ECE 310
Project 1 Report: 8-bit Wallace Multiplier

Ohm Patel

September 18, 2025

Abstract

This project implemented a gate-level 8×8 unsigned Wallace multiplier using Verilog. The objective was to apply structural modeling techniques to realize a fast parallel multiplier architecture composed of AND gates, Half Adders (HA), and Full Adders (FA). The Wallace tree approach reduces critical path delay by performing parallel compression of partial products, unlike ripple-based or shift-and-add multipliers. The design was verified through simulation using multiple representative input vectors, including trivial, edge, and non-trivial cases. All test results matched expected outputs. A detailed analysis of the design, gate counts, and trade-offs is presented, making this report a comprehensive documentation of the implementation.

1 Introduction and Background

Multiplication is a key arithmetic operation in digital systems such as processors. An implementation such as array or ripple-based multiplication suffers from long critical path delays because carries propagate through multiple stages.

The Wallace multiplier addresses this by reducing the partial product matrix using a tree of adders. At each level, sets of three bits in the same column are reduced to two bits using Full Adders, while sets of two bits are reduced with Half Adders. This continues until only two rows remain, which are then summed by a final ripple carry adder.

In this project, an 8×8 unsigned Wallace multiplier was designed, implemented at the gate level in Verilog, and tested through simulation.

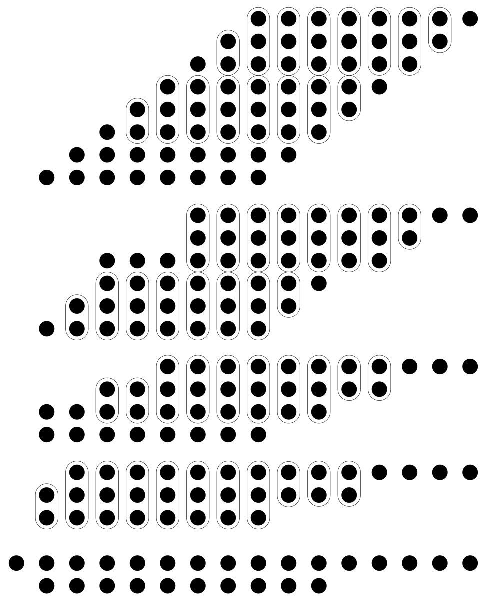

Figure 1: Wallace tree structure for partial product reduction.

2 Design and Implementation

2.1 Half Adder

The Half Adder generates the sum and carry of two inputs:

$$S = A \oplus B, \quad C_{out} = A \cdot B$$

Listing 1: Gate-level Half Adder

```
module half_adder(
    input  A, B,
    output S, Cout
);
    xor(S, A, B);
    and(Cout, A, B);
endmodule
```

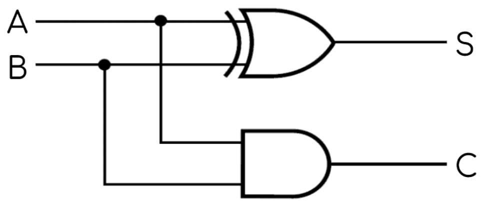

Figure 2: Gate-level diagram of a Half Adder.

2.2 Full Adder

The Full Adder extends the Half Adder with a carry-in:

$$S = A \oplus B \oplus C_{in}, \quad C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

Listing 2: Gate-level Full Adder

```
module full_adder(
    input  A, B, Cin,
    output S, Cout
);
    wire w1, w2, w3;
    xor(w1, A, B);
    xor(S, w1, Cin);
    and(w2, A, B);
    and(w3, w1, Cin);
    or(Cout, w2, w3);
endmodule
```

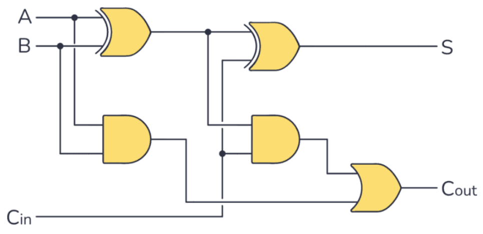

Figure 3: Gate-level diagram of a Full Adder.

2.3 Ripple-Carry Adder (RCA)

A Ripple-Carry Adder connects multiple Full Adders in series, with the carry-out of each stage feeding the carry-in of the next. While simple to design, its delay increases linearly with the number of bits due to carry propagation.

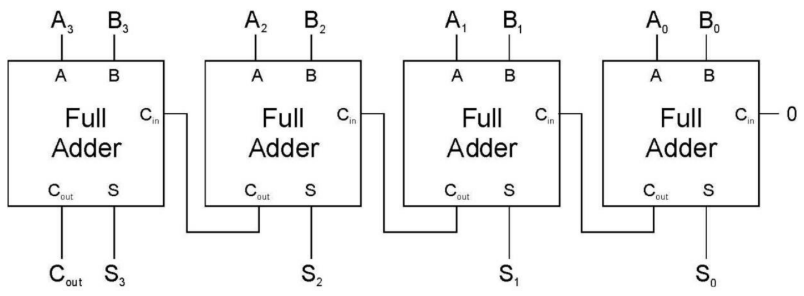

Figure 4: Structural diagram of a 4-bit Ripple-Carry Adder.

2.4 Wallace Multiplier

The 8×8 Wallace multiplier was implemented structurally in Verilog using 64 partial products generated by AND gates. These were reduced in multiple layers with Half Adders (HA) and Full Adders (FA), leaving two final rows that were summed using a Ripple-Carry Adder (RCA). The final product bits were connected to the output bus using `buf` primitives to comply with gate-level modeling requirements.

A high-level Verilog module definition is shown in Listing 3.

Listing 3: Top-level 8×8 Wallace multiplier

```
module wallace_8x8(
    input [7:0] a, b,
    output [15:0] prod
);
    // Partial product generation and Wallace reduction
    // Implementation includes ~64 ANDs, 47 FAs, and 17 HAs
    ...
endmodule
```

The last stage mapped the outputs of the reduction layers into the 16-bit product bus. An excerpt is shown in Listing 4.

Listing 4: Excerpt showing final product mapping

```
    buf (prod[0],  p[0][0]);   // column 0
    buf (prod[1],  hs[0]);     // from ha0
    buf (prod[2],  hs[4]);     // from ha4
    ...
    buf (prod[14], fs[46]);    // from fa46
    buf (prod[15], hs[16]);    // final HA (MSB)
```

2.5 Gate Count Analysis

A count of resources used is summarized in Table 1.

Table 1: Gate count summary for the 8×8 Wallace multiplier.

Component	Quantity	Notes
AND gates (partial products)	64	8×8 matrix
Full Adders (FA)	47	Used across Wallace levels and RCA
Half Adders (HA)	17	Used where only 2 inputs remain
Other gates (inside FA/HA)	Implicit	XOR, AND, OR expanded

This gate count demonstrates the trade-off of Wallace multipliers: increased hardware compared to ripple-adders, but lower delay.

2.6 Design Diagram

The 8×8 Wallace multiplier was first constructed by hand as shown in Figure 5. Each highlighter in the diagram represents either a Half Adder (HA) or a Full Adder (FA) yellow or blue respectively. The partial products in the code are defined by ($p_{i,j} = a_i \cdot b_j$) generated by AND gates. The adders are arranged in successive reduction layers ("slices") that compress the partial product matrix until only two rows remain. These two rows are then summed together to form the final product.

This diagram illustrates the hierarchical structure of the Wallace tree, highlighting the tradeoff between hardware cost (greater number of adders and wires) and reduced propagation delay compared to a ripple-based multiplier. The design ensures that carry signals are managed locally within each slice, limiting the depth of the critical path.

2.7 Testbench

The testbench applied multiple vectors to validate correctness. Representative cases include:

- $0 \times 0 = 0$: Baseline check.
- $0 \times 255 = 0$: Ensures zeroing behavior.
- $1 \times 255 = 255$: Identity property.
- $128 \times 2 = 256$: Power-of-two shifting.
- $15 \times 15 = 225$: Small operand multiplication.
- $25 \times 12 = 300$: Uneven mid-sized values.
- $200 \times 13 = 2600$: Large–small mix.
- $150 \times 151 = 22650$: Arbitrary large operands.
- $255 \times 255 = 65025$: Maximum edge case.

Listing 5: Testbench excerpt for Wallace multiplier

```
initial begin
    a = 8'b00000000; b = 8'b00000000; #10;
    a = 8'b00000000; b = 8'b11111111; #10;
    a = 8'b00000001; b = 8'b11111111; #10;
    a = 8'b10000000; b = 8'b00000010; #10;
    a = 8'b00001111; b = 8'b00001111; #10;
    a = 8'b00011001; b = 8'b00001100; #10;
    a = 8'b11001000; b = 8'b00001101; #10;
    a = 8'b10010110; b = 8'b10010111; #10;
    a = 8'b11111111; b = 8'b11111111; #10;
    $finish;
end
```

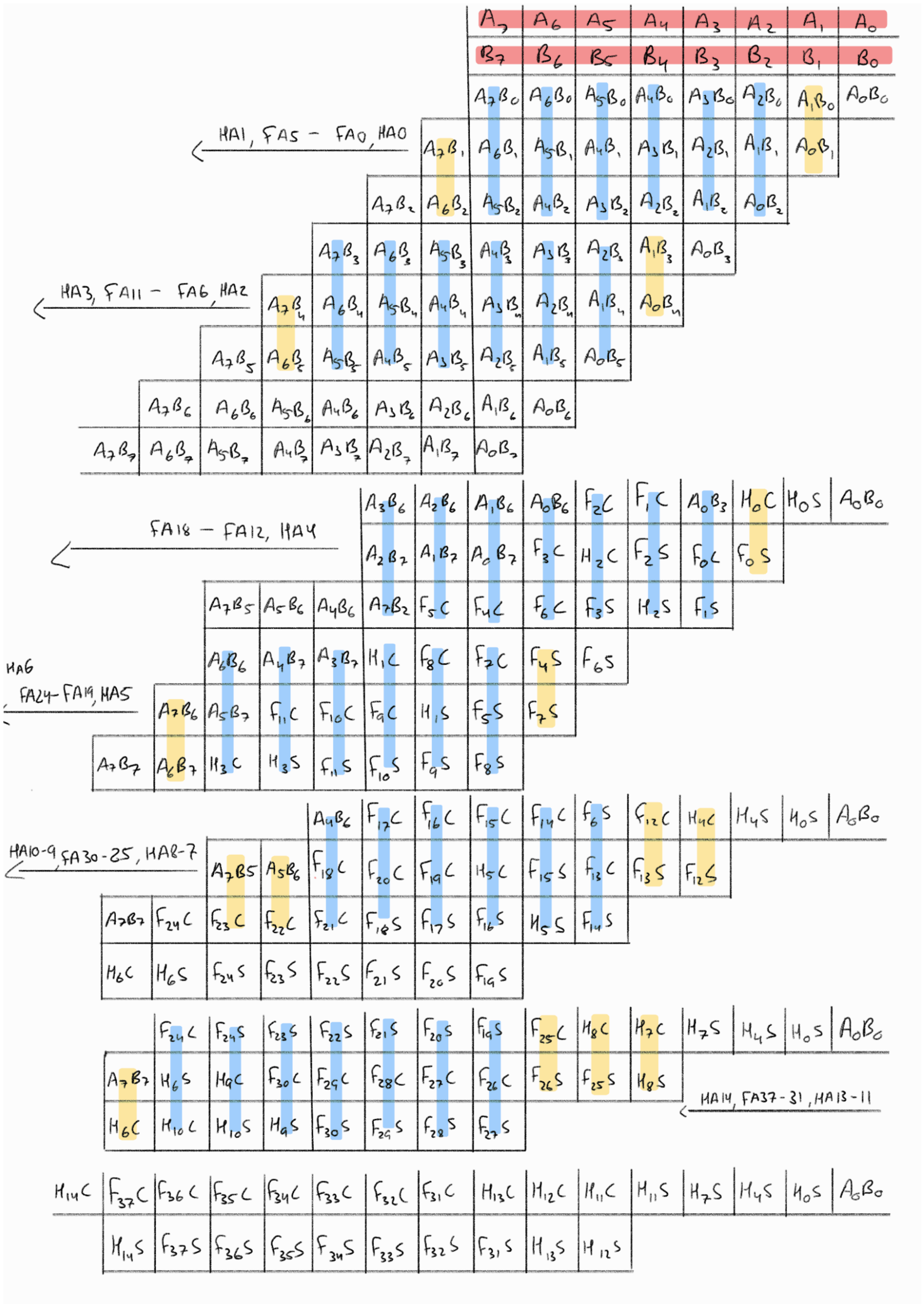

Figure 5: Hand-drawn diagram of the 8×8 Wallace multiplier showing partial products and Wallace reduction stages.

3 Results and Analysis

The simulation confirmed correct operation for all test vectors:

- 00000000 × 00000000 = 0000000000000000
- 00000001 × 11111111 = 0000000011111111
- 10000000 × 00000010 = 0000000100000000
- 00001111 × 00001111 = 0000000011100001
- 00011001 × 00001100 = 0000000100101100
- 11001000 × 00001101 = 0000101000111000
- 10010110 × 10010111 = 0101100001011010
- 11111111 × 11111111 = 1111111000000001

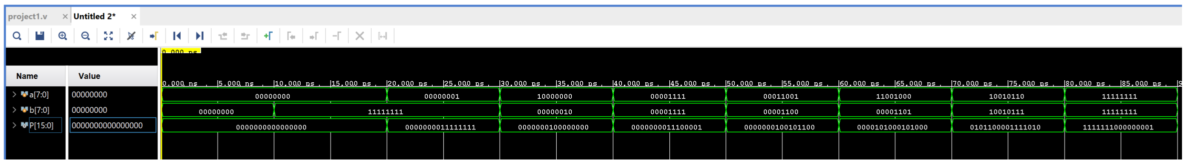

Figure 6: Simulation waveform of the 8×8 Wallace multiplier (Binary).

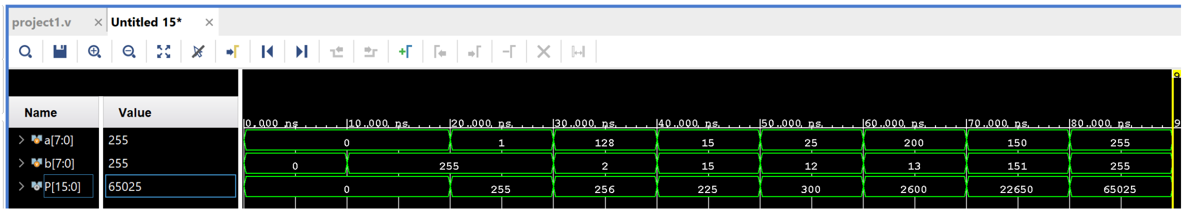

Figure 7: Simulation waveform of the 8×8 Wallace multiplier (Decimal).

All observed outputs matched the expected binary values, confirming that the Wallace multiplier functioned correctly. The results also demonstrated the reduced propagation depth compared to ripple-carry addition.

4 Conclusion

The 8×8 Wallace multiplier was successfully designed and implemented in Verilog at the gate level. Simulation results validated correctness across trivial, edge, and arbitrary test cases. The project demonstrated the Wallace tree's ability to reduce multiplication delay by parallel partial product reduction, at the cost of increased gate usage. This work reinforced hierarchical structural design principles, arithmetic circuit design, and the trade-offs between speed and hardware complexity.

References

[1] Wikipedia, "Wallace tree," *Wikipedia*, https://en.wikipedia.org/wiki/Wallace_tree, accessed September 2025.

This report was compiled using LaTeX .