FPGA Implementation of Multiplier Topologies

Shreyas Ravishankar[†], Samarth Bonthala[†], Atharva Karaguppi[†], Tejas Bhagwat[†]
[†]The University of Texas at Austin
ECE 382N - High Speed Computer Arithmetic - Course Project Report

Abstract

Multipliers are one of the basic building blocks of a digital processor. This report describes the implementation of four multiplier topologies namely the Array Multiplier, Radix-2 Booth Multiplier, Dadda Multiplier and Wallace Multiplier in Verilog. These algorithms have been synthesized and implemented on Avnet Ultra-96v2 FPGA. The main purpose of this implementation was to compare the power, performance and area of each of these topologies by estimating the power, maximum clock frequency and resource utilization respectively through the Vivado implementation step. Additionally two of these multipliers were deployed on to the FPGA to check for functional correctness on actual FPGA hardware. Each of the topology has a 8-bit and a 16-bit multiplier implementation in this project. The speed of the Wallace and Dadda multiplier was observed to be approximately the same and the highest followed by Array Multiplier and Booth Multiplier. The resource utilization of Dadda multiplier turns out to be better than that of Array and Booth multiplier in the 16x16 variant.

Problem Statement

FPGAs are widely gaining popularity due to their versatility since they provide the user the ability to reconfigure the hardware to meet specific use case requirements post fabrication. When changes are required in the hardware, the configuration files can be modified to integrate the new functionality. Due to the benefit of reconfigurability, FPGAs are used to deploy processor designs, accelerator designs, etc. These designs use elaborate computations which require faster multiplier architectures. The performance of the multipliers has a significant impact on the throughput of the processing units. In this project, we aim to implement various types of popular multiplier architectures on an FPGA and compare them.

1 Background

1.1 Radix-2 Booth Multiplier

The Booth algorithm[1] provides a method for efficiently multiplying binary integers in signed 2's complement representation, requiring fewer additions and subtractions. It works on the principle that a string of 0s in the multiplier just needs to be shifted, not added, and that a string of 1s in the multiplier from bit weight 2^k to bit weight 2^m can be considered as 2^{k+1} to 2^m. Here k is the bit index of the partial product and m is the maximum times shift operation needs to be carried out. Note that m is also the number of bits in the multiplier. The booth algorithm necessitates examining the multiplier bits and shifting the partial product, just like all other multiplication techniques.

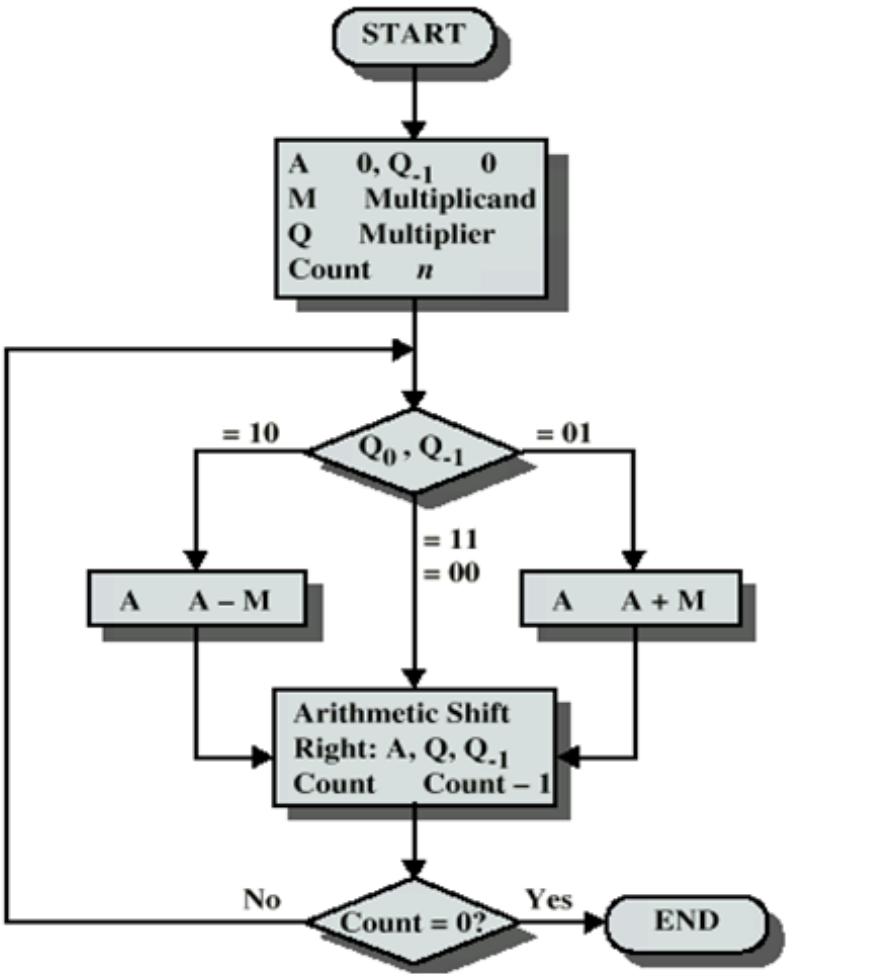

Fig 1: Flowchart for the Radix-2 Booth Multiplier[2]

1.2 Array Multiplier

Array multiplier [3] is used to multiply two binary values by using an array of full adders and half adders. The numerous partial products involved are added almost simultaneously using this array. Before the Adder array, an array of AND gates is utilized to create these partial products.

A series of add and shift micro-operations are needed to perform a sequential operation that involves checking the multiplier's bits one at a time and creating partial products. A combinational circuit that produces the product bits all at once can multiply two binary values in a single micro-operation. Since only the time required for propagation of signals from one gate cell to another is required, these multipliers are faster compared to Booth multipliers.

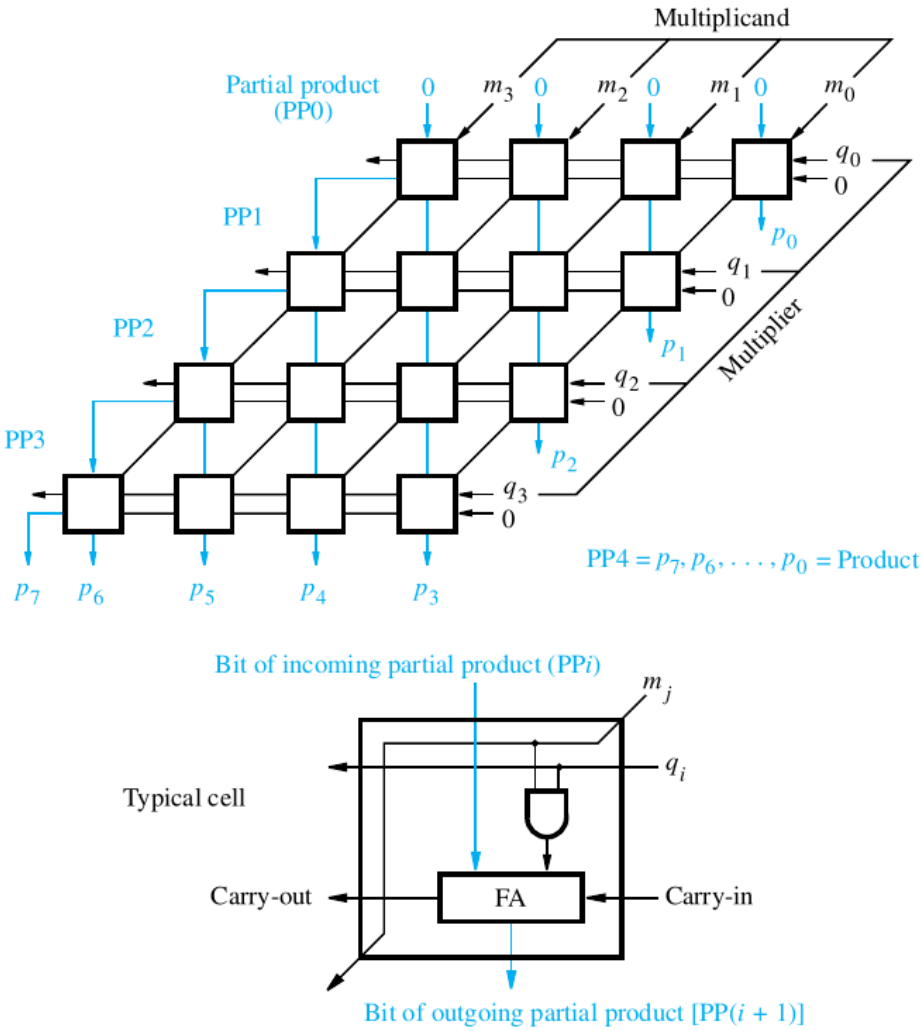

Fig 2: Array Multiplier [4]

1.3 Wallace Multiplier

Wallace multiplier [5] is one of the fast multiplier topologies that uses lesser resources than an array multiplier. First, the partial product array (of N^2 bits) is formed in the traditional Wallace tree multiplier. Three adjacent rows of partial products are grouped in the second stage. Full adders and half adders are used to reduce each set of three rows into two rows. Where there are three bits per column, full adders are used and where there are two bits per column, half adders are used. Any single bit in a column is not reduced and passed through to the next stage of reduction in the same column. This reduction process is repeated in each succeeding level until there are only two rows remaining. The final stage is employing a carry propagate adder to combine the final two rows. The dot diagram for 8x8 Wallace multiplier is shown in Fig 3.

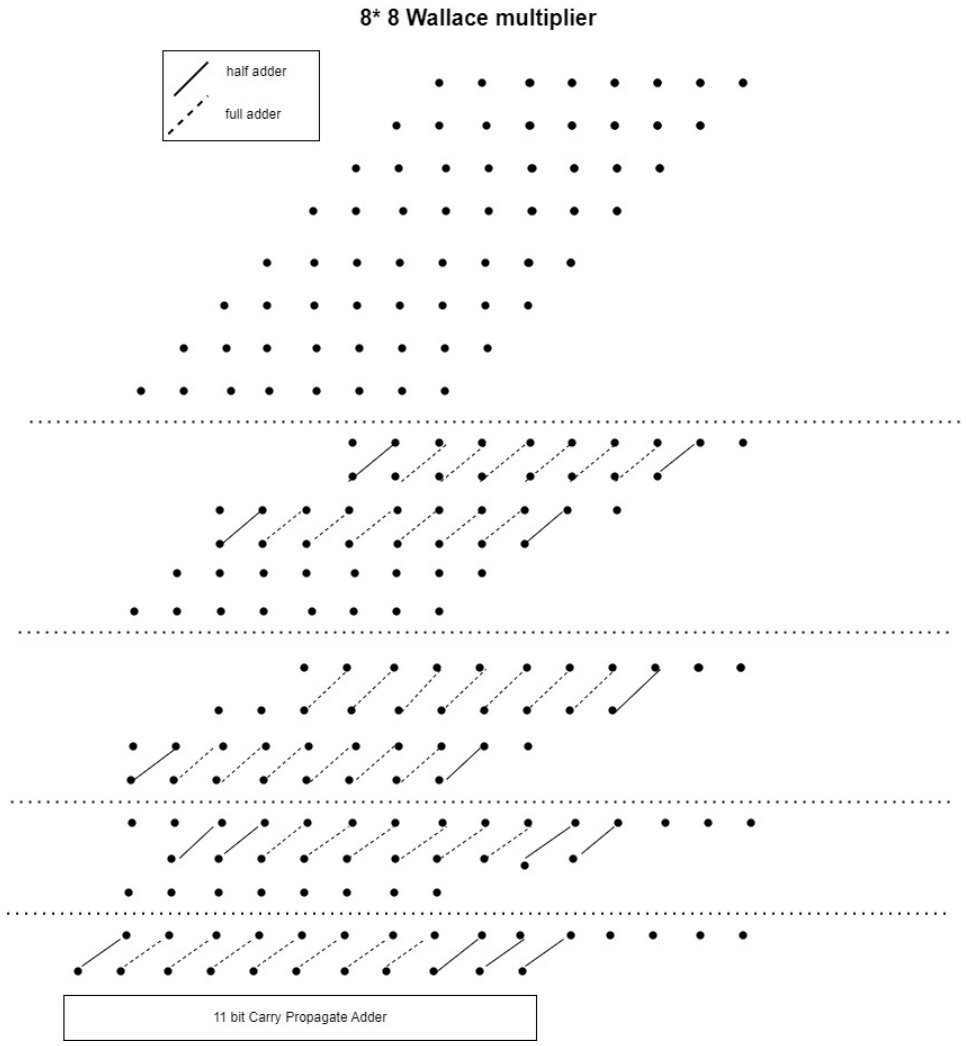

Fig 3: Dot Diagram for 8x8 Wallace Multiplier

1.4 Dadda Multiplier

The Dadda multiplier [6] is a more resource efficient fast multiplier compared to Wallace multiplier. After computing the partial products, a different reduction approach is used to lower the number of rows in the following level.. The height of the Dadda tree at the $j + 1^{th}$ stage is generalized as $d_{j+1} = [1.5{*}d_j]$ where [.] is the floor function. Ex- This means for an 8 bit multiplier we will reduce the partial products to 8⇒6⇒4⇒3⇒2 rows in subsequent steps of the reduction process. The dot diagram for 8x8 Dadda multiplier is shown in Fig 4.

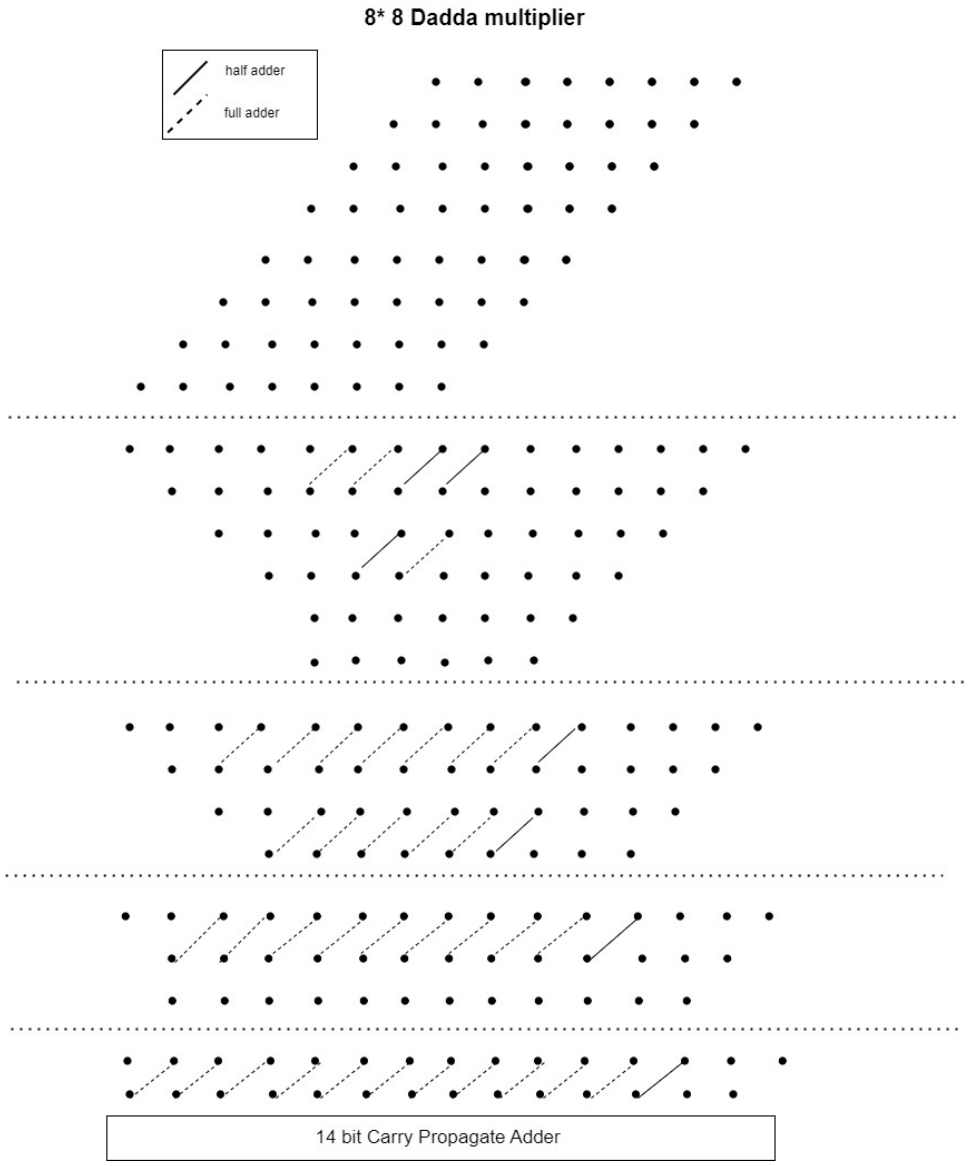

Fig 4: Dot Diagram for 8x8 Dadda Multiplier

2 Approach

The goal was to implement multiplier designs specifically for deploying on FPGAs and compare the different multiplier architectures with respect to power, performance and area. In this pursuit, the following approach was followed:

2.1 Verilog Implementation

- Verilog HDL code was written to describe the functionality of the chosen multipliers - Array multiplier, Radix-2 Booth multiplier, Wallace multiplier and Dadda multiplier.
- A total of 8 designs were coded encompassing 8x8 (8-bit) and 16x16 (16-bit) versions for each of the 4 multipliers.

- Dot diagrams were drawn for the 8x8 and 16x16 versions of Wallace and Dadda multipliers on Microsoft Excel and this was translated into Verilog code.
- From the dot diagrams of 8x8 and 16x16 Wallace and Dadda multiplier, the number of full adders, half adders and carry propagate adder sizes is estimated.

Size	Multiplier Type	Full Adders	Half Adders	CPA Width
8x8	Wallace	38	15	11-bit
16x16	Wallace	200	52	25-bit
8x8	Dadda	35	7	14-bit
16x16	Dadda	195	15	30-bit

- To make the Verilog code modular, full adders and half adders used in the Verilog design were created as separate modules across all the different multipliers.
- Simple re-usable test bench was developed in Verilog to verify the functional correctness of each of these 8 designs.
- Icarus Verilog was used to simulate the designs.

2.2 Synthesis and Implementation on FPGA

The Verilog code was synthesized and implemented for Avnet Ultra-96 FPGA. The AMD Vivado toolchain was used to generate the bitstream for the FPGA. Since the multiplier designs are all completely combinatorial, a different approach is needed to find the critical path and for estimating the worst case delay in the multiplier designs. A virtual clock was introduced as a timing constraint in the Vivado environment. This virtual clock constraint adds a flip-flop at each of the inputs and at each of the outputs while synthesizing the designs. This is shown in the figure 5 below. The tool reports the worst case timing delay and the critical path between the input and output flops. The clock period was minimized to get setup slack to be < 0.2ns. Through this process, we can estimate the maximum speed at which each of these multipliers can operate i.e. performance. The tool also reports out the power consumption and resource utilization which is used as a metric to compare the power and area respectively of different multiplier topologies.

```
create_clock -name clk_virt -period 7.4
set_input_delay -clock clk_virt 0 [get_ports a*]
set_input_delay -clock clk_virt 0 [get_ports b*]
set_output_delay -clock clk_virt 0 [get_ports prod*]
```

Fig 5: Sample Timing Constraints for Vivado Design

2.3 Deployment on FPGA

The multiplier designs were deployed on the FPGA as an AXI(Lite) Slave IP to the ARM Processor (Quad Core Cortex-A53) located on the Ultra-96 FPGA SoC. The Vivado design is shown in Fig 6. The board is shown in Fig 7. The bitstream and the address map (device-tree-blob or DTB) are then loaded onto the processor's boot partition. Register programming was used to provide inputs to the multiplier and read out the outputs from the output registers. Accessing these registers was through the Linux OS that is loaded onto the processor.

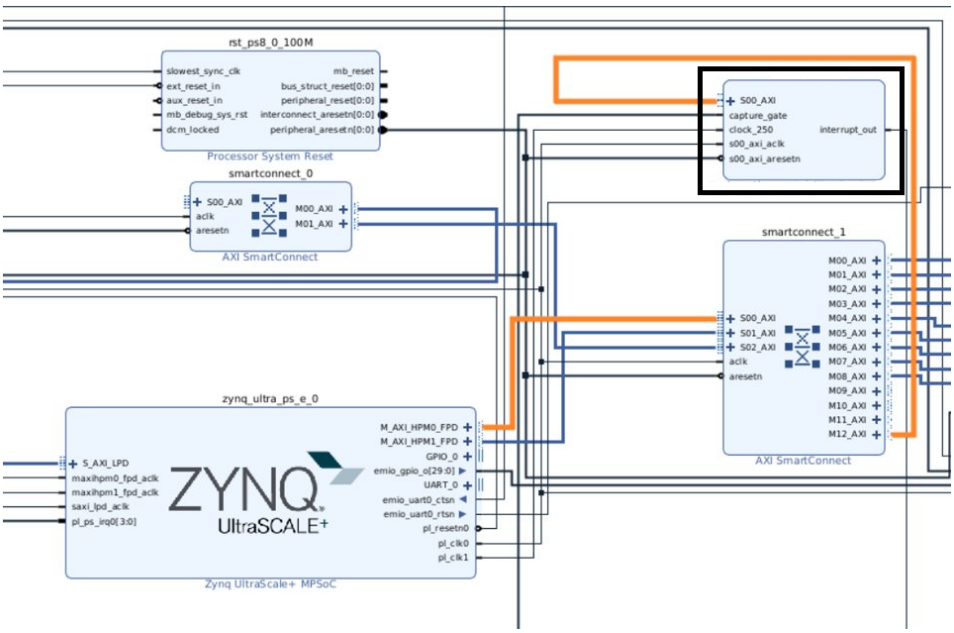

Fig 6: SoC Design for the multiplier. The highlighted path in orange shows that processor (ZYNQ PS) is connected via AXI smart connect to the multiplier. The multiplier IP is highlighted in black.

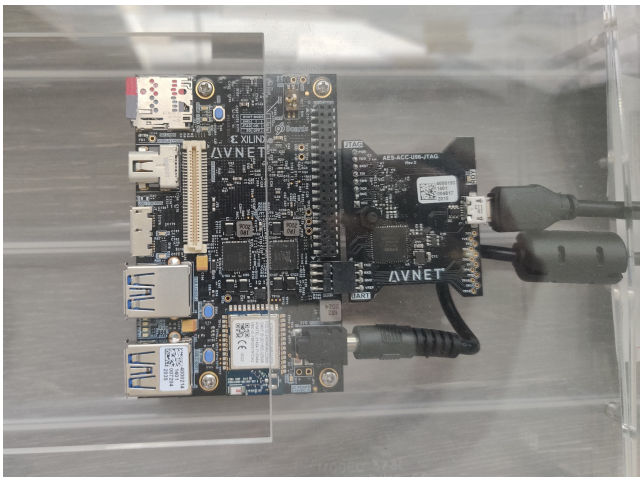

Fig 7: Avnet Ultra-96 FPGA Board.

3 Results

3.1 Simulation Results

Each of the multiplier designs was functionally verified through simulation and tested for various input combinations and all the multipliers are performing the intended operation. The simulation results are as follows:

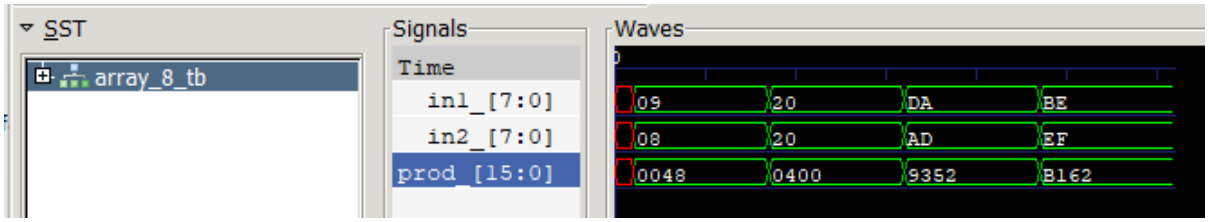

Fig 8: Simulation Results of 8x8 Array Multiplier

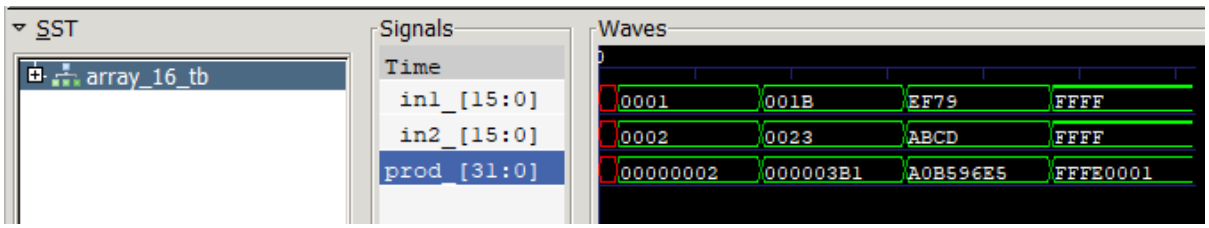

Fig 9: Simulation Results of 16x16 Array Multiplier

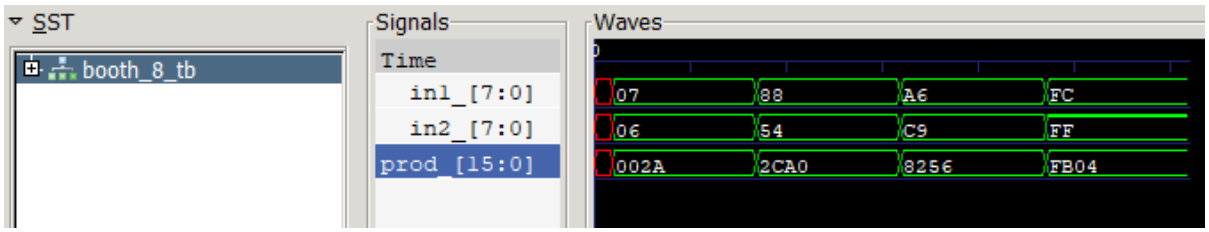

Fig 10: Simulation Results of 8x8 Booth Multiplier

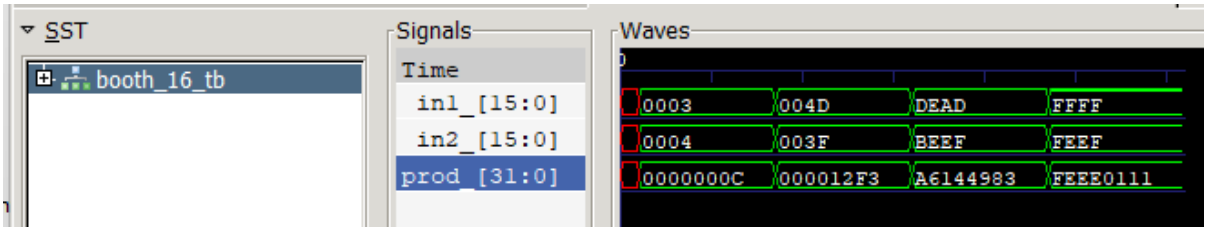

Fig 11: Simulation Results of 16x16 Booth Multiplier

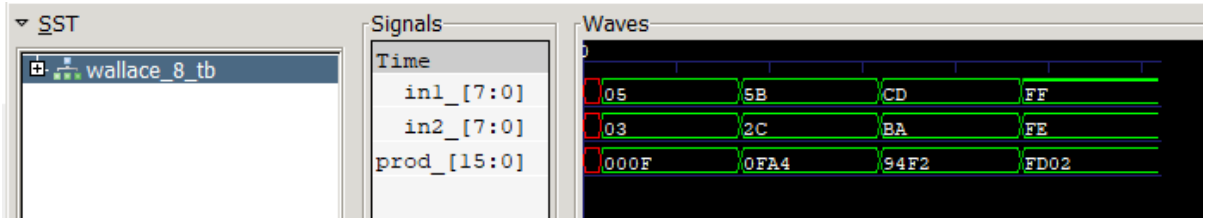

Fig 12: Simulation Results of 8x8 Wallace Multiplier

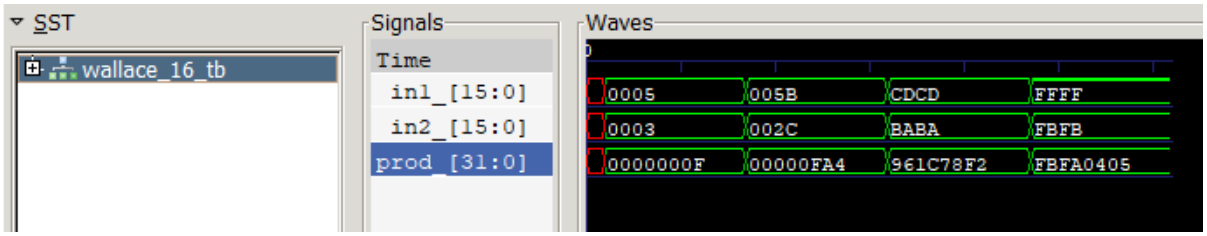

Fig 13: Simulation Results of 16x16 Wallace Multiplier

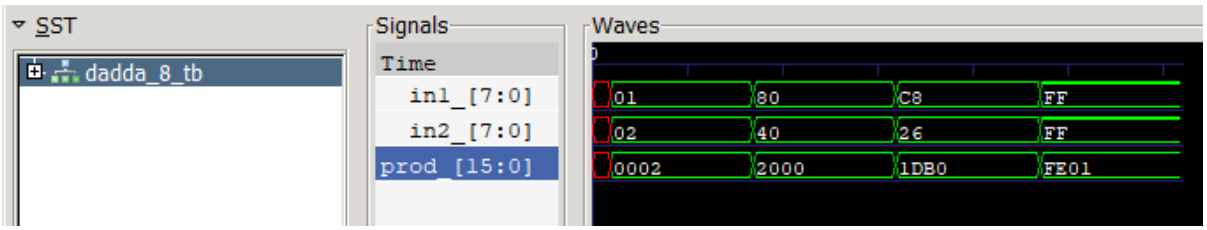

Fig 14: Simulation Results of 8x8 Dadda Multiplier

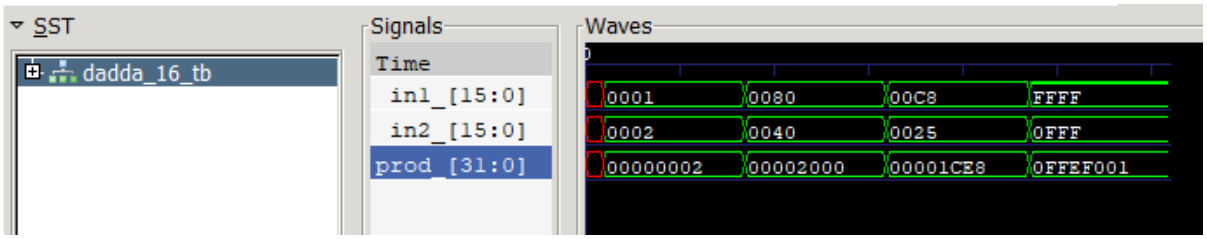

Fig 15: Simulation Results of 16x16 Dadda Multiplier

3.2 FPGA Synthesis and Implementation

Vivado tool was used to synthesize the designs followed by implementation to obtain the resource utilization and power numbers. The results are as follows:

Table 1: Resource Utilization (Post Implementation)

Multiplier Size	Multiplier Type	CLB LUTs	CLB	Bonded IOB	HPIOB_M	HPIOB_S	Total Resources
8x8	Array	90	18	32	16	16	172
8x8	Booth	129	21	32	16	16	214
8x8	Wallace	117	27	32	16	16	208
8x8	Dadda	119	33	32	16	16	216
16x16	Array	456	72	64	32	32	656
16x16	Booth	540	91	64	32	32	759
16x16	Wallace	469	79	64	32	32	676
16x16	Dadda	380	54	64	32	32	562

Table 2: Timing Closure and Power Consumption (Post Implementation)

Multiplier Size	Multiplier Type	Max Clock Frequency (MHz)	Critical Path	Power (W)
8x8	Array	175	in1[4] $\rightarrow$ *prod*[12]	2.512
8x8	Booth	135.14	in1[3] $\rightarrow$ *prod*[13]	2.508
8x8	Wallace	196.07	in1[2] $\rightarrow$ *prod*[14]	2.517
8x8	Dadda	192.31	in1[2] $\rightarrow$ *prod*[6]	2.516
16x16	Array	135.14	in2[0] $\rightarrow$ *prod*[29]	2.553
16x16	Booth	100	in2[0] $\rightarrow$ *prod*[31]	2.538
16x16	Wallace	152.67	in2[2] $\rightarrow$ *prod*[24]	2.562
16x16	Dadda	163.26	in2[12] $\rightarrow$ *prod*[31]	2.568

3.3 FPGA Deployment Results

Wallace Multipliers (both the 8x8 and 16x16 versions) were deployed onto Avnet Ultra-96 FPGA board and are checked for functionality on actual hardware. The multipliers worked functionally and below is a screenshot that shows the Linux window where the registers of the Wallace multiplier IP are being programmed to feed in the inputs and the multiplier output is being read out of another register. For example- In the 16*16 multiplier shown in Fig 17, we are feeding the first input(0x1234) to Register 0 (address: 0xa0050000) and the second input(0x98ab) to Register 1 (address: 0xa0050004). We obtain the results in Register 2 (address: 0xa0050008) as 0x0adb08bc. This matches with the theoretical result. Note that there were some bits that were added to the MSB bits of each register(0xbead to register 0, 0xfeed to register 1), just to make sure we are writing to the correct register as signature bits.

```
root@ultra96:~/hsca# pm 0xa0050000 0x12
0xa0050000 = 0xbead0012
root@ultra96:~/hsca# pm 0xa0050004 0x3f
0xa0050004 = 0xfeed003f
root@ultra96:~/hsca# dm 0xa0050008
0xa0050008 = 0xdead046e
root@ultra96:~/hsca#
```

Fig 16: FPGA Deployment Result - 8x8 Wallace Multiplier

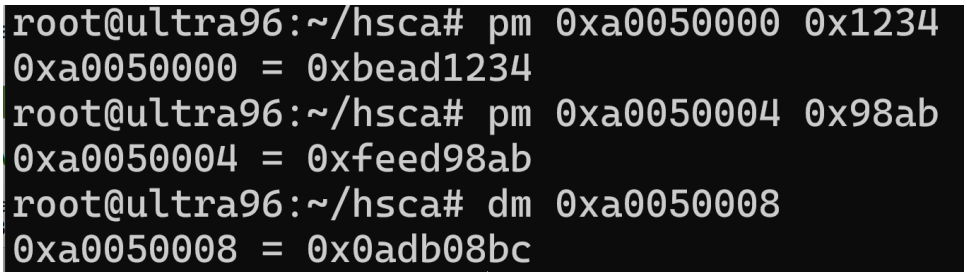

Fig 17: FPGA Deployment Result - 16x16 Wallace Multiplier

3.4 Insights

We can derive the following insights from the resource utilization table (Table 1) above -

- The Booth multiplier surprisingly takes more resources than the Array multiplier even though we expected the array multiplier to be worse in terms of resources. This is because our implementation of the Booth multiplier is a parallel version which unrolls the loop and uses separate resources for each step of the algorithm.
- The Dadda multiplier is better in resource utilization as compared to Array multiplier and Booth multiplier in 16x16 variant which was expected.
- The Dadda multiplier takes more resources compared to Wallace multiplier for the 8x8 variant. This could be because the total number of gates including the carry propagate adder is not enough to offset the reduced number of half adders in Dadda multiplier. However, in the 16x16 variant, we get expected result where Dadda multipler has lesser resource requirements.

Table 2 shows us the timing information that we get for the various multipliers from the FPGA implementation -

- The speed of operation from highest to lowest based on our FPGA implementation is as follows: Wallace multiplier $\sim$ Dadda multiplier $>$ Array multiplier $>$ Booth multiplier, as expected.
- The worst case paths of each of our multipliers do not match what we theoretically expect from the primitive gate diagrams. This is because the placer might place Configurable Logic Blocks (CLBs) connecting the different parts of the multiplier in a non-deterministic manner, which is different from when it is implemented using primitive gates. We all know that wire delays also account for a good portion of delay and if gates/CLBs that are supposed to be placed closer are placed far apart, then there would be differences in which path gets chosen as critical path.

We don't see a lot of difference between the power numbers across the multipliers. This is because our FPGA board is large and the amount of logic all the 4 multipliers take is less than 1% of the board's resources. However we note that 16x16 variants take about take 30mW-50mW more power than 8x8 variants.

4 Conclusion and Future Work

Based on the FPGA implementation that we carried out, our conclusions in terms of resource utilization and power are very similar to what we expected theoretically. The only anomaly noted was in the Booth's algorithm in which we expected to have a better resource utilization than Array multiplier but due to a parallelized approach more resources have been deployed inflating the LUT and CLB count.

The future work regarding this project involves implementation of a scalable (parameterized) Verilog code for higher multiplication order. Another approach would be to explore performance of software benchmarks (involving large number of multiplications), to see which kind of multiplier gives us the best performance.

5 References

[1] Andrew Donald Booth, "A signed binary multiplication technique," *Birbeck College Electronic Computer Project*, 1950.

[2] William Stallings, "Cryptography and network security," .

[3] Guoping Wang and J. Shield, "The efficient implementation of an array multiplier," in *2005 IEEE International Conference on Electro Information Technology*, 2005, pp. 5 pp.–5.

[4] C. Hamacher, V.C. Hamacher, Z. Vranesic, Z.G. Vranesic, and S. Zaky, *Computer Organization*, Electrical Engineering Series. McGraw-Hill Companies,Incorporated, 2002.

[5] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[6] Luigi Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, pp. 349–356, 1965.

Appendix

Verilog codes (design and testbench files), Excel sheets that contain dot diagrams for 8x8 and 16x16 versions of Wallace and Dadda multipliers are all hosted on Github. The link to the site is as follows:
`https://github.com/shreyas1998/HSCA_Multipliers`