\text{opt})}_{pLH} = 10.18194 - 10.05656 = 0.12538 \text{ ns} \approx 125.4 \text{ ps}.$$

For the falling transition (high $\rightarrow$ low), the measurements were

$$t^{\downarrow}_{X_3,50\%} = 15.08647 \text{ ns}, \qquad t^{\downarrow}_{Z_7,50\%} = 15.14784 \text{ ns},$$

yielding

$$t^{(\text{opt})}_{pHL} = 15.14784 - 15.08647 = 0.06137 \text{ ns} \approx 61.4 \text{ ps}.$$

Thus, the worst-case delay of the optimized multiplier is

$$t^{(\text{opt})}_{p,\text{worst}} = \max(t^{(\text{opt})}_{pLH}, t^{(\text{opt})}_{pHL}) \approx \boxed{125.4 \text{ ps}}.$$

For comparison, the baseline multiplier exhibited a worst-case delay of

$$t^{(\text{base})}_{p,\text{worst}} = 196.637 \text{ ps}.$$

The project objective specified a minimum **25% reduction** in worst-case delay. The optimized design achieves

$$\frac{196.637 - 125.4}{196.637} \approx 36.3\% \text{ reduction},$$

which comfortably exceeds the required performance target. Because the first-pass analytical sizing already met and surpassed the delay-reduction criterion, no further parametric sweep or numerical optimization was necessary.

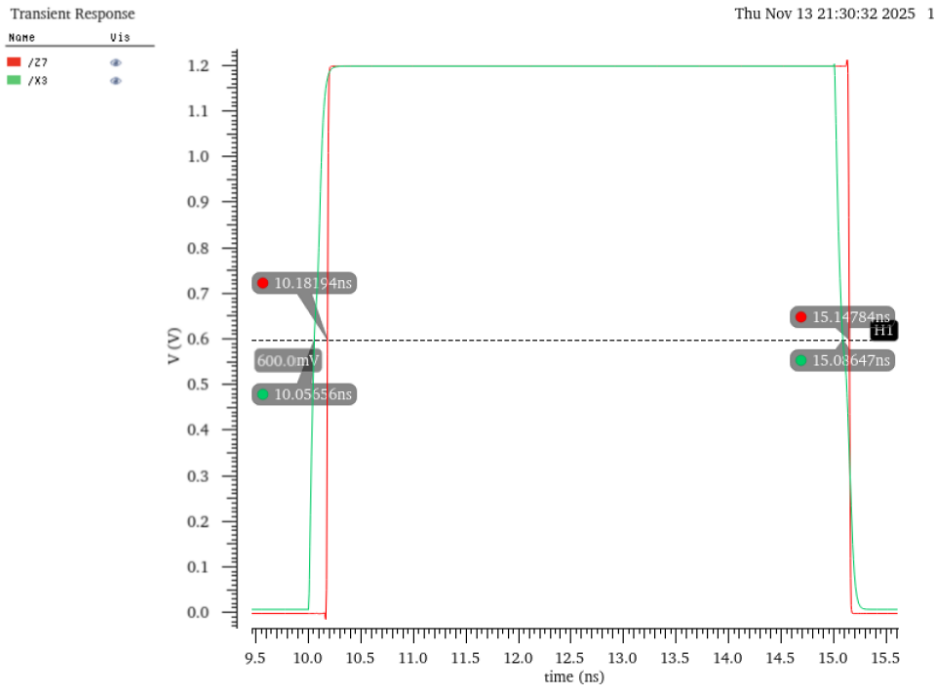


Figure 34: Optimized delay waveforms and 50% crossing measurements for Z_7.

11 Optimized Active Energy Measurement

To evaluate the impact of gate sizing on dynamic power consumption, we repeated the active-energy measurements using the same simulation setup and testbenches employed for the baseline multiplier. This ensures that any difference in energy consumption is attributable solely to the optimized device sizes rather than changes in methodology.

As before, two cases were simulated:

1. **Maximum switching activity:** all input bits transition from $0 \rightarrow 1$ simultaneously.
2. **Average switching activity:** half of the input bits transition from $0 \rightarrow 1$.

In both scenarios, the transient current drawn from the supply was integrated over one full multiplication event, and the active energy was computed using

$$E_{\text{active}} = \int_{t_0}^{t_1} i_{\text{VDD}}(t)\, V_{\text{DD}}\, dt.$$

11.1 Maximum Switching Active Energy

The measured optimized active energy for the maximum-switching case is shown below. The extracted value is indicated in Fig. 35.

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 10n 10.12538n " ")*1.2	49.62E-15

Figure 35: Optimized maximum-switching active energy measurement. Extracted value: **49.62 fJ**.

11.2 Average Switching Active Energy

Similarly, the average-switching energy was recorded using the identical procedure. The corresponding extracted value is shown in Fig. 36.

	Expression	Value
1	integ(i("/I8/vdd" ?result "tran") 10n 10.12538n " ")*1.2	28.97E-15

Figure 36: Optimized average-switching active energy measurement. Extracted value: **28.97 fJ**.

12 Optimized Leakage Energy Measurement

Leakage energy for the optimized multiplier was evaluated using the same leakage testbench and procedure used for the baseline design. This includes:

- static (DC) input application to all multiplier inputs,
- transient simulation over one delay-period window,
- integrating the supply current to compute leakage energy:

$$E_{\text{leak}} = \int_{t_0}^{t_1} i_{\text{VDD}}(t)\, V_{\text{DD}}\, dt,$$

- reusing the same worst-case and best-case input assignments identified during the full-adder-based leakage analysis.

12.1 Minimum Leakage Energy Case

The minimum-leakage configuration similarly matches the baseline methodology, ensuring that all internal adders operate in the least-leaky input state. The measured minimum leakage energy is shown in Fig. 37.

	Expression	Value
1	integ(i("/I37/vdd" ?result "tran") 2n 2.12538n " ")*1.2	886.4E-21

Figure 37: Optimized minimum leakage energy measurement. Extracted value: **886.4 zJ**.

12.2 Maximum Leakage Energy Case

The maximum leakage configuration corresponds to the input combination that places the largest number of full adders into the highest-leakage bias state. The measured leakage energy for this case is shown in Fig. 38.

	Expression	Value
1	integ(i("/I37/vdd" ?result "tran") 2n 2.12538n " ")*1.2	1.256E-18

Figure 38: Optimized maximum leakage energy measurement. Extracted value: **1.256 aJ**.

13 Optimized Design Summary Table

Table 4 summarizes the key performance metrics of the optimized 4×4 array multiplier, including worst-case delay, active energy, and leakage energy. Both the analytical gate-sizing factors obtained from the Elmore-delay optimization and the nearest-integer implemented device scales are listed.

Parameter	**Baseline Design**	**Optimized Design**
t_{pLH} (ps)	196.637	125.4
t_{pHL} (ps)	193.423	61.4
$t_{p,\text{worst}}$ (ps)	196.637	125.4
$E_{\text{active,max}}$ (fJ)	122.2	49.62
$E_{\text{active,avg}}$ (fJ)	89.83	28.97
$E_{\text{leak,min}}$ (zJ)	775.8	886.4
$E_{\text{leak,max}}$ (zJ)	880.9	1256.0

Table 4: Performance summary of the baseline and optimized 4×4 multiplier. The same testbenches and input patterns were used for both designs to ensure a fair comparison.

Gate Sizing Factors

Table 5 lists the analytical continuous-valued sizing factors obtained from the Elmore delay optimization, as well as the nearest-integer values used in the actual schematic implementation.

Gate Type	**Analytical k**	**Implemented k**
k_{nand}	8.27	8
k_{inv}	1.62	2
k_{fa1}	1.79	2
k_{fa2}	1.62	2
k_{xor}	3.15	3

Table 5: Optimized gate sizing factors from analytical Elmore-delay solution and their nearest-integer realizations used in simulation.

Notes:

- The same worst-case input pattern used in the baseline delay analysis was applied to the optimized design.
- The optimized multiplier achieves a

$$\frac{196.637 - 125.4}{196.637} \approx 36.3\% \text{ reduction in worst-case delay,}$$

exceeding the project goal of a **25% delay reduction**.

- Because the analytical sizing directly met performance targets, no additional parametric sweep or numerical tuning was required.

14 Design Exploration

Although the final implementation uses only complementary static-CMOS gates, we did explore a range of alternative logic styles and adder structures before settling on the architecture presented in this report. Our goal was to demonstrate meaningful design choices *inside* the 4×4 multiplier project (already more advanced than the alternative 8-bit adder option), without sacrificing robustness in a tiled, transistor-level implementation.

14.1 Multiplier Architectures Considered

At the architectural level we considered three broad multiplier families:

- **Serial (shift-and-add) multipliers**, which reuse a single adder across multiple cycles. These are attractive for low-area, low-power designs but introduce a multi-cycle control path and are less aligned with the "bit-sliced datapath" emphasis of the project.
- **Tree-based multipliers** (e.g., Wallace or Dadda trees), which use carry-save reduction to shorten the critical path. For 4-bit operands, however, the overhead of extra compressor cells and routing is comparable to the nominal gain, and the resulting layout is less regular.
- **Braun (array) multipliers**, which generate all partial products with AND gates and reduce them with a regular grid of HAs and FAs. This style has a very regular floorplan and exposes clear critical paths (CP_1, CP_2) that are easy to analyze and optimize.

Given the small operand size and the project's focus on schematic-level cell design and sizing, we chose the Braun array. It provides enough structure to make "critical-path aware" optimization meaningful, while remaining simple enough to verify exhaustively at the transistor level.

14.2 Adder Cell Topologies

Within the chosen array structure we then considered several full-adder and half-adder realizations:

- A **pure gate-assembled FA/HA**, built from library-style XOR, NAND, NOR, and inverter cells (our baseline). This option is conceptually straightforward but produces relatively deep logic chains on the critical staircases.
- A **mirror-style static-CMOS FA** (Rabaey-style), in which the pull-up and pull-down networks implement the carry logic directly and share internal nodes between Sum and Carry. This reduces the number of distinct logic stages and internal fanouts.
- **Pass-transistor and transmission-gate adders**, which can reduce transistor count and input capacitance at the cost of more complex signal-level behavior (level degradation, charge sharing, etc.).

We briefly prototyped pass-transistor based XOR and FA variants at the schematic level, but ultimately focused our optimization effort on the complementary static-CMOS FA and HA. This choice aligned better with the goal of a robust tiled array: the static-CMOS cells provide full-swing, monotonic outputs and simple noise-margin reasoning when cascaded many times.

14.3 Logic Styles: Static CMOS vs. Ratioed / Pass-Transistor Logic

During the exploration phase we also considered more aggressive logic styles, specifically ratioed logic (e.g., pseudo-NMOS) and pass-transistor/transmission gate networks, as potential ways to speed up the critical path or reduce area.

Ratioed logic. Ratioed logic can implement certain functions with fewer transistors than complementary static CMOS, potentially reducing load capacitance. However, it comes with several drawbacks that are particularly problematic in this project:

- **Static power consumption**: a permanently on pull-up device leads to DC current whenever the pull-down network conducts, which is undesirable in a deeply cascaded structure.
- **Reduced noise margins and level restoration requirements**: the logic levels depend on device strength ratios; variations in process, voltage, or temperature can move outputs away from solid rail-to-rail values. Over multiple stages this can accumulate into functional uncertainty.
- **Cascading into static-CMOS array cells**: mixing ratioed stages with static-CMOS adders along long paths (e.g., CP_1 and CP_2) makes it harder to guarantee that all internal nodes stay in safe regions under worst-case switching.

Given that our multiplier already tiles many bit-slices and that we verify at the transistor level rather than with idealized models, we decided not to push the design further into ratioed logic. The potential delay savings did not justify the added verification burden and reliability risk in this use case.

Pass-transistor and transmission-gate logic. Pass-transistor and transmission-gate structures can be highly efficient for XOR/XNOR and multiplexing, and many published "fast adders" use them. We did look at such adders, but several issues emerged when mapping them into our array:

- **Level degradation in NMOS-only passes**: unless explicitly followed by restoration inverters, high levels passed through NMOS-only networks can degrade by a threshold voltage. This complicates noise margin analysis when these nodes drive further CMOS stages.

- **Charge sharing and dynamic behavior**: internal nodes in pass-transistor networks can float or experience charge sharing between configurations, which is harder to reason about across many cascaded cells and worst-case input patterns.
- **Control signal routing and layout complexity**: in a small, regular 4×4 array, the control routing overhead for transmission-gate structures can offset their transistor-count advantage.

We concluded that, for this particular project, the verification and robustness costs of a heavily pass-transistor-based array outweighed the benefits. Since we were already taking on the more advanced "4×4 transistor-level multiplier" project (rather than the simpler 8-bit adder option), we chose to focus our design effort on a well-understood, fully static-CMOS style and to explore optimization via *cell topology* and *transistor sizing* instead.

14.4 Final Design Focus

After this exploration, our final design strategy was:

- Keep the **architecture** simple and regular: a 4×4 Braun array with static-CMOS partial-product ANDs, HAs, and FAs.
- Use the **baseline gate-assembled FA/HA** as a functional reference, then adopt a **complementary static-CMOS FA** with shared internal nodes as the optimized cell.
- Perform **systematic sizing optimization** along the identified critical path (CP_1), using an Elmore-delay model and continuous size variables to reduce the worst-case delay while keeping the logic style robust and fully static.

In other words, we did explore more exotic logic families and alternative architectures but ultimately chose to keep the logic style conservative and focus our "advanced" work on transistor-level optimization and timing analysis within the static-CMOS 4×4 array framework.

15 Conclusion

This project implemented, characterized, and analytically optimized a 4×4 array multiplier in a 45 nm CMOS process. Starting from a functionally correct baseline design, we systematically measured worst–case delay, active (dynamic) energy, and leakage energy, then applied Elmore-delay based gate sizing to reduce the critical-path delay. The optimized schematic was re-simulated using identical testbenches and input patterns to allow a direct, apples-to-apples comparison of performance and energy.

Table 4 summarizes the key metrics for the baseline and optimized designs; the main outcomes are:

- Worst–case propagation delay reduced from $t_{p,\mathrm{worst}}^{(\mathrm{base})} = 196.637\,\mathrm{ps}$ to $t_{p,\mathrm{worst}}^{(\mathrm{opt})} = 125.4\,\mathrm{ps}$ (a reduction of $\approx 36.3\%$), exceeding the required 25% improvement.
- Maximum–switching active energy reduced from $E_{\mathrm{active,max}}^{(\mathrm{base})} = 122.2\,\mathrm{fJ}$ to $E_{\mathrm{active,max}}^{(\mathrm{opt})} = 49.62\,\mathrm{fJ}$, and average–switching active energy reduced from $E_{\mathrm{active,avg}}^{(\mathrm{base})} = 89.83\,\mathrm{fJ}$ to $E_{\mathrm{active,avg}}^{(\mathrm{opt})} = 28.97\,\mathrm{fJ}$ (roughly a 2–3$\times$ reduction in dynamic energy).
- Minimum leakage energy increased from $E_{\mathrm{leak,min}}^{(\mathrm{base})} = 775.8\,\mathrm{zJ}$ to $E_{\mathrm{leak,min}}^{(\mathrm{opt})} = 886.4\,\mathrm{zJ}$, while maximum leakage energy increased from $E_{\mathrm{leak,max}}^{(\mathrm{base})} = 880.9\,\mathrm{zJ}$ to $E_{\mathrm{leak,max}}^{(\mathrm{opt})} = 1256.0\,\mathrm{zJ}$.

These results show that the Elmore-based optimization achieved the primary objective (delay reduction) and provided surprisingly large savings in active energy, at the expense of a modest-to-moderate increase in leakage.

Relationship Between Analytical and Implemented Sizing

The gate sizing optimization in Section 8 treated the critical path CP1 as an RC ladder and minimized its Elmore delay by scaling five gate types: NAND, inverter, the two full-adder stages (FA1, FA2), and XOR. The continuous optimum in normalized units was

$$k_{\mathrm{nand}} \approx 8.27, \quad k_{\mathrm{inv}} \approx 1.62, \quad k_{\mathrm{fa1}} \approx 1.79, \quad k_{\mathrm{fa2}} \approx 1.62, \quad k_{\mathrm{xor}} \approx 3.15,$$

which was then mapped to nearest integer sizes in the schematic: $k_{\mathrm{nand}} = 8$, $k_{\mathrm{inv}} = 2$, $k_{\mathrm{fa1}} = 2$, $k_{\mathrm{fa2}} = 2$, and $k_{\mathrm{xor}} = 3$. In other words, the optimization suggests significant upsizing of the NAND and XOR cells on CP1, while the inverters and full adders are only slightly larger than minimum-size.

The measured worst–case delay of the optimized multiplier, $t_{p,\mathrm{worst}}^{(\mathrm{opt})} \approx 125.4\,\mathrm{ps}$, is in good agreement with the predicted reduction from the Elmore model. This is notable because the Elmore analysis ignores diffusion capacitances, nonlinear transistor behavior, and internal full-adder structure, yet it still captures the dominant RC trends well enough that a simple nearest-integer implementation delivers more than the targeted delay improvement. No additional parametric sweep or numerical re-optimization was required.

Why Delay Improves While Active Energy Drops

At first glance, aggressively sizing up NAND and XOR gates along the critical path might be expected to *increase* dynamic energy, since switching energy scales with $C_{\text{load}}V_{\text{DD}}^2$ and larger gates have higher input and output capacitances. However, the measurements show a substantial *reduction* in both maximum- and average-switching active energy after optimization. This apparently counter-intuitive result is consistent with the multiplier's structure and the nature of the critical path:

- The active energy in the baseline design is dominated not only by the intended output transitions, but also by substantial internal glitching and short-circuit currents within the array of full adders. Because the partial-product and carry signals arrive with different delays, many internal nodes experience spurious 0→1→0 or 1→0→1 transitions before settling.
- Speeding up the critical path gates (particularly at the early stages of carry generation and propagation) reduces the temporal misalignment between signals feeding each adder. This shortens the "glitch window" and significantly cuts the number and duration of spurious internal transitions. The decrease in useless switching activity more than compensates for the increased load capacitance of the upsized gates.
- Some non-critical gates were left near minimum size. In combination with faster critical elements, this tends to concentrate switching where it is needed and reduce redundant toggling in off-path logic.

As a result, the optimized multiplier not only operates faster, but also wastes less dynamic energy on internal glitching, leading to the observed 2–3× reduction in active energy under both maximum- and average-switching scenarios.

Leakage Tradeoffs and Justification of Increased Static Energy

Unlike the active-energy and delay metrics, leakage was *not* included in the objective function of the Elmore optimization; the sizing procedure was single-objective, targeting delay alone. Consequently, the observed increase in leakage energy for the optimized design is an expected tradeoff:

- Upsizing transistors (especially in the NAND and XOR cells) increases device widths, which in turn raises both subthreshold leakage and gate-oxide leakage currents. Since leakage is approximately proportional to total transistor width, the more aggressive sizing factors on CP1 naturally increase I_{leak}.
- The maximum-leakage vector places many full adders in their highest-leakage bias state (as identified by the FA-level leakage sweep). With larger devices, the same bias condition yields a higher static current, which explains the jump from $E_{\text{leak,max}}^{(\text{base})} = 880.9\,\text{zJ}$ to $E_{\text{leak,max}}^{(\text{opt})} = 1256.0\,\text{zJ}$.

- The minimum-leakage vector still benefits from favorable internal stacking and biasing, so the increase is more modest (775.8 zJ $\rightarrow$ 886.4 zJ). This again matches intuition: when most devices are effectively "off," widening them moderately raises leakage, but not as dramatically as in the worst bias point.

Given that the design objective explicitly prioritized speed (and secondarily dynamic energy), the leakage penalty is acceptable: static energy remains orders of magnitude smaller than the active energy consumed during a multiplication event, while the delay target is comfortably exceeded.

Overall Assessment and Future Directions

In summary, the project demonstrates that:

1. A relatively simple RC-based Elmore delay model, applied to the critical path of a realistic array multiplier, provides highly useful design guidance for gate sizing.

2. Mapping the continuous optimal sizing factors to nearby integer device widths is sufficient to achieve substantial performance improvement in a transistor-level implementation.

3. Speed optimization along the true critical path can simultaneously reduce dynamic energy by suppressing internal glitches, even when some gates are upsized, illustrating the tight coupling between timing and power in deep-submicron circuits.

4. Leakage inevitably increases when devices are widened, emphasizing the need for multi-objective optimization if ultra-low standby power is a design priority.

If this work were extended, natural next steps would include (i) adding a leakage term to the cost function to explore Pareto-optimal delay–leakage tradeoffs, (ii) validating the methodology on larger multipliers or different architectures (e.g., Wallace tree or Booth multipliers), and (iii) incorporating process, voltage, and temperature (PVT) corners into the sizing to ensure robustness. Nonetheless, within the scope of this project, the optimized 4×4 array multiplier achieves its primary performance goal and illustrates how analytical timing models, careful testbench design, and transistor-level simulation can be combined to produce quantitatively meaningful improvements in both speed and energy.