COA Assignment

Course: Computer Organization and Architecture (COA)

Assignment Title: Verilog Implementation of 16-bit Multipliers

Instructor: Prof. Rajat Sadhukhan

Name: Ishaan Arora

Roll Number: 24114041

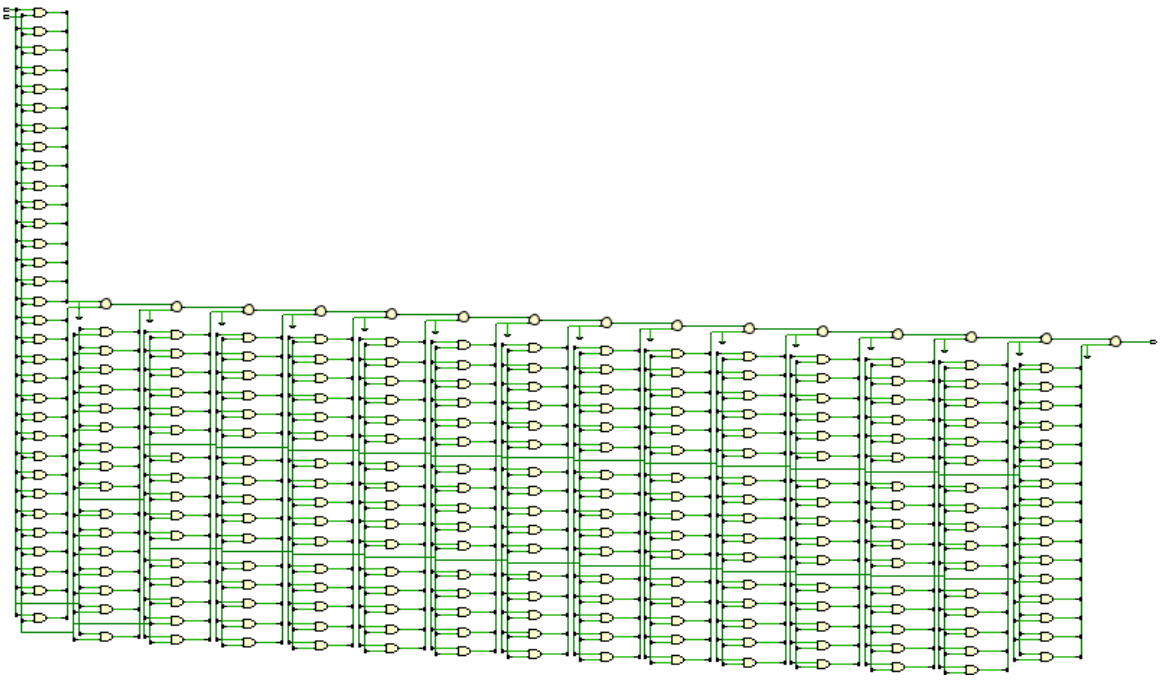
1. Objective

The aim of this project is to design and compare two types of 16-bit multipliers — an Array Multiplier and a Wallace-Tree Multiplier — using Verilog. The main goal is to understand how both architectures work, analyze their speed and resource usage, and see how certain optimizations improve performance.

2. Design Description

2.1 Array Multiplier

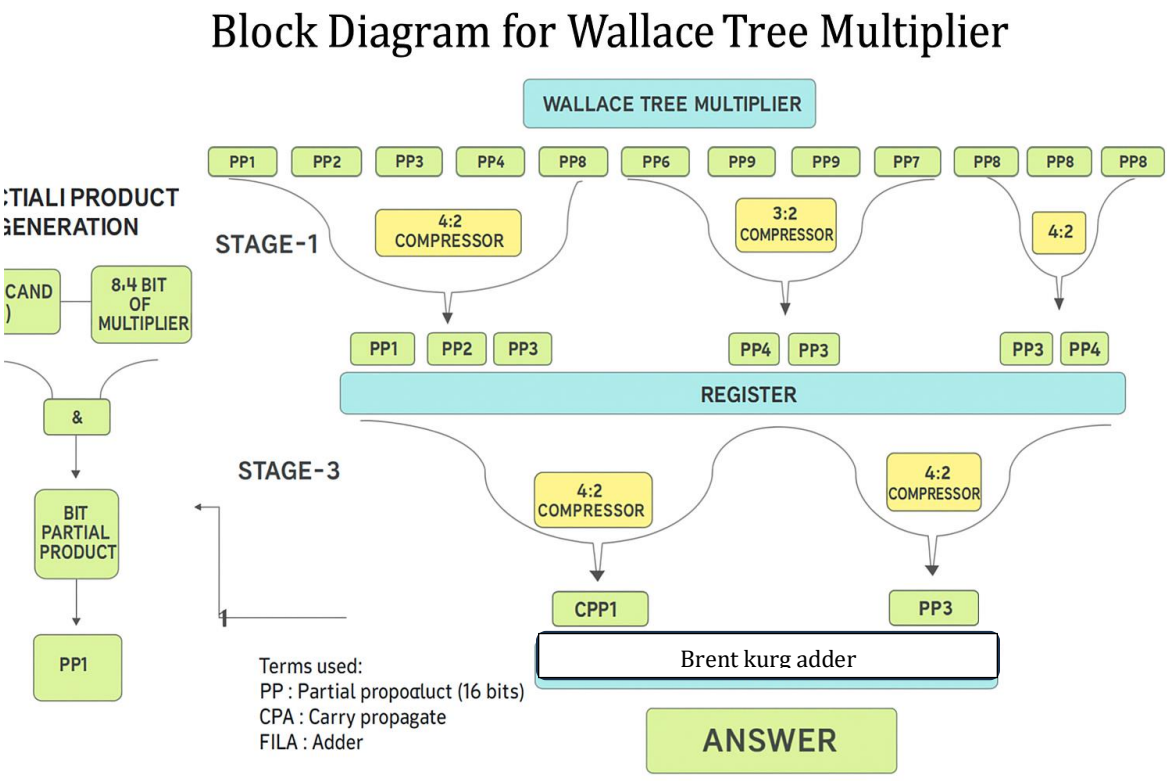
An array multiplier works just like normal hand multiplication but in binary form. Each bit of one number is multiplied with every bit of the other using AND gates. The results are then added together using a grid of adders. The design is simple but becomes slow for large bit sizes.

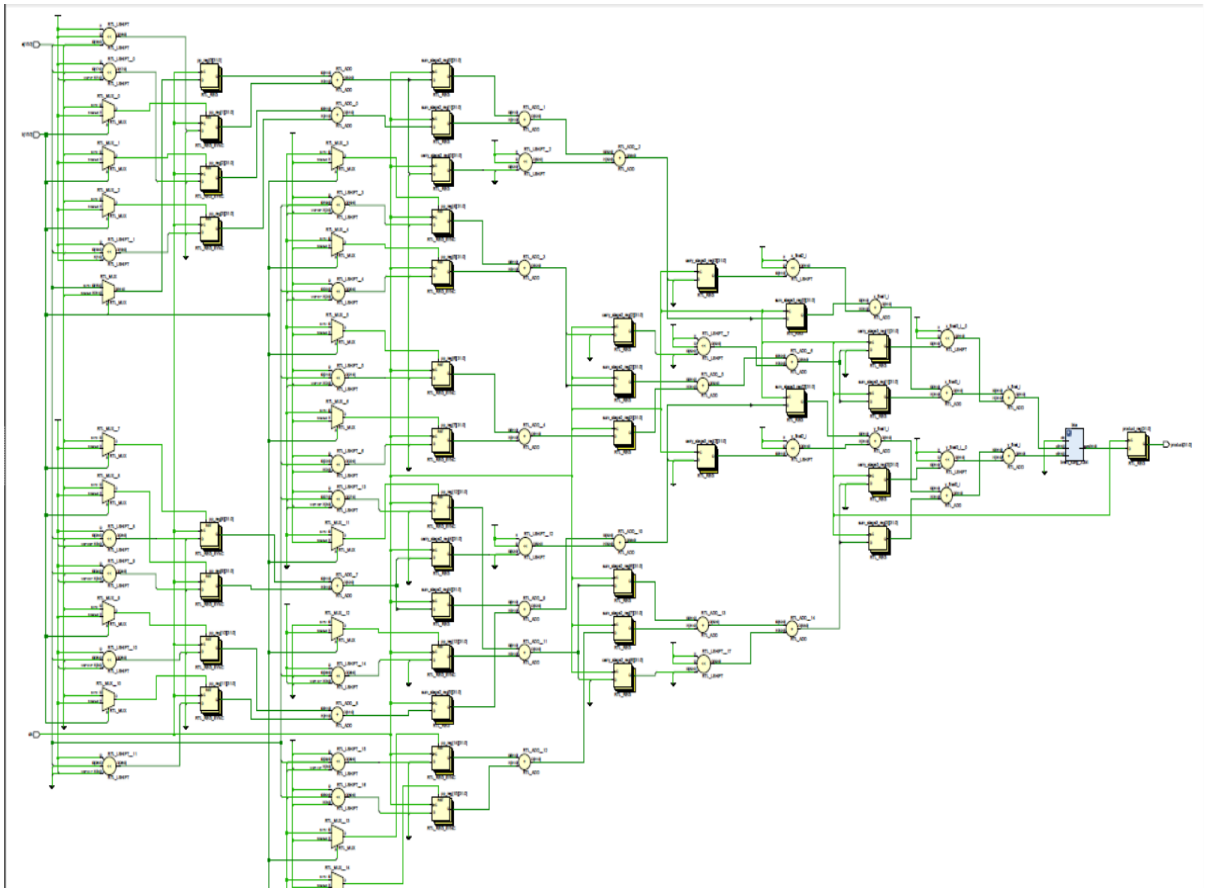


^Block diagram by Vivado

2.2 Wallace-Tree Multiplier

The Wallace-tree multiplier improves speed by reducing partial products in parallel using a structured compression tree. First, 16 partial products are generated by ANDing the multiplicand with each bit of the multiplier and extending them to 32 bits. Instead of adding them sequentially like an array multiplier, they are grouped by columns and compressed level-by-level. I used a combination of **3:2 compressors (carry-save adders)** and **4:2 compressors** to reduce the height of the partial product matrix efficiently. The reduction is done in **three pipelined stages**, which helps shorten the critical path and improves maximum frequency. After the tree produces only two final rows, the result is computed using a **Brent–Kung parallel prefix adder**, which provides fast and area-efficient final addition. This structure makes the design significantly faster than a simple array multiplier while keeping the hardware cost controlled





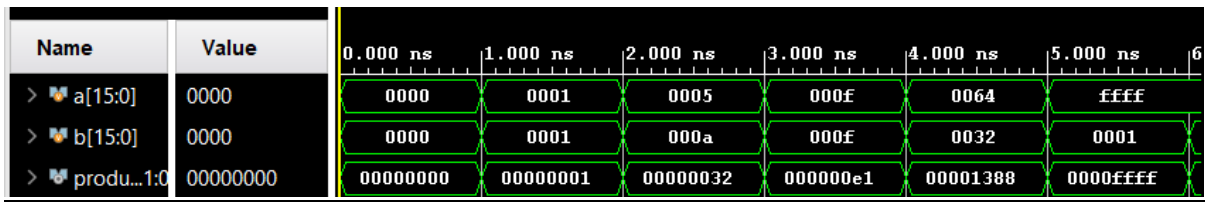
^Block Diagram by Vivado

3. Verilog Code

https://drive.google.com/drive/folders/1eo2zsJF0VTPdKIk5T8XKAhAu-6nKxLcM?usp=drive_link

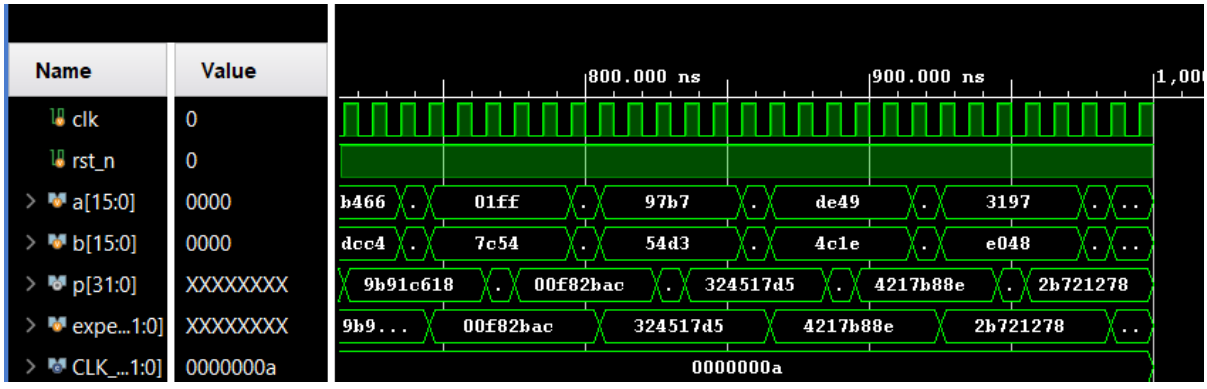
4. Simulation & Testbench

Array multiplier



```
================================================
       STARTING ARRAY MULTIPLIER TESTS
================================================
TEST PASSED | a=    0 b=    0 | product=         0
TEST PASSED | a=    1 b=    1 | product=         1
TEST PASSED | a=    5 b=   10 | product=        50
TEST PASSED | a=   15 b=   15 | product=       225
TEST PASSED | a=  100 b=   50 | product=      5000
TEST PASSED | a=65535 b=    1 | product=     65535
TEST PASSED | a=65535 b=65535 | product=4294836225
TEST PASSED | a=    0 b=43981 | product=         0
TEST PASSED | a=32768 b=    2 | product=     65536
TEST PASSED | a=13604 b=24193 | product= 329121572
TEST PASSED | a=54793 b=22115 | product=1211747195
TEST PASSED | a=31501 b=39309 | product=1238272809
TEST PASSED | a=33893 b=21010 | product= 712091930
TEST PASSED | a=58113 b=52493 | product=3050525709
TEST PASSED | a=61814 b=52541 | product=3247769374
TEST PASSED | a=22509 b=63372 | product=1426440348
```

Wallace Multiplier



Tcl Console × Messages Log

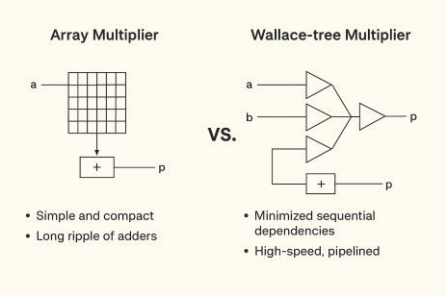
```
Running 20 randomized tests...
PASS: a= 1540, b=52858 â†' product=  81401320 (expected   81401320)
PASS: a=12546, b=45769 â†' product= 574217874 (expected  574217874)
PASS: a=39430, b=63859 â†' product=2517960370 (expected 2517960370)
PASS: a=35312, b=40970 â†' product=1446732640 (expected 1446732640)
PASS: a=51182, b=47859 â†' product=2449519338 (expected 2449519338)
PASS: a=21204, b=25076 â†' product= 531711504 (expected  531711504)
PASS: a=46182, b=56516 â†' product=2610021912 (expected 2610021912)
PASS: a=  511, b=31828 â†' product=  16264108 (expected   16264108)
PASS: a=38839, b=21715 â†' product= 843388885 (expected  843388885)
PASS: a=56905, b=19486 â†' product=1108850830 (expected 1108850830)
PASS: a=12695, b=57416 â†' product= 728896120 (expected  728896120)
```

5. Synthesis Results

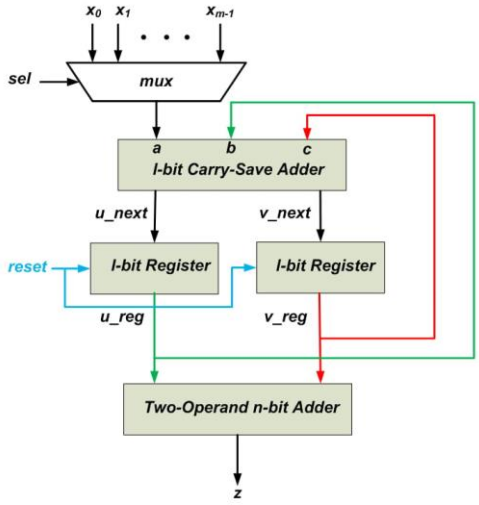
Multiplier	LUTs	FFs	DSPs	Max Freq (MHz)	Critical Path Delay (ns)
Array	274	0	0	89.60	11.158ns
Wallace-Tree	321	514	0	252.52	3.96

6. Analysis & Discussion

The performance comparison clearly highlights the fundamental trade-off between a conventional array multiplier and the optimized Wallace-tree architecture. The baseline array design is simple and compact, but its critical path is dominated by a long ripple of adders that sequentially accumulate partial products. This results in significant carry-propagation delay and restricts the maximum achievable clock frequency. In contrast, the Wallace-tree implementation restructures the computation to minimize sequential dependencies, enabling the design to run at a substantially higher speed while using slightly more hardware resources.



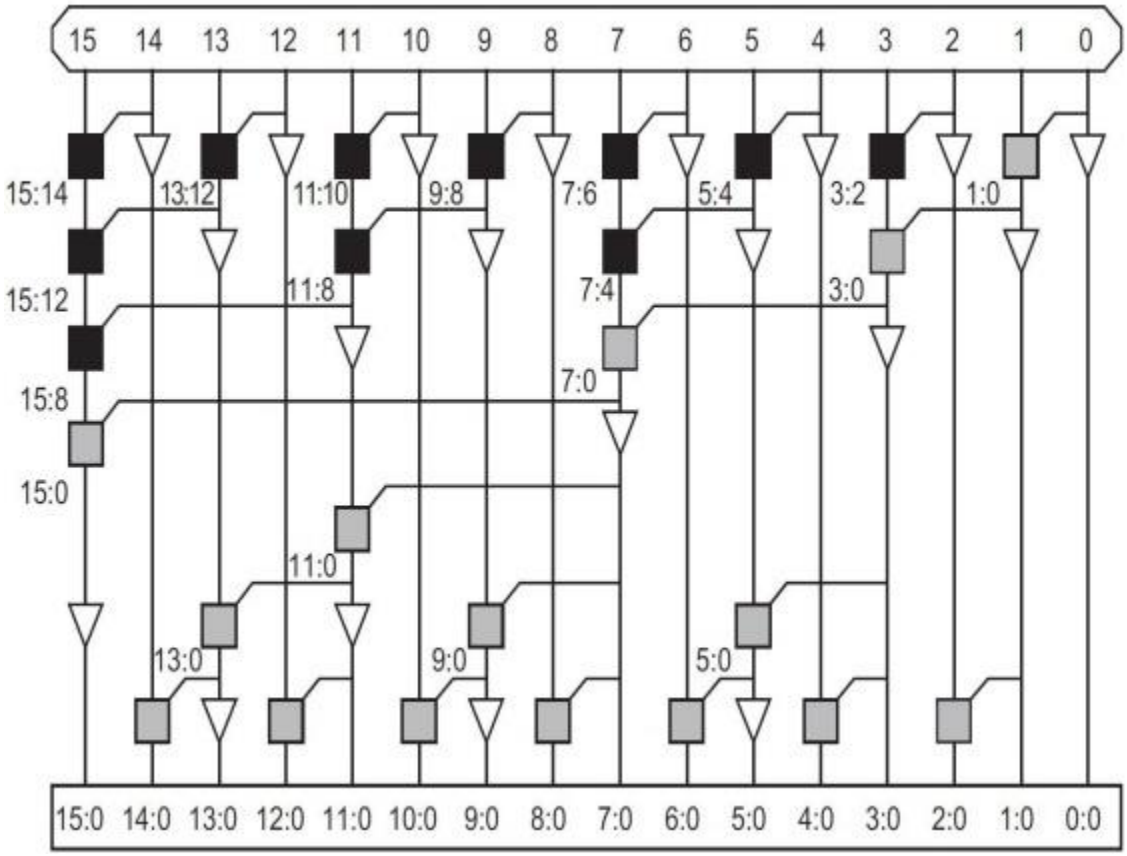
The first optimization appears in the partial-product generation stage. Instead of forming a full AND matrix, each partial product is generated using conditional shifts of the multiplicand (a << i), which synthesizes into efficient re-wiring rather than large LUT-based logic. This reduces both area and combinational depth and provides clean, aligned 32-bit vectors ready for reduction. The core performance improvement arises from the introduction of carry-save arithmetic using 3:2 and 4:2 style compressors. Carry-save adders avoid immediate carry propagation and instead generate sum and carry vectors in constant time. By rapidly compressing the partial-product matrix into just two vectors, the architecture removes the long cascade of dependent additions that limits the array multiplier's speed.



^ Carry save adder

The design further improves timing through balanced multi-stage pipelining. Registers inserted between reduction levels divide the logic into shorter segments, significantly reducing the critical path between clock edges. This enables the FPGA's internal CARRY4 chains and fast-carry resources to be utilized effectively. Each pipeline stage carries only localized logic, so routing becomes more predictable and timing closure is easier.

Another major optimization in my Wallace multiplier design is the use of a **32-bit Brent–Kung parallel-prefix adder** for the final addition stage. Unlike a normal ripple-carry adder—which waits for each bit's carry to propagate from LSB to MSB—the Brent–Kung adder computes carries in a **logarithmic tree structure**. This reduces the carry-propagation delay from **O(32)** in ripple adder to **O($\log_2$32) ≈ 5 stages**, which is a huge timing improvement. Brent–Kung is slightly slower than fully parallel adders like Kogge-Stone, but it is much more area-efficient and avoids long, complicated wiring. This makes it a very good balance between speed and hardware cost.



^ Brentkurg optimization

Together, these optimizations—efficient partial-product formation, aggressive carry-save reduction, balanced pipelining, and a prefix-based final adder—explain the significantly improved clock speed and reduced critical path delay observed in synthesis. Although the Wallace architecture uses more flip-flops and has higher routing demand than the simple array version, the performance gain far outweighs the additional resource cost. Overall, the optimized implementation demonstrates a clear and measurable improvement in timing efficiency while maintaining functional accuracy.

7. Conclusion

Working on this project helped me understand not just how multipliers work, but *why* different architectures behave so differently in real hardware. The array multiplier turned out to be the simplest design—easy to write, easy to visualize, and uses fewer resources. But because it depends on long chains of ripple additions, its critical path becomes very slow. This makes it good only for basic, low-speed applications where area matters more than performance.

The Wallace multiplier, on the other hand, showed why it is preferred in high-speed digital systems. By reducing partial products in parallel, grouping them intelligently, and using a fast Brent–Kung adder at the end, the design drastically cuts down the delay. Even though it uses more LUTs and registers, the improvement in speed is huge and very noticeable in the synthesis results.

Overall, the key takeaway for me is the classic engineering trade-off:
Array = simpler and smaller, Wallace = faster and more optimized.

If the goal is just functionality, the array multiplier works fine. But if the design requires high frequency, better timing, or needs to fit into a real CPU/DSP pipeline, the Wallace-Tree multiplier is definitely the better choice.

8. References

1. Wallace-Tree Multiplier – Wikipedia: https://en.wikipedia.org/wiki/Wallace_tree

2. Array Multiplier – GeeksforGeeks: https://www.geeksforgeeks.org/array-multiplier-in-digital-logic/

https://www.geeksforgeeks.org/array-multiplier-in-digital-logic/

3. Oklobdzija et al., IEEE TVLSI, 1995.