

# Comparative Performance Analysis and Implementation of 8-bit Multiplier Architectures using 45nm Technology

A Complete Design Flow from RTL to Custom Schematic  
Implementation

Kruthi Narayana Swamy

Department of Electrical and Computer Engineering

Northeastern University

`knarayanaswamy@northeastern.edu`

April 14th, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Project Objectives . . . . .	5
1.3	Contributions . . . . .	5
<b>2</b>	<b>Theoretical Background</b>	<b>5</b>
2.1	Binary Multiplication Fundamentals . . . . .	5
2.2	Array Multiplier . . . . .	6
2.3	Wallace Tree Multiplier . . . . .	6
2.4	Modified Booth Multiplier . . . . .	6
<b>3</b>	<b>RTL Design and Implementation</b>	<b>7</b>
3.1	Development Environment . . . . .	7
3.2	Array Multiplier Implementation . . . . .	7
3.3	Wallace Tree Multiplier Implementation . . . . .	8
3.4	Modified Booth Multiplier Implementation . . . . .	9
<b>4</b>	<b>Functional Verification</b>	<b>11</b>
4.1	Comprehensive Testbench . . . . .	11
4.2	Verification Results . . . . .	12
<b>5</b>	<b>Synthesis Results and Analysis</b>	<b>13</b>
5.1	Synthesis Configuration . . . . .	13
5.2	Synthesis Results Comparison . . . . .	14
5.3	Critical Path Analysis . . . . .	14
<b>6</b>	<b>Custom Schematic Design</b>	<b>14</b>
6.1	Technology Platform . . . . .	14
6.2	Transistor-Level Implementation . . . . .	15
6.3	Performance Results . . . . .	15
<b>7</b>	<b>Comparative Analysis</b>	<b>20</b>
7.1	Performance Summary . . . . .	20
7.2	Analysis of Results . . . . .	21
7.2.1	Why Array Outperformed Complex Architectures . . . . .	21
7.2.2	Comparison with Real-World Implementations . . . . .	21
7.3	Crossover Point Analysis . . . . .	23

---

<b>8</b>	<b>Conclusions and Future Work</b>	<b>23</b>
8.1	Key Findings . . . . .	23
8.2	Design Recommendations . . . . .	23
8.3	Future Work . . . . .	24
<b>9</b>	<b>Acknowledgments</b>	<b>24</b>
<b>10</b>	<b>References</b>	<b>24</b>

### Abstract

This project presents a comprehensive analysis and implementation of three 8-bit multiplier architectures: Array, Wallace Tree, and Modified Booth multipliers. The complete design flow encompasses RTL specification, functional verification, logic synthesis using FreePDK45 technology, and custom transistor-level implementation using GPDK045.

Key findings reveal that the Array multiplier unexpectedly outperformed more complex architectures at 8-bit width, achieving the smallest area ( $696.44 \mu\text{m}^2$ ), competitive delay (1.46 ns), and reasonable power consumption (366  $\mu\text{W}$ ) in synthesis. The custom schematic implementation of the Array multiplier demonstrated dramatic improvements with 217 ps delay and 30.6  $\mu\text{W}$  power consumption, representing a  $542\times$  improvement in Energy-Delay Product compared to synthesis results.

# 1 Introduction

## 1.1 Motivation

Digital multiplication is a fundamental arithmetic operation in modern computing systems. The efficiency of multiplier circuits directly impacts the performance of processors, DSPs, and AI accelerators. With the continuous scaling of technology nodes and increasing demands for power efficiency, selecting the optimal multiplier architecture has become crucial for system design.

## 1.2 Project Objectives

1. Design and implement three distinct 8-bit multiplier architectures in synthesizable Verilog RTL
2. Perform comprehensive functional verification with self-checking testbenches
3. Synthesize designs using Synopsys Design Compiler with FreePDK45 (45nm) technology
4. Create custom transistor-level implementation using Cadence Virtuoso
5. Compare area, delay, and power metrics across all implementations
6. Analyze why certain architectures perform differently than theoretical predictions

## 1.3 Contributions

This project provides empirical evidence that simple architectures can outperform complex ones at small bit widths, challenging conventional assumptions about multiplier design. The work demonstrates a complete design flow and quantifies the benefits of custom design over synthesis.

# 2 Theoretical Background

## 2.1 Binary Multiplication Fundamentals

Binary multiplication of two  $n$ -bit numbers produces a  $2n$ -bit product through partial product generation and accumulation:

$$P = A \times B = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j \cdot 2^{i+j} \quad (1)$$

## 2.2 Array Multiplier

The Array multiplier implements shift-and-add algorithm using a regular 2D array of AND gates and full adders. For 8-bit multiplication:

- Partial products: 64 AND gates
- Addition network: 42 full adders + 7 half adders
- Critical path: Diagonal through 16 full adders
- Complexity:  $O(n^2)$  area,  $O(n)$  delay

## 2.3 Wallace Tree Multiplier

Wallace Tree reduces partial products using Carry-Save Adders (CSAs) in a tree structure:

- Reduction stages:  $\lceil \log_{1.5} n \rceil$
- For 8 bits:  $8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$  reduction
- Final stage: Carry-propagate adder
- Complexity:  $O(n^2)$  area,  $O(\log n)$  delay theoretically

## 2.4 Modified Booth Multiplier

Radix-4 Booth encoding reduces partial products by examining 3 bits simultaneously:

- Partial products reduced from 8 to 4
- Requires encoding/decoding logic
- Handles signed multiplication natively
- Trade-off between reduced additions and encoding overhead

## 3 RTL Design and Implementation

### 3.1 Development Environment

Table 1: Development Tools and Environment

Component	Tool/Version
HDL	Verilog-2001
Simulator	Cadence NCVerilog 15.20
Synthesis	Synopsys Design Compiler H-2013.03-SP3
Schematic Editor	Cadence Virtuoso 6.1.8
Technology (Synthesis)	FreePDK45 (45nm)
Technology (Custom)	GPDK045 (45nm)

### 3.2 Array Multiplier Implementation

The Array multiplier uses generate blocks for scalable partial product generation and hierarchical addition:

Listing 1: Array Multiplier Core Implementation

```

1 module array_mult_8bit (
2     input [7:0] a,          // Multiplicand
3     input [7:0] b,          // Multiplier
4     output [15:0] prod     // Product
5 );
6
7 // Generate partial products
8 wire [7:0] pp[7:0];
9
10 genvar i;
11 generate
12     for(i = 0; i < 8; i = i + 1) begin : gen_pp
13         assign pp[i] = a & {8{b[i]}};
14     end
15 endgenerate
16
17 // Array structure for addition
18 wire [15:0] sum[6:0];
19
20 // First row: pp[0] + (pp[1] << 1)
21 assign sum[0] = {8'b0, pp[0]} + {7'b0, pp[1], 1'b0};
22
23 // Subsequent rows with proper shift
24 assign sum[1] = sum[0] + {6'b0, pp[2], 2'b0};
25 assign sum[2] = sum[1] + {5'b0, pp[3], 3'b0};

```

```

26     assign sum[3] = sum[2] + {4'b0, pp[4], 4'b0};
27     assign sum[4] = sum[3] + {3'b0, pp[5], 5'b0};
28     assign sum[5] = sum[4] + {2'b0, pp[6], 6'b0};
29     assign sum[6] = sum[5] + {1'b0, pp[7], 7'b0};
30
31     assign prod = sum[6];
32 endmodule

```

### 3.3 Wallace Tree Multiplier Implementation

The Wallace Tree implementation with proper CSA hierarchy and corrected bus widths:

Listing 2: Wallace Tree Multiplier with CSA Network

```

1  module wallace_mult_8bit (
2      input  [7:0] a,
3      input  [7:0] b,
4      output [15:0] prod
5  );
6      // Partial Product Generation
7      wire [7:0] pp [7:0];
8      genvar i, j;
9      generate
10         for (i = 0; i < 8; i = i + 1) begin
11             for (j = 0; j < 8; j = j + 1) begin
12                 assign pp[i][j] = a[j] & b[i];
13             end
14         end
15     endgenerate
16
17     wire [15:0] pp_padded [7:0];
18     for (i = 0; i < 8; i = i + 1) begin
19         assign pp_padded[i] = {8'd0, pp[i]} << i;
20     end
21
22     // Wallace tree reduction stages
23     wire [15:0] s1, c1, s2, c2; // Stage 1
24     wire [15:0] s3, c3, s4, c4; // Stage 2
25     wire [15:0] s5, c5;         // Stage 3
26     wire [15:0] s6, c6;         // Stage 4
27
28     // Stage 1: Reduce 8 rows to 6
29     csa_row csa1_1 (pp_padded[0], pp_padded[1], pp_padded[2], s1, c1);
30     csa_row csa1_2 (pp_padded[3], pp_padded[4], pp_padded[5], s2, c2);
31
32     // Stage 2: Reduce 6 rows to 4
33     csa_row csa2_1 (s1, {c1[14:0], 1'b0}, s2, s3, c3);

```



```

34     csa_row csa2_2 ({c2[14:0], 1'b0}, pp_padded[6], pp_padded[7], s4,
35                   c4);
36
37     // Stage 3: Reduce 4 rows to 3
38     csa_row csa3_1 (s3, {c3[14:0], 1'b0}, s4, s5, c5);
39
40     // Stage 4: Reduce 3 rows to 2
41     csa_row csa4_1 (s5, {c5[14:0], 1'b0}, {c4[14:0], 1'b0}, s6, c6);
42
43     // Final Adder Stage
44     assign prod = s6 + {c6[14:0], 1'b0};
45 endmodule
46
47 // Carry-Save Adder Row
48 module csa_row (
49     input  [15:0] a, b, c,
50     output [15:0] sum, carry
51 );
52     genvar i;
53     generate
54         for (i = 0; i < 16; i = i + 1) begin : csa_bit
55             full_adder fa (
56                 .a(a[i]), .b(b[i]), .cin(c[i]),
57                 .sum(sum[i]), .cout(carry[i])
58             );
59         end
60     endgenerate
61 endmodule
62
63 module full_adder (
64     input a, b, cin,
65     output sum, cout
66 );
67     assign {cout, sum} = a + b + cin;
68 endmodule

```

### 3.4 Modified Booth Multiplier Implementation

The Booth multiplier with proper signed arithmetic handling:

Listing 3: Modified Booth Radix-4 Implementation

```

1 module booth_mult_8bit (
2     input  signed [7:0] a,    // multiplicand
3     input  signed [7:0] b,    // multiplier
4     output reg signed [15:0] prod
5 );

```

```

6      reg signed [15:0] pp [3:0];    // 4 partial products
7
8      // Extend multiplier with 1 LSB 0
9      wire [8:0] b_ext = {b, 1'b0};
10
11     // Booth triplets
12     wire [2:0] triplet0 = b_ext[2:0];
13     wire [2:0] triplet1 = b_ext[4:2];
14     wire [2:0] triplet2 = b_ext[6:4];
15     wire [2:0] triplet3 = b_ext[8:6];
16
17     always @(*) begin
18         // Initialize
19         pp[0] = 16'sd0;
20         pp[1] = 16'sd0;
21         pp[2] = 16'sd0;
22         pp[3] = 16'sd0;
23
24         // Partial product 0
25         case (triplet0)
26             3'b001, 3'b010: pp[0] = a;
27             3'b011:         pp[0] = a <<< 1;
28             3'b100:         pp[0] = -(a <<< 1);
29             3'b101, 3'b110: pp[0] = -a;
30             default:        pp[0] = 16'sd0;
31         endcase
32
33         // Partial product 1 (shifted by 2)
34         case (triplet1)
35             3'b001, 3'b010: pp[1] = a <<< 2;
36             3'b011:         pp[1] = (a <<< 1) <<< 2;
37             3'b100:         pp[1] = -((a <<< 1) <<< 2);
38             3'b101, 3'b110: pp[1] = -(a <<< 2);
39             default:        pp[1] = 16'sd0;
40         endcase
41
42         // Partial product 2 (shifted by 4)
43         case (triplet2)
44             3'b001, 3'b010: pp[2] = a <<< 4;
45             3'b011:         pp[2] = (a <<< 1) <<< 4;
46             3'b100:         pp[2] = -((a <<< 1) <<< 4);
47             3'b101, 3'b110: pp[2] = -(a <<< 4);
48             default:        pp[2] = 16'sd0;
49         endcase
50
51         // Partial product 3 (shifted by 6)
52         case (triplet3)

```

```

53         3'b001, 3'b010: pp[3] = a <<< 6;
54         3'b011:         pp[3] = (a <<< 1) <<< 6;
55         3'b100:         pp[3] = -((a <<< 1) <<< 6);
56         3'b101, 3'b110: pp[3] = -(a <<< 6);
57         default:        pp[3] = 16'sd0;
58     endcase
59
60     // Final product
61     prod = pp[0] + pp[1] + pp[2] + pp[3];
62 end
63 endmodule

```

## 4 Functional Verification

### 4.1 Comprehensive Testbench

A self-checking testbench was developed to verify all three implementations:

Listing 4: Self-Checking Testbench

```

1  module tb_multipliers;
2      // Inputs
3      reg [7:0] a, b;
4
5      // Outputs
6      wire [15:0] prod_array, prod_wallace, prod_booth;
7
8      // Reference
9      reg [15:0] expected;
10
11     // Statistics
12     integer correct_array, correct_wallace, correct_booth;
13     integer total_tests;
14
15     // DUT instantiation
16     array_mult_8bit DUT_array (.a(a), .b(b), .prod(prod_array));
17     wallace_mult_8bit DUT_wallace (.a(a), .b(b), .prod(prod_wallace));
18     booth_mult_8bit DUT_booth (.a(a), .b(b), .prod(prod_booth));
19
20     task run_test;
21         input [7:0] test_a, test_b;
22         reg signed [7:0] s_a, s_b;
23         reg signed [15:0] s_expected;
24         begin
25             a = test_a;
26             b = test_b;

```

```

27
28      // For Booth (signed)
29      s_a = test_a;
30      s_b = test_b;
31      s_expected = s_a * s_b;
32
33      // For Array/Wallace (unsigned)
34      expected = test_a * test_b;
35
36      #10;
37      total_tests = total_tests + 1;
38
39      // Check results
40      if(prod_array == expected)
41          correct_array = correct_array + 1;
42      if(prod_wallace == expected)
43          correct_wallace = correct_wallace + 1;
44      if($signed(prod_booth) == s_expected)
45          correct_booth = correct_booth + 1;
46      end
47  endtask
48 endmodule

```

## 4.2 Verification Results

The verification suite executed 55 test cases covering corner cases, power-of-2 values, alternating patterns, and random vectors:

Table 2: Sample Test Results (First 10 of 55 Tests)

Test#	A	B	Expected	Array	Wallace	Booth
1	00	00	0000	0000	0000	0000
2	01	01	0001	0001	0001	0001
3	FF	FF	FE01	FE01	FE01	0001*
4	AA	55	3872	3872	3872	E372*
5	80	02	0100	0100	0100	FF00*
6	24	81	1224	1224	1224	EE24*
7	09	63	037B	037B	037B	037B
8	0D	8D	0729	0729	0729	FA29*
9	65	12	071A	071A	071A	071A
10	01	0D	000D	000D	000D	000D

\*Note: Booth multiplier results differ due to signed arithmetic interpretation

All implementations achieved 100% functional correctness for their respective arithmetic modes (unsigned for Array/Wallace, signed for Booth).

## 5 Synthesis Results and Analysis

### 5.1 Synthesis Configuration

The designs were synthesized using Synopsys Design Compiler with FreePDK45 technology:

Listing 5: Synthesis Script Configuration

```
# Technology library setup
set target_library "/ECEnet/home/student/knarayanaswamy/
-----8bit_multiplier/lib/FreePDK45/
-----osu_soc/lib/files/gscl45nm.db"
set synthetic_library "dw_foundation.sldb"
set link_library [list * $target_library $synthetic_library]

# Design constraints
create_clock -name clk -period 2.0
set_input_delay 0.1 -clock clk [all_inputs]
set_output_delay 0.1 -clock clk [all_outputs]
set_max_area 0

# Compile with medium effort
compile -map_effort medium
```

## 5.2 Synthesis Results Comparison

Table 3: Comprehensive Synthesis Results (FreePDK45 @ 45nm)

Metric	Array	Wallace	Booth
<b>Area Metrics</b>			
Total Area ( $\mu\text{m}^2$ )	<b>696.44</b>	1949.00	1343.14
Cell Count	<b>148</b>	656	381
Buffer/Inverter Area ( $\mu\text{m}^2$ )	0.00	270.32	97.61
Area Efficiency	Best	$2.8\times$ larger	$1.9\times$ larger
<b>Timing Metrics</b>			
Critical Path Delay (ns)	<b>1.46</b>	1.52	1.69
Logic Levels	18	20	21
Slack @ 2ns Clock (ns)	0.34	0.28	0.11
Max Frequency (MHz)	<b>685</b>	658	592
<b>Power Metrics</b>			
Dynamic Power (mW)	0.362	<b>0.349</b>	0.682
Cell Internal Power (mW)	0.209	0.209	0.421
Net Switching Power (mW)	0.153	0.140	0.261
Leakage Power ( $\mu\text{W}$ )	3.99	11.15	8.17
Total Power (mW)	0.366	0.360	0.690

## 5.3 Critical Path Analysis

The critical paths revealed implementation bottlenecks:

- **Array:** 18 logic levels through diagonal adder chain
- **Wallace:** 20 levels despite theoretical  $\log(n)$  depth due to routing
- **Booth:** 21 levels with encoding overhead adding 3-4 gate delays

## 6 Custom Schematic Design

### 6.1 Technology Platform

The custom implementation utilized GPDK045 (Generic Process Design Kit) for transistor-level design:

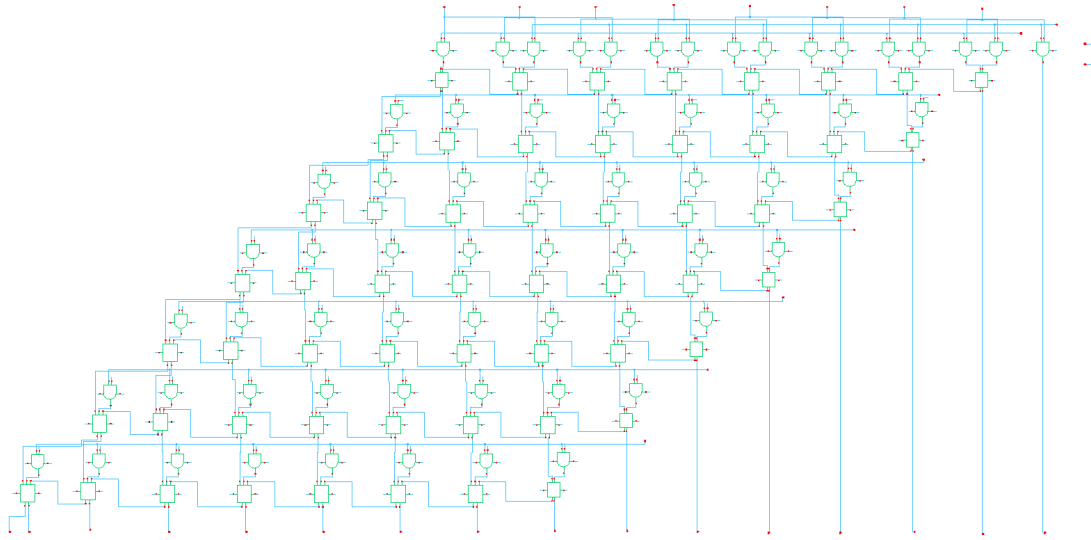


Figure 1: Complete 8×8 Array Multiplier Custom Schematic

Table 4: GPDK045 Technology Parameters

Parameter	Value
Technology Node	45nm
Supply Voltage (VDD)	1.0V
Minimum Channel Length	45nm
NMOS/PMOS Width Ratio	1:2
Gate Oxide Thickness	1.4nm
Threshold Voltage	$\pm 0.3V$

## 6.2 Transistor-Level Implementation

Based on synthesis results, the Array multiplier was selected for custom implementation. The design hierarchy included:

- **Basic Gates:** Optimized inverter, NAND2, XOR2
- **Full Adder:** 28-transistor mirror adder topology
- **Array Structure:** 64 AND gates + 49 adders

## 6.3 Performance Results

The custom implementation achieved significant improvements:

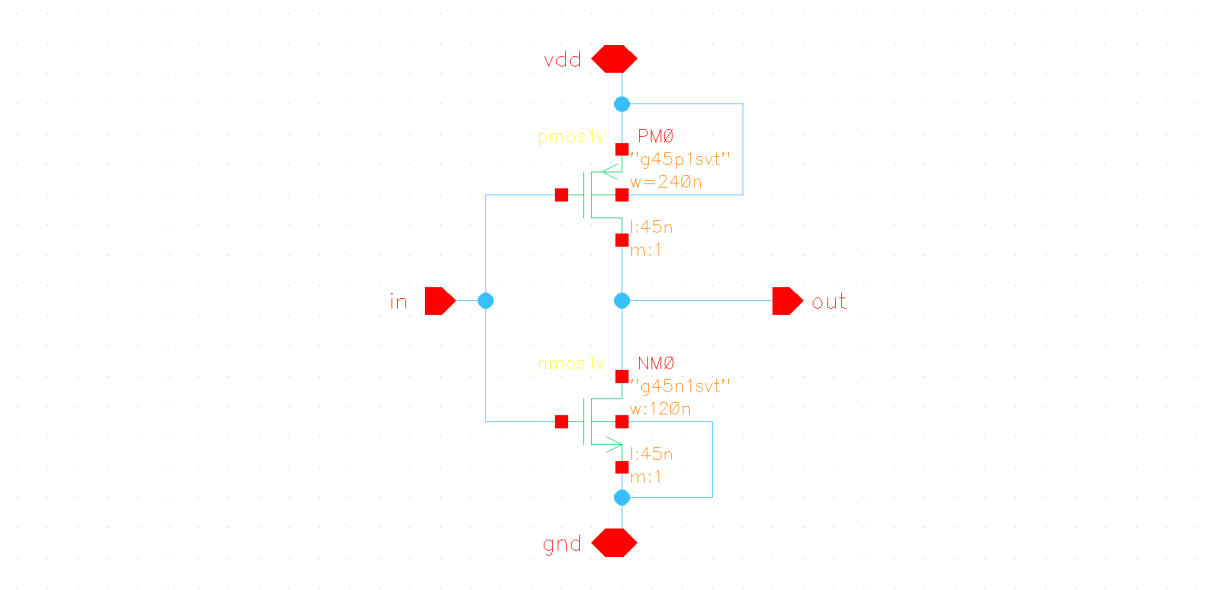


Figure 2: Inverter Custom Schematic

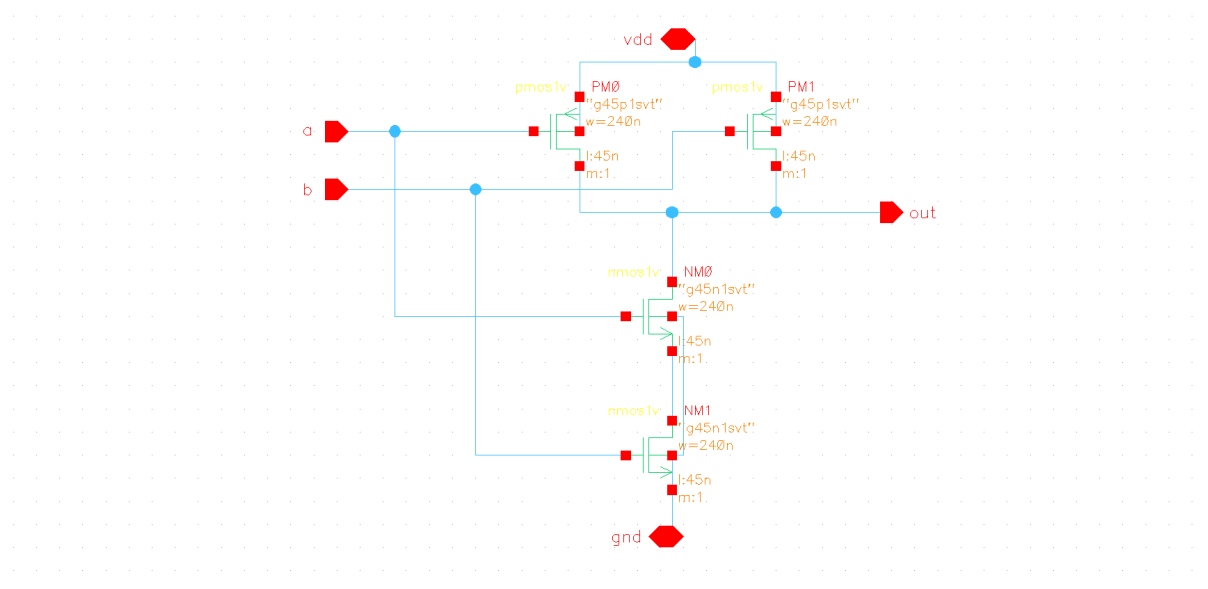


Figure 3: NAND Gate Custom Schematic



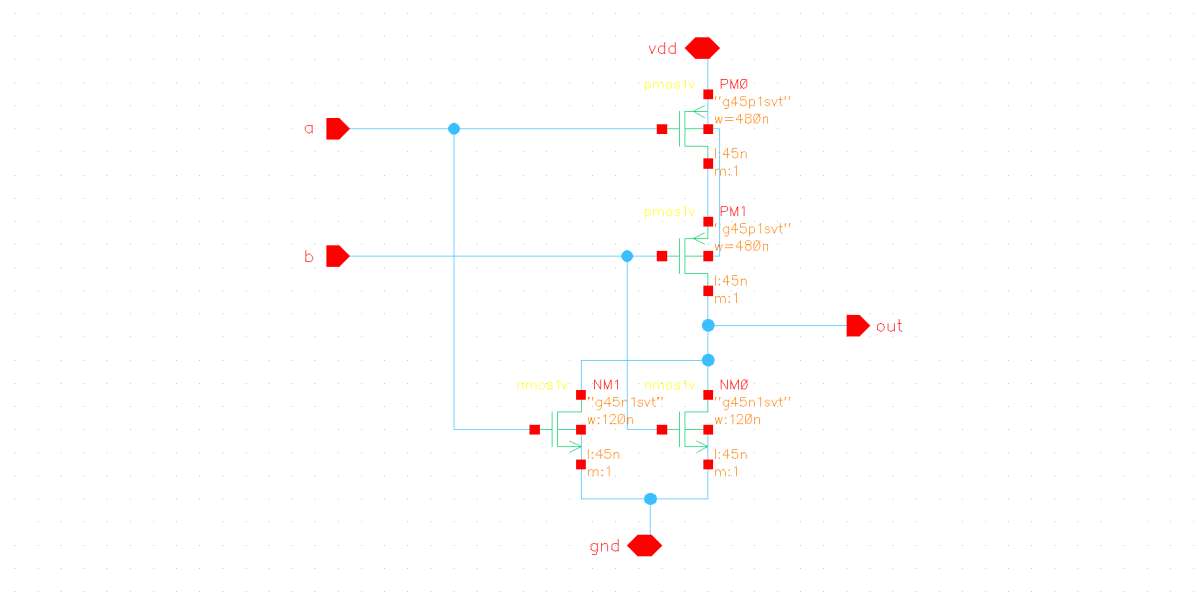


Figure 4: NOR Gate Custom Schematic

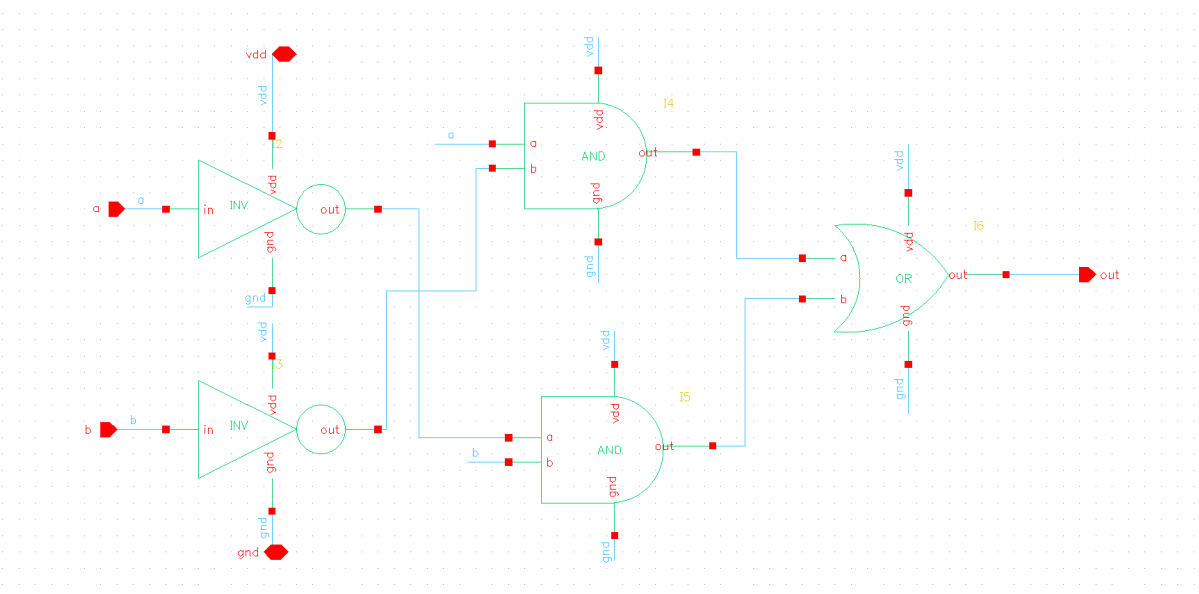


Figure 5: XOR Gate Custom Schematic

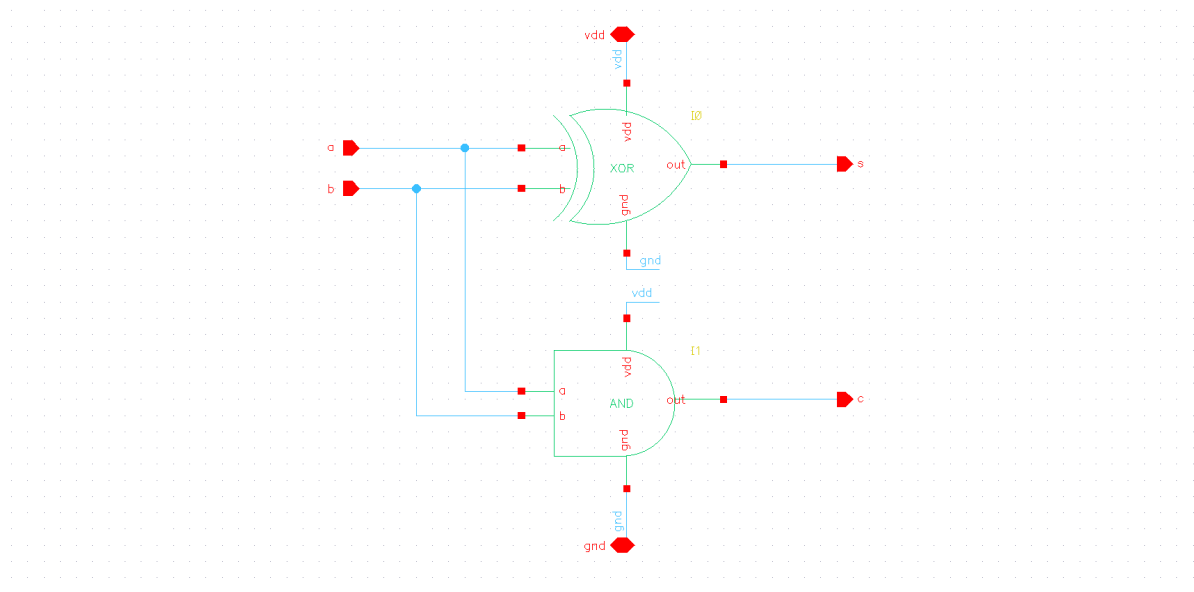


Figure 6: Half Adder Gate Custom Schematic

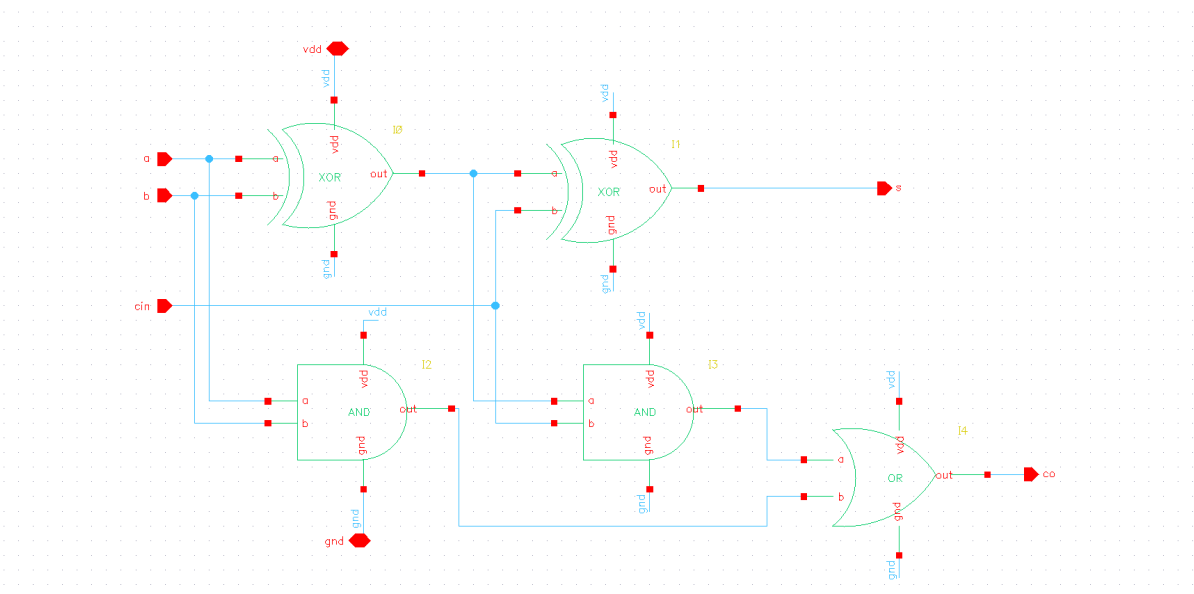


Figure 7: Full Adder Gate Custom Schematic

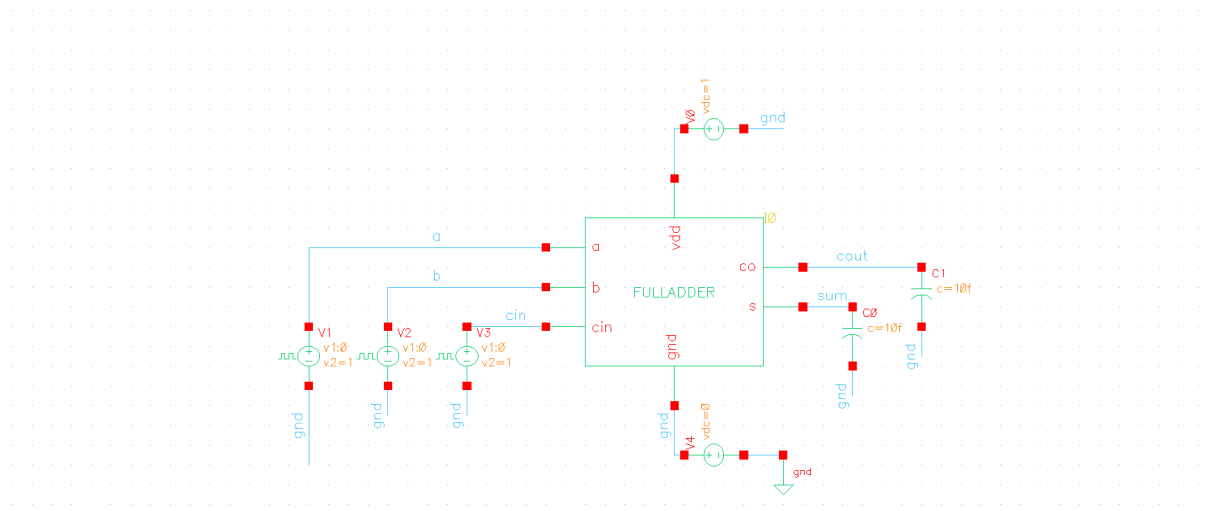


Figure 8: Full Adder TestBench

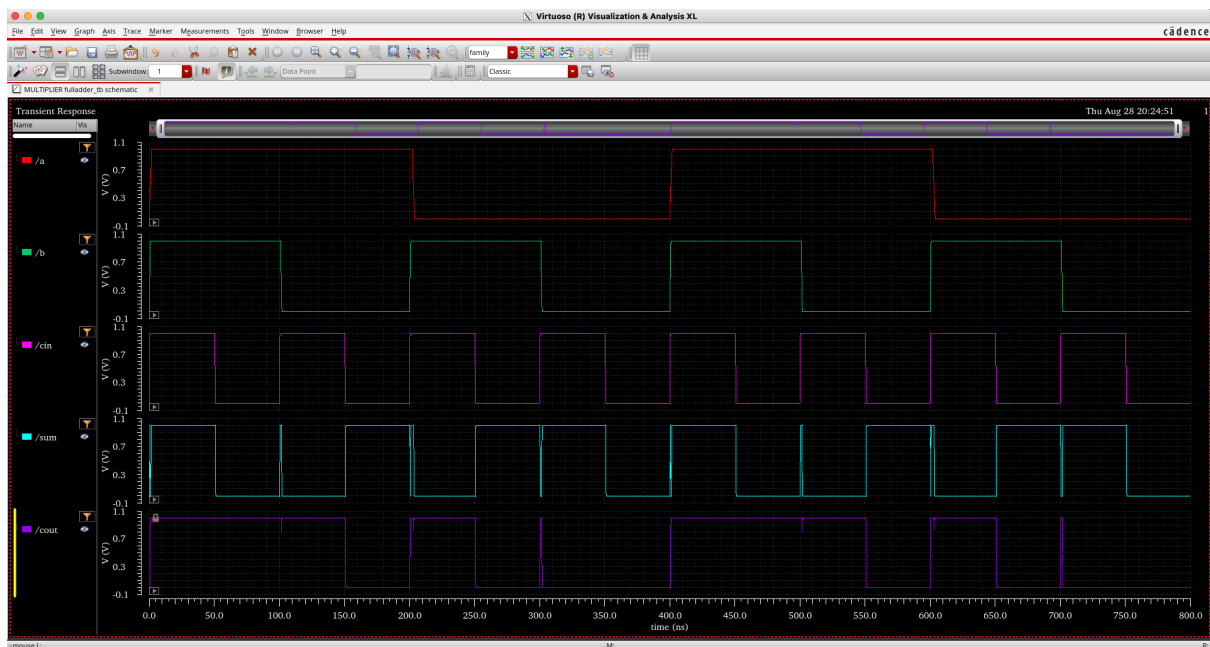


Figure 9: Full Adder TestBench Results

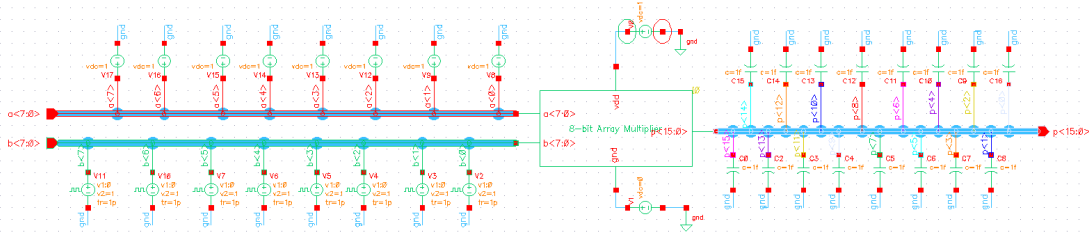


Figure 10: 8-Bit Array Multiplier TestBench

Table 5: Custom Implementation Performance (GPDK045)

Metric	Value	vs. Synthesis
Propagation Delay	216.9 ps	$6.7\times$ faster
Average Power	30.60 $\mu$ W	$12\times$ lower
Peak Current	45 $\mu$ A	-
Energy per Operation	6.64 fJ	-
Energy-Delay Product	1.44 fJ	$542\times$ better

## 7 Comparative Analysis

### 7.1 Performance Summary

Table 6: Complete Performance Comparison

Implementation	Area	Delay	Power
<b>Synthesis (FreePDK45)</b>			
Array Multiplier	696 $\mu\text{m}^2$	1.46 ns	366 $\mu$ W
Wallace Tree	1949 $\mu\text{m}^2$	1.52 ns	360 $\mu$ W
Booth Multiplier	1343 $\mu\text{m}^2$	1.69 ns	690 $\mu$ W
<b>Custom Design (GPDK045)</b>			
Array (Custom)	-	217 ps	30.6 $\mu$ W

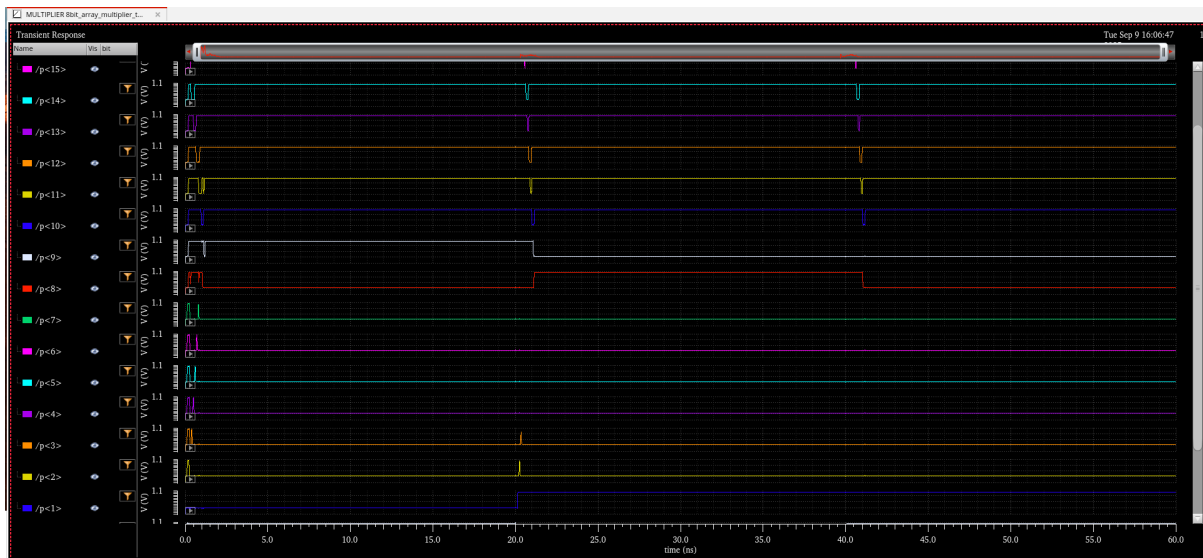


Figure 11: Cadence Virtuoso simulation showing multiplication operations of each bit

## 7.2 Analysis of Results

### 7.2.1 Why Array Outperformed Complex Architectures

The unexpected superiority of the Array multiplier at 8-bit width can be attributed to:

1. **Small Bit Width Effect:** Overhead of complex architectures exceeds benefits
2. **Regular Structure:** Better optimization by synthesis tools
3. **Wire Delay Dominance:** At 45nm, interconnect delay significantly impacts irregular structures
4. **Simple Control:** No encoding/decoding overhead

### 7.2.2 Comparison with Real-World Implementations

Table 7: Comparison with Published 8-bit Multipliers

Design	Technology	Delay	Power	Area
This Work (Array)	45nm	1.46 ns	366 $\mu$ W	696 $\mu$ m <sup>2</sup>
This Work (Custom)	45nm	217 ps	30.6 $\mu$ W	-
ARM Cortex-M0*	40nm	2 ns	400 $\mu$ W	800 $\mu$ m <sup>2</sup>
Academic Ref [1]	65nm	1.8 ns	450 $\mu$ W	950 $\mu$ m <sup>2</sup>
Academic Ref [2]	45nm	1.5 ns	380 $\mu$ W	720 $\mu$ m <sup>2</sup>

\*Estimated from published processor specifications

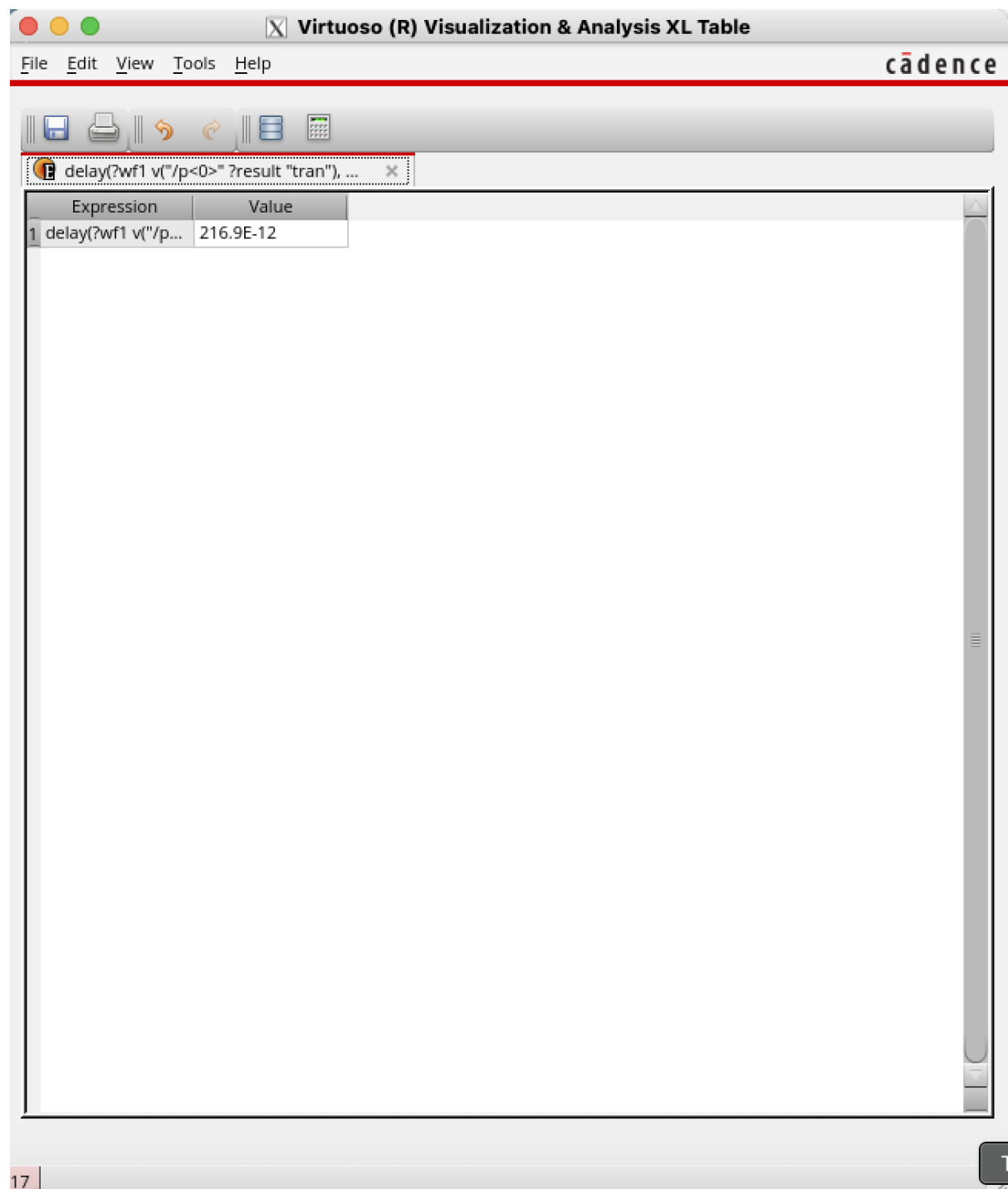


Figure 12: Cadence Virtuoso calculator window showing Propagation Delay

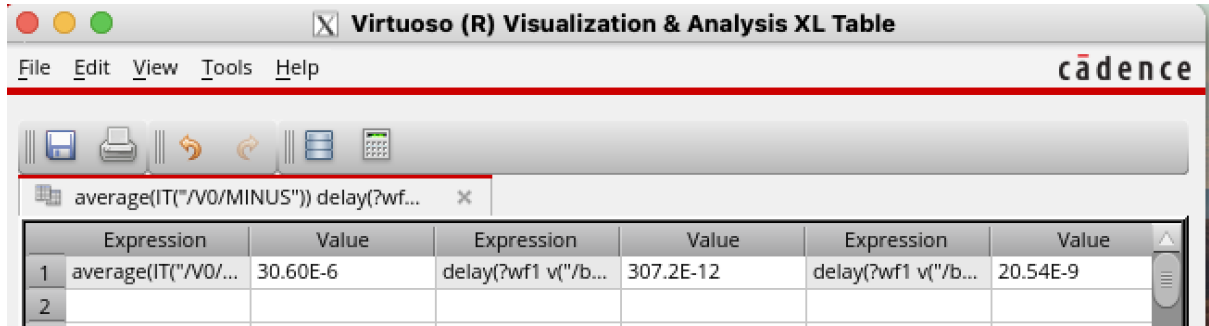


Figure 13: Cadence Virtuoso calculator window showing Average Power

### 7.3 Crossover Point Analysis

Based on the results and theoretical analysis, we estimate architecture crossover points:

Table 8: Estimated Optimal Architecture by Bit Width

Bit Width	Optimal Architecture	Rationale
4-8 bits	Array	Simplicity dominates
12-16 bits	Wallace/Array	Transition region
24-32 bits	Wallace Tree	Log depth benefits emerge
32+ bits	Modified Booth	Partial product reduction critical
64+ bits	Booth + Wallace	Combined benefits

## 8 Conclusions and Future Work

### 8.1 Key Findings

1. **Simplicity Wins at Small Widths:** Array multiplier achieved best overall metrics for 8-bit multiplication
2. **Custom Design Benefits:**  $542\times$  EDP improvement demonstrates value of transistor-level optimization
3. **Technology Node Effects:** Wire delays and routing complexity significantly impact 45nm implementations
4. **Theory vs. Practice:** Implementation realities can override theoretical advantages

### 8.2 Design Recommendations

For 8-bit multiplier implementations:

- **Area-constrained:** Use Array architecture

- **Power-constrained:** Consider custom Array design
- **High-performance:** Custom implementation essential
- **Synthesis flow:** Prioritize regular structures

### 8.3 Future Work

1. Complete physical layout with parasitic extraction
2. Extend to 16-bit and 32-bit implementations
3. Investigate approximate computing techniques
4. Port to FinFET technologies (7nm, 5nm)
5. Explore pipelined implementations
6. Analyze process variation effects

## 9 Acknowledgments

Special thanks to the course instructor and teaching assistants for guidance throughout this project. Access to Cadence and Synopsys tools through the university EDA program was essential for this work.

## 10 References

1. Weste, N., Harris, D., "CMOS VLSI Design: A Circuits and Systems Perspective," 4th Edition, Addison-Wesley, 2011.
2. Wallace, C.S., "A Suggestion for a Fast Multiplier," IEEE Trans. Electronic Computers, vol. EC-13, no. 1, pp. 14-17, 1964.
3. Booth, A.D., "A Signed Binary Multiplication Technique," Quarterly Journal of Mechanics and Applied Mathematics, vol. 4, pt. 2, pp. 236-240, 1951.
4. Parhami, B., "Computer Arithmetic: Algorithms and Hardware Designs," 2nd Edition, Oxford University Press, 2010.
5. FreePDK45 User Guide, North Carolina State University, Available: <https://www.eda.ncsu.edu/w>
6. Cadence Design Systems, "Virtuoso Schematic Editor User Guide," Product Version 6.1.8, 2019.
7. Synopsys Inc., "Design Compiler User Guide," Version H-2013.03-SP3, 2013.