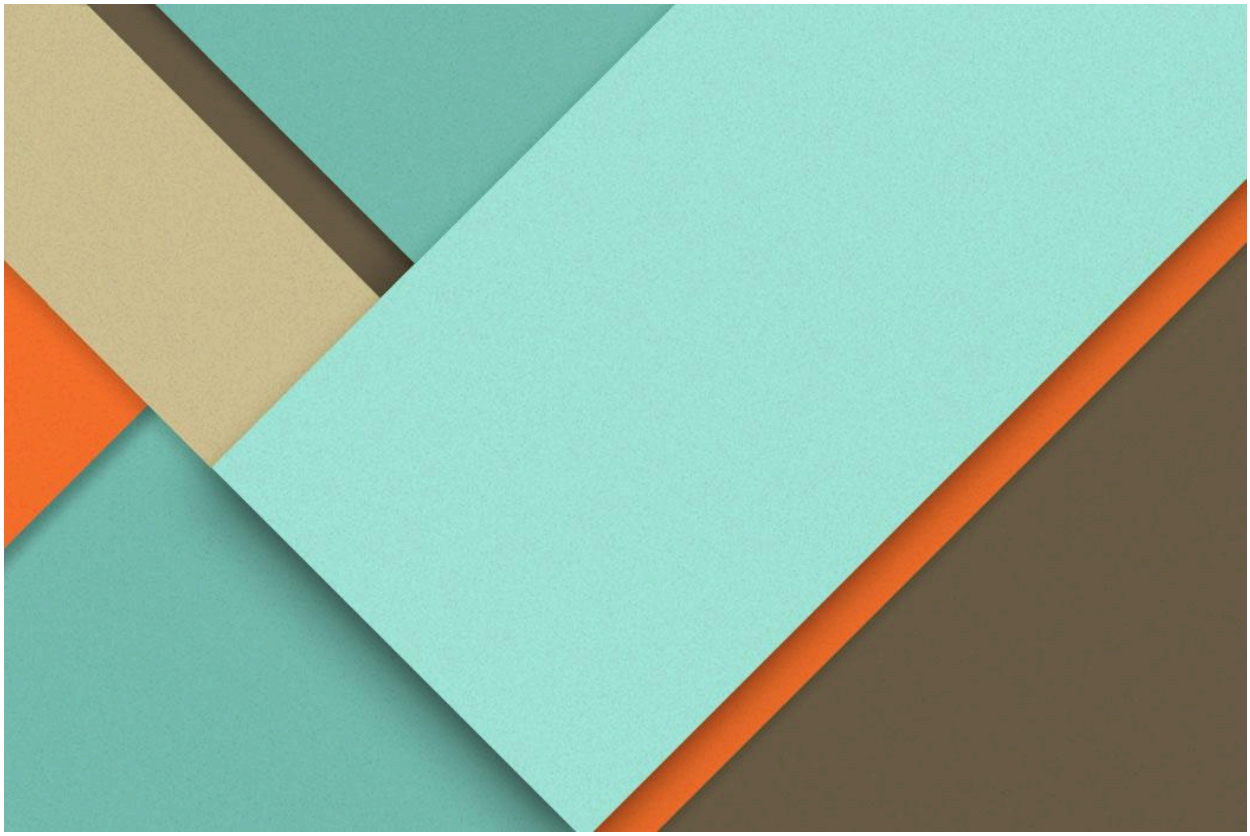


Improved Logarithmic Multipliers for Energy-Efficient Neural Networks

Sreyas Janamanchi (IMT2022554)

Chinmay Krishna (IMT2022561)

Shannon Muthanna IB (IMT2022552)

Ajitesh Kumar Singh (IMT2022559)

Margasahayam Venkatesh Chirag (IMT2022583)

Vedant Vinit Mangrulkar (IMT2022519)

Github Link

Conventional Multipliers

Wallace Tree Multiplier

Overview

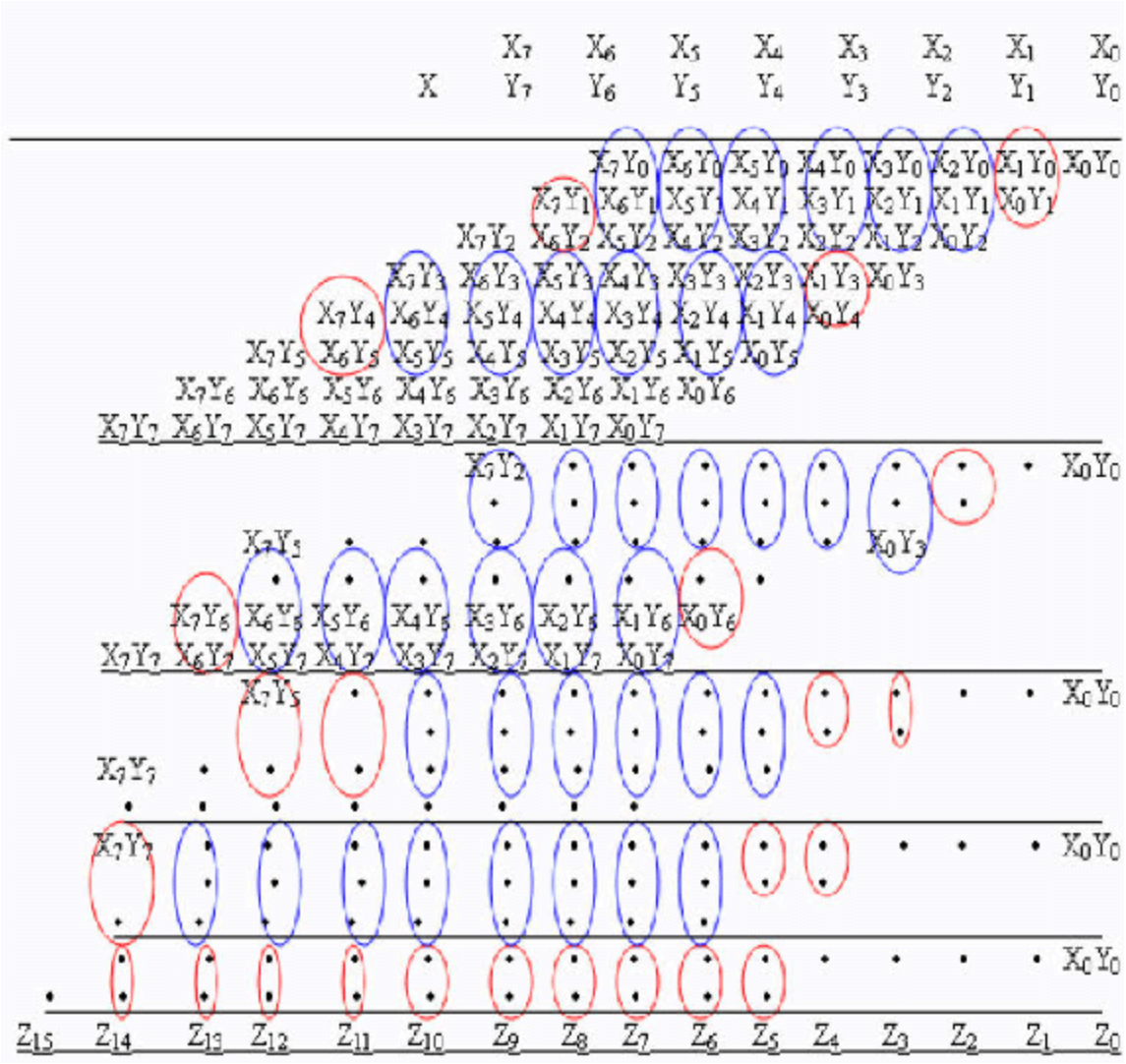
The Wallace Tree Multiplier (WTA) is a parallel multiplication technique that utilizes the Wallace Tree algorithm to efficiently perform integer multiplication.

In this approach:

- Three wires of equal weight are fed into a full adder, producing one output wire of the same weight and another with a higher weight.

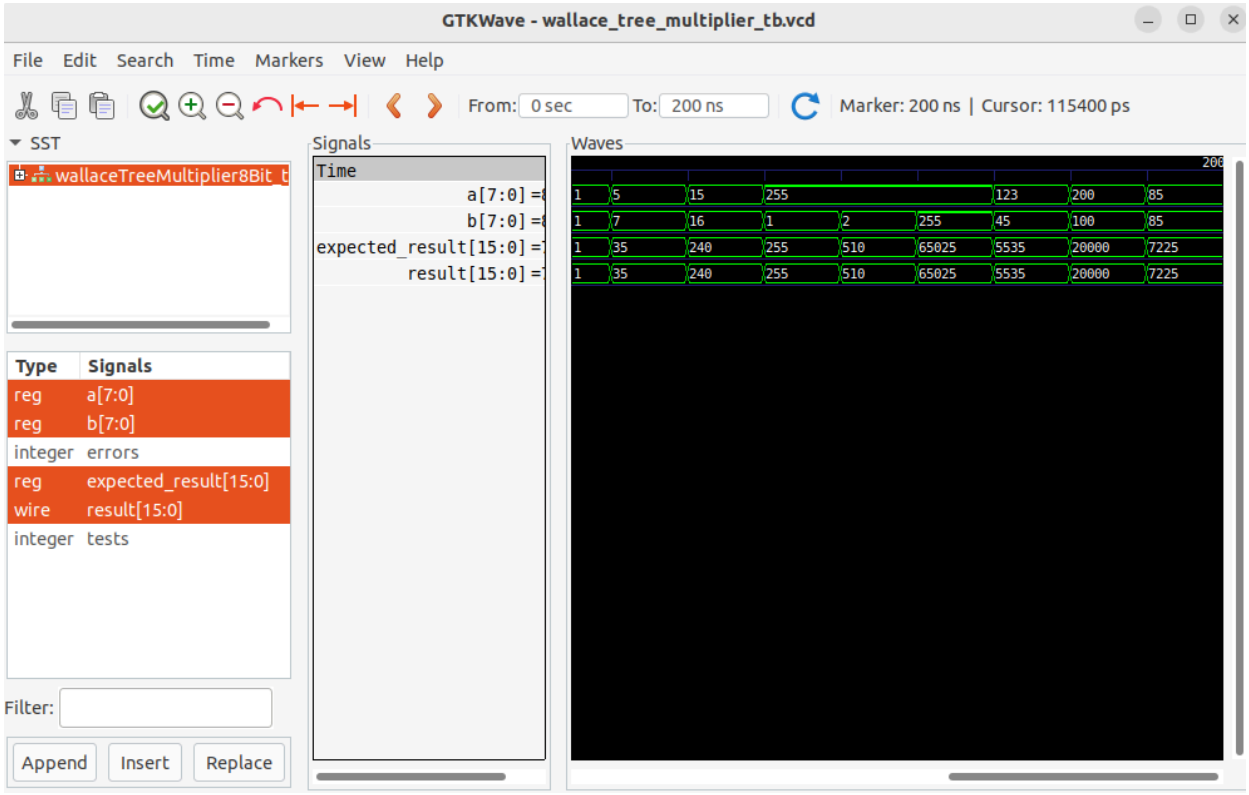
- If only two wires of the same weight remain, they are processed using a half adder.

- A single remaining wire is directly passed to the next layer.



Advantages: Fast performance with moderate complexity
Disadvantages: Large chip area
Outputs:

```
chirag@chirag-Dell-G15-5511:~/VL_Arch/Wallace_tree$ vvp wallace
VCD info: dumpfile wallace_tree_multiplier_tb.vcd opened for output.
PASS at time 110000: a=0, b=0, result=0
PASS at time 120000: a=1, b=1, result=1
PASS at time 130000: a=5, b=7, result=35
PASS at time 140000: a=15, b=16, result=240
PASS at time 150000: a=255, b=1, result=255
PASS at time 160000: a=255, b=2, result=510
PASS at time 170000: a=255, b=255, result=65025
PASS at time 180000: a=123, b=45, result=5535
PASS at time 190000: a=200, b=100, result=20000
PASS at time 200000: a=85, b=85, result=7225
```



Array Multiplier

Array multiplier is similar to how we perform multiplication with pen and paper i.e. finding a partial product and adding them together. It is a simple architecture for implementation

$$
\begin{array}{cccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & \\
 & & \times & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & p_{30} & p_{20} & p_{10} & p_{00} & & \\
 & p_{31} & p_{21} & p_{11} & p_{01} & x & & \\
p_{32} & p_{22} & p_{12} & p_{02} & x & x & & \\
p_{33} & p_{23} & p_{13} & p_{03} & x & x & x & \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}
$$

Here,

A – Multiplicand

B – Multiplier

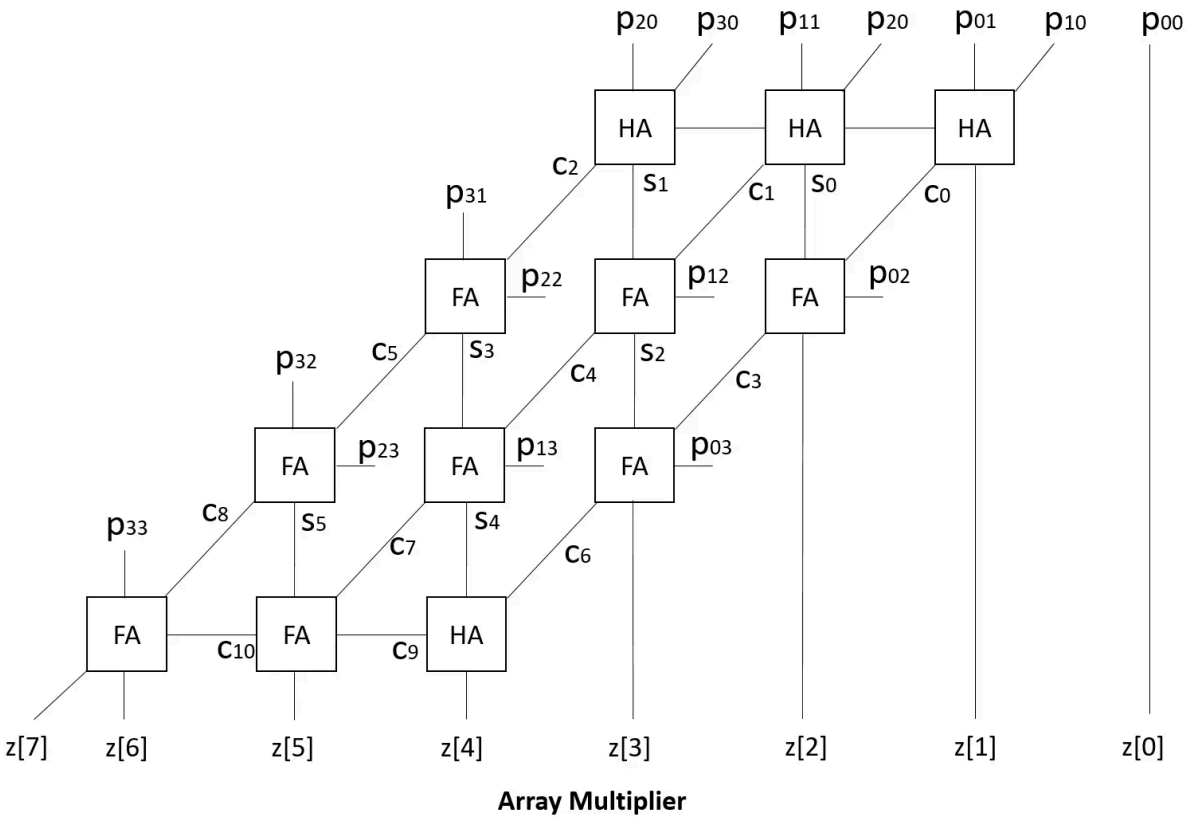
p00 – a0b0

p10 – a1b0

p20 – a2b0

.....

Diagram

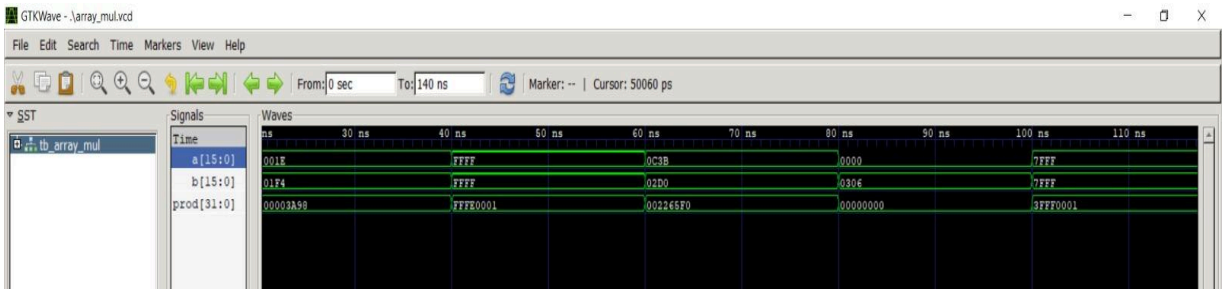


Array Multiplier

OUTPUT

```
 ~/Desktop/ArrayMultiplier                                  base  09:46:08 PM
└─ vvp multiplier.vvp
VCD info: dumpfile array_mul.vcd opened for output.
time=  0, a=    x, b=    x, prod=         x
time= 20, a=   30, b=  500, prod=     15000
```

GTKWave output



Let's extract some values from the waveform:

1. **At 30 ns:**
    - **`a = 0x001E` (Decimal: 30)**
    - **`b = 0x01F4` (Decimal: 500)**
    - **Expected product: 30×500=1500030 \times 500 = 1500030×500=15000 → `0x00003A98`**
    - **Observed `prod = 0x00003A98` ✅ Correct**
2. **At 50 ns:**
    - **`a = 0xFFFF` (Decimal: 65535)**
    - **`b = 0xFFFF` (Decimal: 65535)**
    - **Expected product: 65535×65535=429483622565535 \times 65535 = 429483622565535×65535=4294836225 → `0xFFFE0001`**
    - **Observed `prod = 0xFFFE0001` ✅ Correct**
3. **At 100 ns:**
    - **`a = 0x7FFF` (Decimal: 32767)**
    - **`b = 0x7FFF` (Decimal: 32767)**
    - **Expected product: 32767×32767=107367628932767 \times 32767 = 107367628932767×32767=1073676289 → `0x3FFF0001`**
    - **Observed `prod = 0x3FFF0001` ✅ Correct**

Advantages of 4×4 Array Multiplier

The advantages of array multiplier are,

- Minimum complexity
- Easily scalable
- Easily pipelined
- Regular shape, easy to place and route

Disadvantages of 4×4 Array Multiplier

The disadvantages of array multiplier are as follows,

- High power consumption
- More digital gates resulting in large areas.

Approximate Multipliers

Log Multipliers

Improved Log Multiplier (Selected Paper Design)

Overview of Logarithmic Multiplication

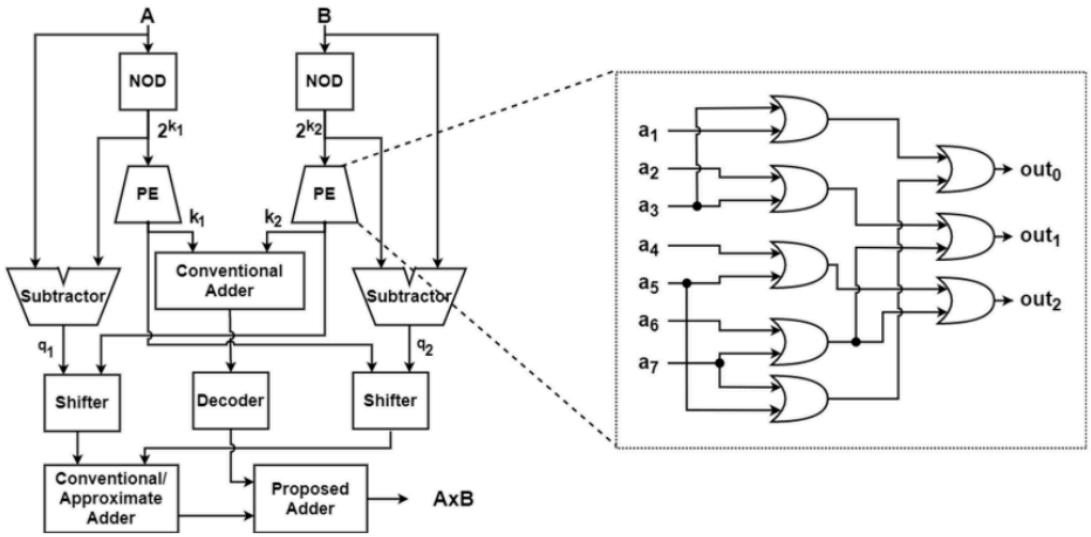
The ILM circuit implements multiplicative operations using logarithmic principles. Instead of directly multiplying two numbers A and B, it converts them to the logarithmic domain,

adds the logarithms, and then converts back to obtain the product. This approach is based on the mathematical identity:

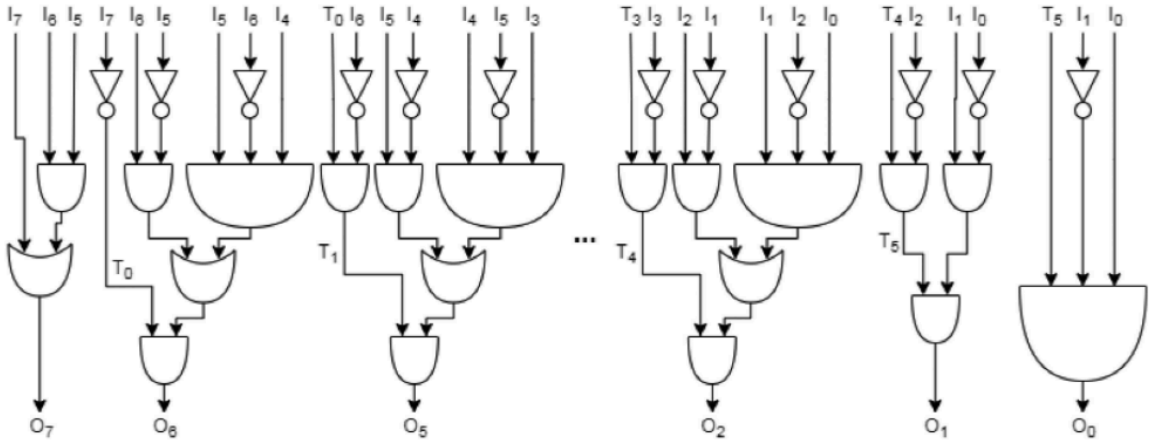
$A \times B = 2^{(\log_2 A + \log_2 B)}$

The key advantage of this method is that it can potentially reduce hardware complexity for multiplication operations. Now the multiplication operation is reduced to shift and add operation.

Algorithm and Circuit Mapping



(a) Block diagram of the ILM and the priority encoder [35].



(b) Proposed nearest-one detector (NOD) Circuit.

Algorithm 2. Proposed Logarithmic Multiplication

1: **procedure** MA, B
2: A, B: inputs, γ: approximate output
3: $2^{k_1} \leftarrow \text{NOD}(A)$,
4: $k_1 \leftarrow \text{PE}(2^{k_1})$,
5: $q_1 \leftarrow A - 2^{k_1}$, ▷ for steps 3-5 see (14)
6: $2^{k_2} \leftarrow \text{NOD}(B)$,
7: $k_2 \leftarrow \text{PE}(2^{k_2})$,
8: $q_2 \leftarrow B - 2^{k_2}$, ▷ for steps 6-8 see (15)
9: $q_1 2^{k_2} \leftarrow q_1 << k_2$,
10: $q_2 2^{k_1} \leftarrow q_2 << k_1$,
11: $2^{k_1+k_2} \leftarrow \text{DEC}(k_1 + k_2)$,
12: $\gamma \leftarrow 2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}$. ▷ see (16)

The circuit in Image 1 is a hardware implementation of the Algorithm from Image 2, which describes logarithmic multiplication. The explanation of how each component corresponds to the algorithm steps are given below:

Nearest-One Detector (NOD)

The NOD blocks in the circuit identify the position of the most significant '1' bit in the input numbers A and B. This corresponds to steps 3 and 6 in Algorithm 2:

- $2^{k_1} \leftarrow \text{NOD}(A)$
- $2^{k_2} \leftarrow \text{NOD}(B)$

The NOD circuit implementation is shown in Image 1 (b), using a network of logic gates to detect the highest '1' bit position. This effectively finds the largest power of 2 that is less than or equal to the input value based on a certain condition or else it finds the smallest power of 2 which is greater than or equal to the number. The algorithmic intuition of NOD is given in the image below. Also if the number exceeds 128 NOD always downscales the number to 128 and does not upscale it.

Algorithm 1. Proposed Approximation for $log_2 N$

1: $N = 2^k(1+x) = 2^{k+1}(1-y)$
2: **if** $N - 2^k < 2^{(k+1)} - N$ **then** ▷ use underestimate
3: $x = N/2^k - 1$
4: $log_2 N \approx k + x$
5: **else** ▷ use overestimate
6: $y = 1 - N/2^{k+1}$
7: $log_2 N \approx k + 1 - y$
8: **end if**

Priority Encoder (PE)

The PE blocks convert the power-of-2 values (2^{k_1} and 2^{k_2}) to their corresponding exponents (k_1 and k_2). This implements steps 4 and 7:

- $k_1 \leftarrow PE(2^{k_1})$
- $k_2 \leftarrow PE(2^{k_2})$

Essentially, the PE determines the binary logarithm of the highest power of 2 in each input.

Subtractors

The subtractor blocks compute:

- $q_1 = A - 2^{k_1}$ (step 5)
- $q_2 = B - 2^{k_2}$ (step 8)

These values represent the "fractional" parts of the logarithmic representation, or how much A and B exceed their respective highest powers of 2.

Shifters

The shifter blocks implement the bit-shift operations in steps 9 and 10:

- $q_1 2^{k_2} \leftarrow q_1 << k_2$ (left shift q_1 by k_2 bits)
- $q_2 2^{k_1} \leftarrow q_2 << k_1$ (left shift q_2 by k_1 bits)

These operations prepare the terms needed for the final calculation.

Decoder

The decoder implements step 11: $2^{k_1+k_2} \leftarrow DEC(k_1 + k_2)$

This converts the sum of logarithms back to the linear domain, generating $2^{(k_1+k_2)}$.

Conventional/Approximate Adder and Proposed Adder

These components work together to implement step 12: $y \leftarrow 2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}$

The final calculation combines:

- The main term: $2^{(k_1+k_2)}$
- The first correction term: $q_2 2^{k_1}$
- The second correction term: $q_1 2^{k_2}$

The result is an approximation of A×B.

Connection to Algorithm 1

Algorithm 1 (Image 3) provides the approximation method for calculating $\log_2 N$, which is fundamental to the logarithmic multiplication process. It uses a piecewise approximation:

1. If $N - 2^k < 2^{(k+1)} - N$, it uses the underestimate: $\log_2 N \approx k + x$, where $x = N/2^k - 1$
2. Otherwise, it uses the overestimate: $\log_2 N \approx k + 1 - y$, where $y = 1 - N/2^{(k+1)}$

This approximation method is implicit in how the ILM circuit processes the inputs to derive their logarithmic representations.

Circuit Operation Summary

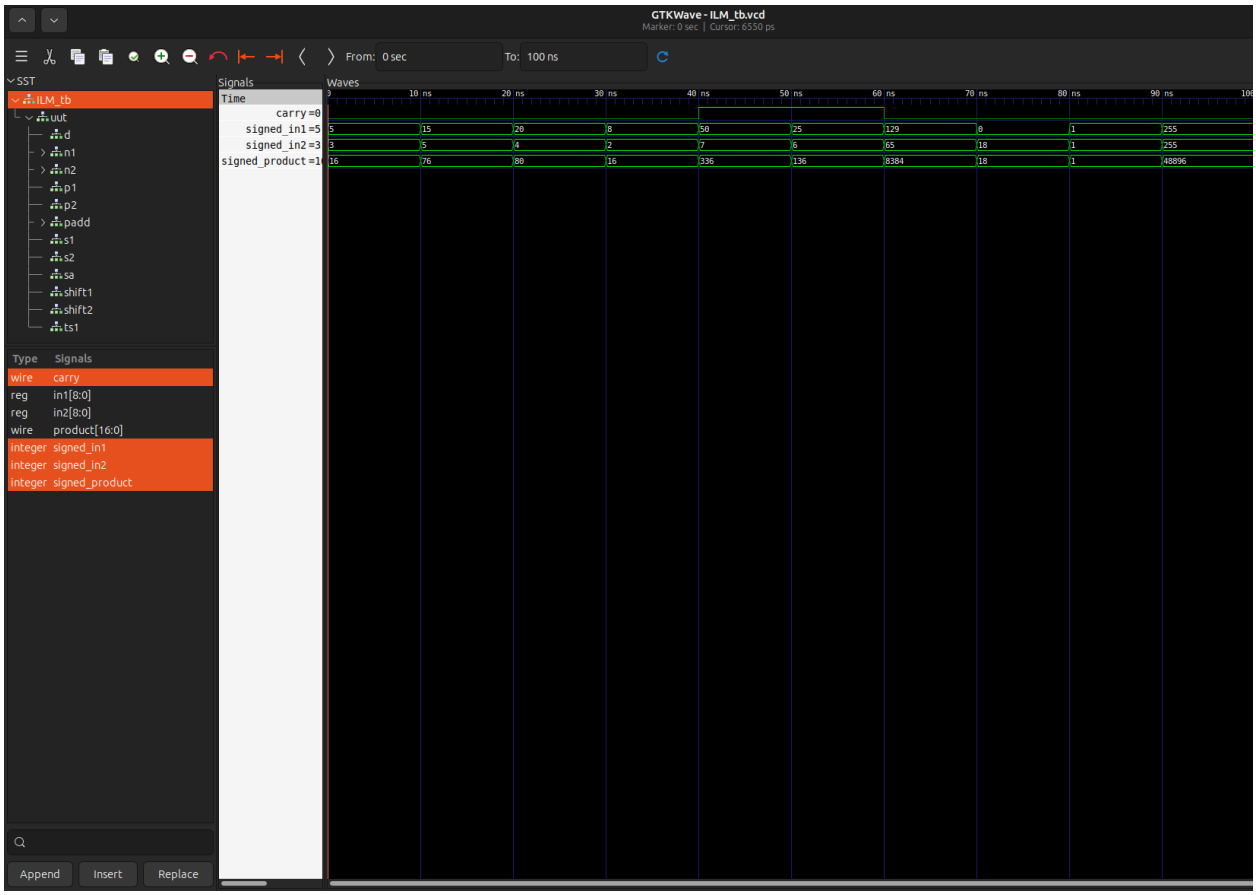
The overall operation of the ILM circuit can be summarized as:

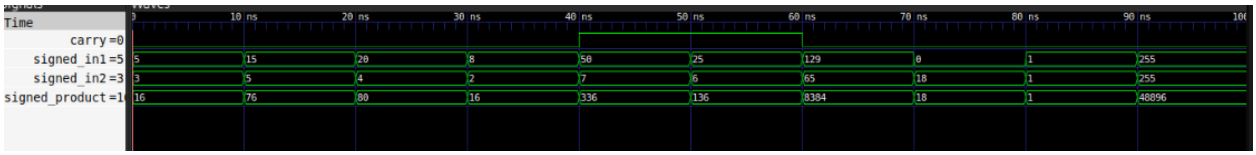
1. Decompose inputs A and B into their logarithmic components using NOD and PE
2. Calculate the error terms q_1 and q_2
3. Perform logarithmic addition $(k_1 + k_2)$
4. Convert back to linear domain and add correction terms
5. Output the final product approximation

The key insight is that by working in the logarithmic domain, multiplication is converted to addition, which can be more efficient in hardware implementation. The correction terms ($q_1 2^{k_2}$ and $q_2 2^{k_1}$) improve the accuracy of the approximation.

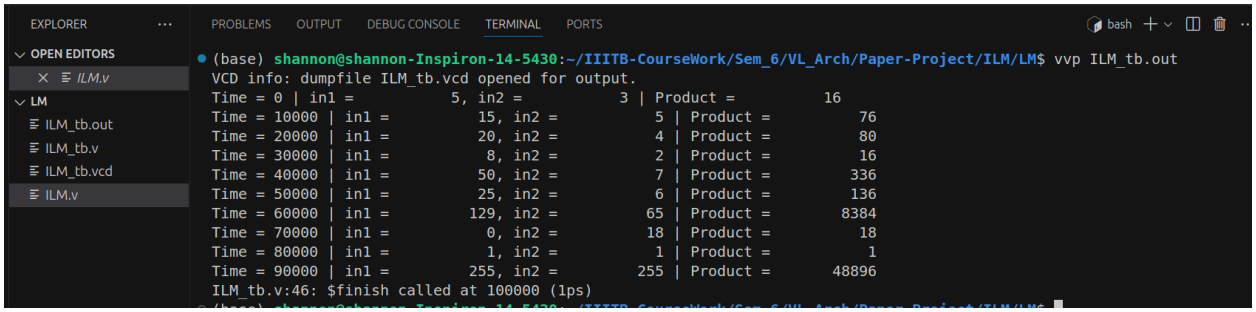
Screenshots

GTK Wave output of the ILM multiplier

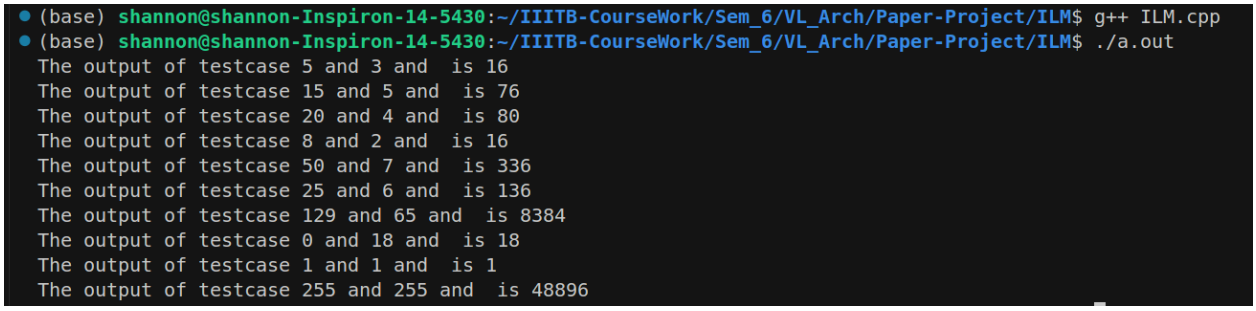




Testbench output



Cpp implementation output



Mitchell Multiplier

The Mitchell Multiplier: A Logarithmic Approach to Binary Multiplication

The Mitchell multiplier simplifies binary multiplication using logarithms. Instead of direct multiplication, it converts numbers to logarithmic form, adds them, and converts back using antilogarithms to get an approximate product.

Advantages:

• Replaces multiplication with addition and bit shifting

• Simpler hardware implementation

• Trade-off: Results are approximate

This method provides a practical approach for hardware multiplication, though with some accuracy loss due to the approximations used.

Ig N = log2N.

Consider Fig. 4. Registers A and B contain two numbers. Suppose it is desired to multiply or divide these numbers (AxB or A÷B). The word size in this example is eight bits so the largest possible characteristic will be seven. X3X2X1 and Y3Y2Y1 each initially contain 111.

Step 1—Shift A and B left until their most significant "one" bits are in the left-most positions and count down counters $x_3x_2x_1$ and $y_3y_2y_1$ during the shifting. After the shifting is completed the counters will contain the characteristics of the logarithms of A and B.

Step 2—Shift bits 0–6 of A and B into bit positions 0–6 of C and D as indicated in Fig. 4. C and D now contain the logarithms of the original numbers.

Step 3—Add or subtract $C \pm D \rightarrow E$. This puts the logarithm of the result in E.

Step 4—Decode $z_4z_3z_2z_1$ and insert a "one" in appropriate position of F. Insert the right-hand portion of E immediately to the right of this "one." F now contains the result.

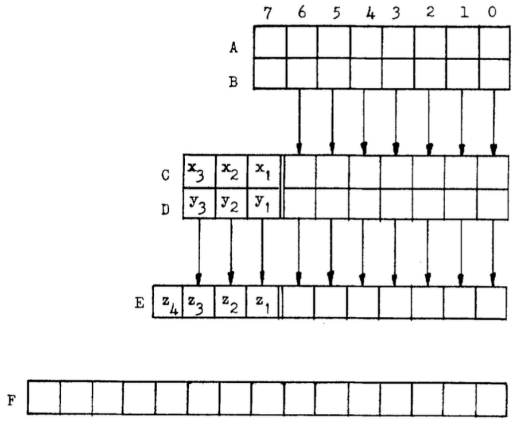


Fig. 4—Example of machine organization to generate and use binary logarithms.

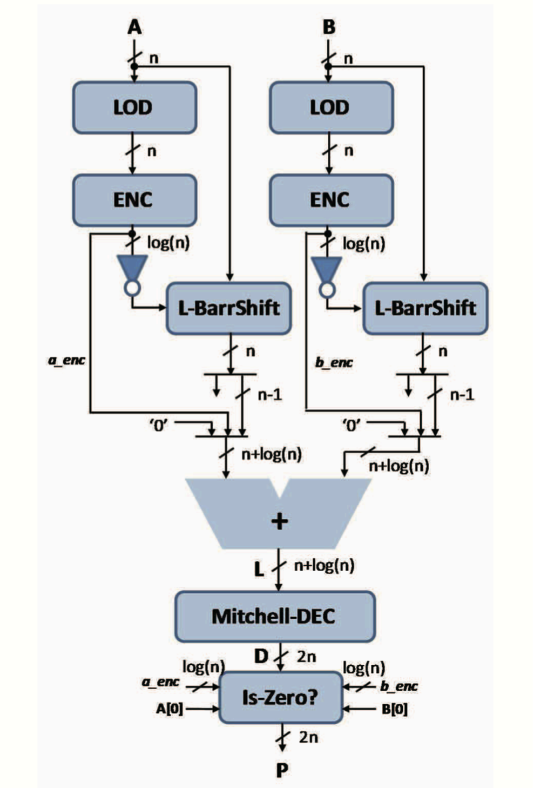


Fig. 1. Mitchell Logarithmic Multiplier according to Algorithm 1

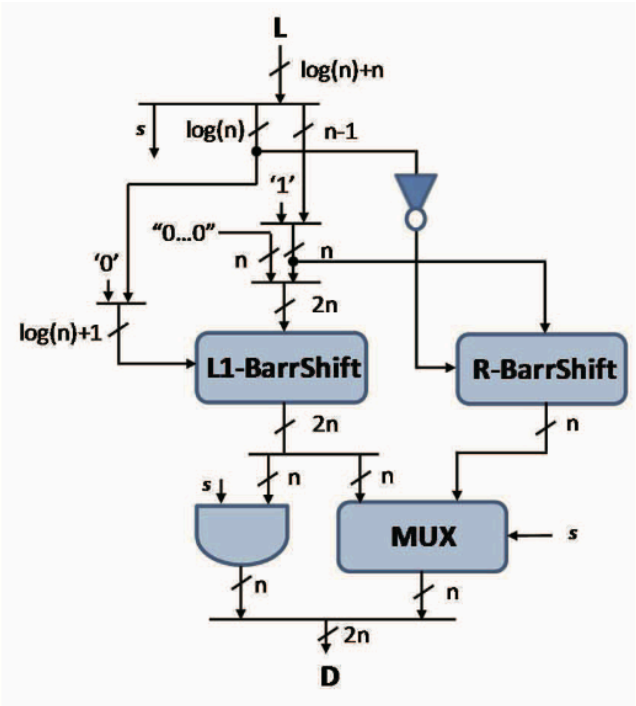


Fig. 2. Mitchell Decoder

Key Steps:

1. Normalization
   - Every positive binary number is written as $N = 2^k \cdot (1 + x)$
   - k is the position of the leftmost 1 (characteristic)
   - x is the fractional part ($0 \leq x < 1$)

   Example: 13 (1101 in binary) = $2^3 \cdot (1 + 0.625)$

2. Logarithm Approximation
   - Exact: $\lg N = k + \lg(1 + x)$
   - Mitchell's approximation: $\lg N \approx k + x$

3. Multiplication Process

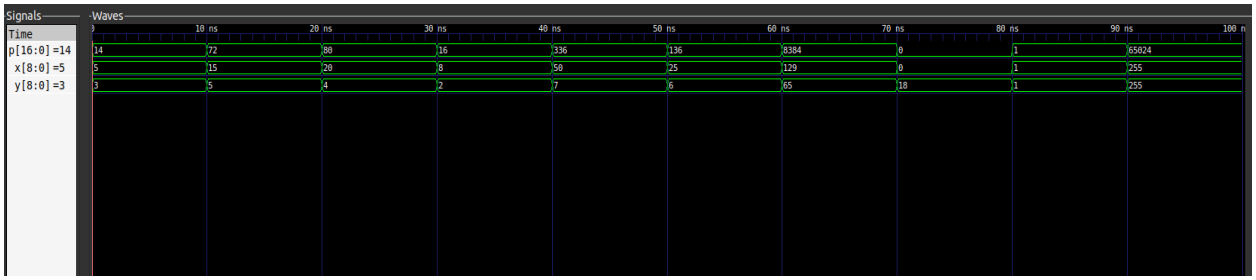
   For $A = 2^{k_1} \cdot (1 + x_1)$ and $B = 2^{k_2} \cdot (1 + x_2)$:
   - Add logarithms: $\lg P \approx (k_1 + k_2) + (x_1 + x_2)$
   - Take antilogarithm with two cases:

     Case 1 ($x_1 + x_2 < 1$): $P' = 2^{(k_1+k_2)} \cdot (1 + x_1 + x_2)$

     Case 2 ($x_1 + x_2 \geq 1$): $P' = 2^{(k_1+k_2+1)} \cdot (x_1 + x_2)$

Screenshots

GTK Wave output of the Mitchell's multiplier



Cpp implementation output

```
(base) dell@dell-Inspiron-5415:~/Desktop/IIITB/6thSem/VLSIarch/Project/mitchell$ g++ -std=c++20 mitchel.cpp
(base) dell@dell-Inspiron-5415:~/Desktop/IIITB/6thSem/VLSIarch/Project/mitchell$ ./a.out
Multiplying 5 × 3  Mitchell Approximation: 14
Multiplying 15 × 5  Mitchell Approximation: 72
Multiplying 20 × 4  Mitchell Approximation: 80
Multiplying 8 × 2  Mitchell Approximation: 16
Multiplying 50 × 7  Mitchell Approximation: 336
Multiplying 25 × 6  Mitchell Approximation: 136
Multiplying 0 × 18  Mitchell Approximation: 0
Multiplying 1 × 1  Mitchell Approximation: 1
Multiplying 129 × 65  Mitchell Approximation: 8384
Multiplying 255 × 255  Mitchell Approximation: 65024
```

Testbench output

```
(base) dell@dell-Inspiron-5415:~/Desktop/IIITB/6thSem/VLSIarch/Project/mitchell/LM$ iverilog -o MITCHEL_tb.out mitchell.v mitchel_tb.v
(base) dell@dell-Inspiron-5415:~/Desktop/IIITB/6thSem/VLSIarch/Project/mitchell/LM$ vvp MITCHEL_tb.out
VCD info: dumpfile MITCHEL_tb.vcd opened for output.
Time = 0 | x = 000000101 (  5), y = 000000011 (  3) | Product = 00000000000001110 (    14)
Time = 10000 | x = 000001111 ( 15), y = 000000101 (  5) | Product = 00000000001001000 (    72)
Time = 20000 | x = 000010100 ( 20), y = 000000100 (  4) | Product = 00000000001010000 (    80)
Time = 30000 | x = 000001000 (  8), y = 000000010 (  2) | Product = 00000000000010000 (    16)
Time = 40000 | x = 000110010 ( 50), y = 000000111 (  7) | Product = 00000000101010000 (   336)
Time = 50000 | x = 000011001 ( 25), y = 000000110 (  6) | Product = 00000000010001000 (   136)
Time = 60000 | x = 010000001 (129), y = 001000001 ( 65) | Product = 00010000011000000 (  8384)
Time = 70000 | x = 000000000 (  0), y = 000010010 ( 18) | Product = 00000000000000000 (     0)
Time = 80000 | x = 000000001 (  1), y = 000000001 (  1) | Product = 00000000000000001 (     1)
Time = 90000 | x = 011111111 (255), y = 011111111 (255) | Product = 01111111000000000 ( 65024)
```

ALM-SOA Multiplier

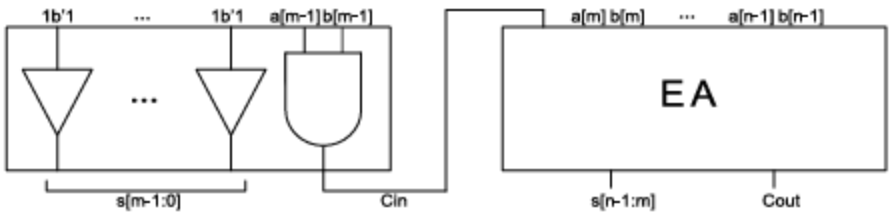


Fig. 5. SOA with m inexact bits (modified from [37]).

As shown in the above diagram the ALM-SOA uses both the set-one adder as mantissa adder and a truncated BLC(TLBC). A n-bit SOA consists of two parts, i.e. an m-bit inexact adder and an (n-m) bit exact adder.

The sum of the m-bit adder is set to logic one. The logic expressions for the lower m-bit adder are as follows:

$$Sum[m-1:0] = 1$$

$$Cin = a[m-1]b[m-1]$$

As the lower significant sum bits of SOA are set to one already, it is not necessary to perform an exact binary-logarithm conversion. Therefore, TBLC is used in the ALM-SOA to further reduce its complexity.

The number of truncated bits is the same as the number of inexact bits in SOA which is = m.

The below is the 16 bit example for the same:

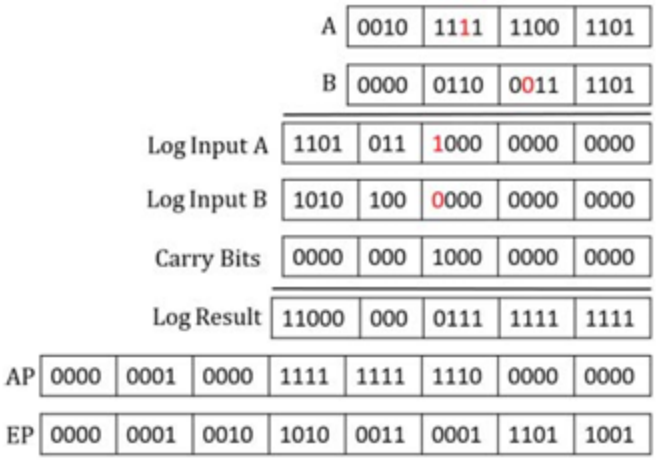


Fig. 6. A 16-bit example of ALM-SOA ($M = 11$) with $A = 12,237$ and $B = 1,597$. AP: approximate product. EP: exact product.

The ALM-SOA is designed to overcome the complexity which is required in the Mitchell adder.

It can also be said that the ALM-SOA is a modified version of the Mitchell adder.

The entire process of the ALM-SOA is similar to the Mitchell adder, but the way the addition is performed is a bit different. Rather than simply adding the log output of the BLC , we also add the Cin which is generated from the Estimator.

PROS:

- Since we are setting the lower significant sum of bits of SOA to one already, we can avoid the unnecessary process of performing an exact binary-logarithm conversion.
- To avoid the exact binary-logarithm conversion, we perform the TLBC which further reduces the complexity.

CONS:

- The accuracy is lower compared to the Mitchell multiplier at the cost of lower complexity.
- For M>4 the error tends to increase more, hence the optimal value for ALM-SOA for 8 bit multiplier is for M<5.

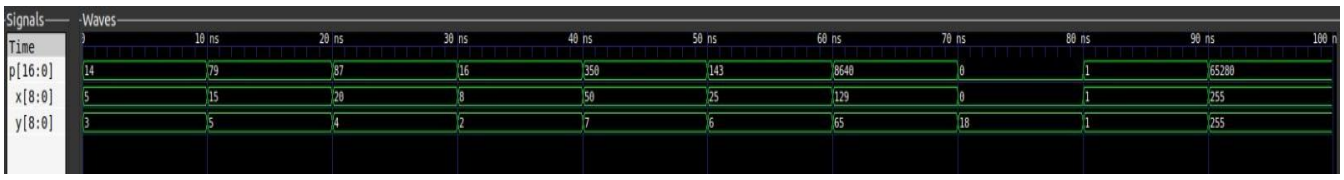
VERILOG & CPP CODE OUTPUTS:

The below is the verilog-gtkwave output for ALM-SOA-5.

The below is the cpp output for ALM-SOA-5.

```
(ap) ap@ap:~/cf$ ./a.out
Multipying 5 X 3 using alm-soa-5 gives : 15..
Multipying 15 X 5 using alm-soa-5 gives : 79..
Multipying 20 X 4 using alm-soa-5 gives : 95..
Multipying 8 X 2 using alm-soa-5 gives : 19..
Multipying 50 X 7 using alm-soa-5 gives : 382..
Multipying 25 X 6 using alm-soa-5 gives : 159..
Multipying 129 X 65 using alm-soa-5 gives : 10176..
Multipying 0 X 18 using alm-soa-5 gives : 0..
Multipying 1 X 1 using alm-soa-5 gives : 1..
(ap) ap@ap:~/cf$
```

The below is the verilog-gtkwave output for ALM-SOA-3.



The below is the cpp output for ALM-SOA-5.

```
(ap) ap@ap:~/cf$ g++ cp.cpp
(ap) ap@ap:~/cf$ ./a.out
Multipying 5 X 3 using alm-soa-5 gives : 14..
Multipying 15 X 5 using alm-soa-5 gives : 79..
Multipying 20 X 4 using alm-soa-5 gives : 87..
Multipying 8 X 2 using alm-soa-5 gives : 16..
Multipying 50 X 7 using alm-soa-5 gives : 350..
Multipying 25 X 6 using alm-soa-5 gives : 143..
Multipying 129 X 65 using alm-soa-5 gives : 8640..
Multipying 0 X 18 using alm-soa-5 gives : 0..
Multipying 1 X 1 using alm-soa-5 gives : 1..
(ap) ap@ap:~/cf$
```

Estimation of the optimal m values:

The below shown is the normalization of the mean-error-distance based on the exhaustive and monte carlo simulations.

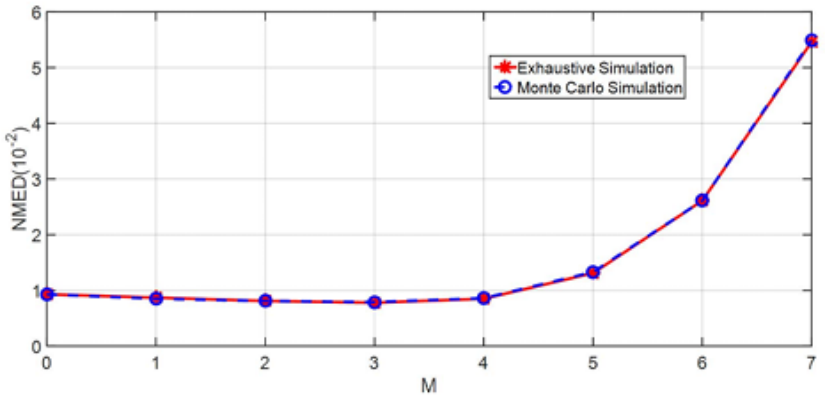


Fig. 10. NMED of 8-bit ALM-SOAs from both exhaustive and Monte Carlo simulations.

Below is the table showing the variation in the errors based on different values of M:

Design	M	Exhaustive Simulation					Monte Carlo Simulation				
		NMED (10^{-2})	MRED (10^{-2})	P_{RED} (%)	ER (%)	WCE	NMED (10^{-2})	MRED (10^{-2})	P_{RED} (%)	ER (%)	WCE

ALM-SOA	1	0.87	3.50	40.54	96.61	4032	0.85	3.50	40.77	96.27	4026
	2	0.81	3.23	45.54	97.43	4223	0.81	3.20	46.33	97.10	4223
	3	0.78	3.06	44.42	98.06	4605	0.79	3.09	44.28	98.27	4573
	4	0.85	3.44	35.62	98.42	5369	0.86	3.42	35.87	98.48	5369
	5	1.31	5.53	22.43	98.80	7680	1.33	5.44	22.68	98.78	7680
	6	2.61	11.16	11.89	98.96	15104	2.61	11.31	11.68	98.82	14880
	7	5.46	23.28	6.35	99.01	28416	5.49	23.44	6.27	99.22	28032

This method provides a practical approach for hardware multiplication and reduces the complexity of the calculations further compared to the other non-iterative logarithmic multipliers.