Array Multiplier Architecture: Design & Implementation

By: KEVAL BHAVSAR

🔹 What is an Array Multiplier?

An **Array Multiplier** is a combinational circuit used for **binary multiplication**. It is designed using **AND gates** and **adders** to perform **parallel multiplication**. The key idea behind an array multiplier is that it **mimics the traditional long multiplication method used** in arithmetic but applies it using logic gates.

📌 How It Works?

- **Partial products** are generated using **AND gates**.
- Each row is **shifted left** to align properly.
- The shifted partial products are **added** using **Half Adders (HA)** and **Full Adders (FA)** to get the final product.

🔹 **Why is it Important in Digital Circuits?**

Multiplication is a fundamental operation in **processors, ALUs (Arithmetic Logic Units)**, and **DSPs (Digital Signal Processors)**. The speed of multiplication directly affects the performance of these systems.

📌 **Key Advantages of Array Multipliers:**

✔ **Simple & Regular Structure** → Easy to design in VLSI.
✔ **Fast Execution** → Performs multiplication in parallel, unlike repeated addition.
✔ **Efficient in Hardware Implementation** → Commonly used in digital processors.

📌 **Challenges:**

❌ **Consumes More Power** → Since all operations happen simultaneously.
❌ T**akes Up Larger Area** → More gates mean more silicon area in VLSI.

Where is an Array Multiplier Used?

1 Arithmetic Logic Units (ALUs)

- The **ALU in CPUs and microcontrollers** performs arithmetic operations like addition, subtraction, and multiplication.
- **Example**: In a **microprocessor**, when executing a multiplication instruction (MUL), an **array multiplier** is used.

2 Digital Signal Processing (DSPs)

- DSPs use multiplication in **audio, video, and image processing**.
- Example: In **image filtering**, each pixel value is multiplied by a filter coefficient.

3 Image & Video Processing

- **Image convolution operations** require fast multiplications.
- Example: **Face recognition algorithms** use multiplications in matrix operations.

4 Machine Learning & AI Accelerators

- AI models involve **matrix multiplications**, which are implemented using **array multipliers.**
- Example: **Google's Tensor Processing Unit** (TPU) uses optimized multipliers for faster computations.

How Does an Array Multiplier Perform Multiplication?

The **Array Multiplier** follows the same principle as **long multiplication** in decimal but applies it to **binary numbers** using **logic gates**.

📌 Basic Concept

- Each **bit of the multiplier** is multiplied with each bit of the multiplicand using **AND gates**.
- The results (partial products) are **shifted** according to their position.
- These **partial products are added** using **Half Adders (HA)** and **Full Adders (FA)** to get the final product.

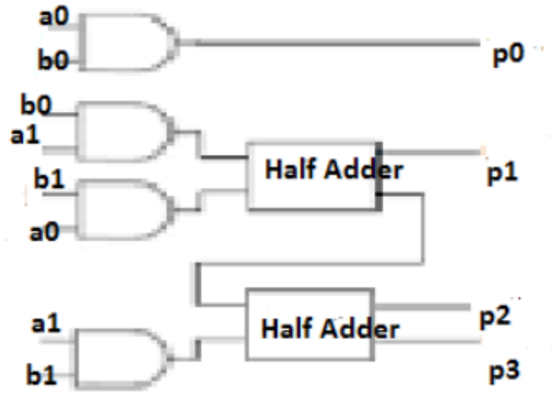


Figure 11. Block diagram of 2*2 Array Multiplier

Understanding How Binary Addition is Done in an Array Multiplier

We are multiplying two **4-bit binary numbers:**

```
Multiplicand (a) = 1011 (Decimal 11)
Multiplier (b)  = 1101 (Decimal 13)
```

Multiplication in binary follows the **same rule as decimal multiplication**:

- Multiply each bit of the **multiplier** with the **multiplicand**.
- **Shift** each row **left** depending on the position of the multiplier bit.
- **Add all the partial products** together.

Generating Partial Products:

multiplier (1101) is multiplied with the entire multiplicand (1011) using AND gates.

Multiplier Bit	Partial Product	Shifted
1 (LSB)	`1011`	No shift
0	`0000`	Shift left by 1
1	`1011`	Shift left by 2
1 (MSB)	`1011`	Shift left by 3

```
      1011   (Multiplicand = 11)
    × 1101   (Multiplier = 13)
  ------------
      1011    (1011 × 1)
+   0000    (1011 × 0, shifted)
+  1011    (1011 × 1, shifted)
+ 1011   (1011 × 1, shifted)
------------
  10001111   (Final Product = 143)
```

```
     1011      (1 × 1011, No Shift)
+   0000      (0 × 1011, Shift Left 1)
+  1011      (1 × 1011, Shift Left 2)
+ 1011      (1 × 1011, Shift Left 3)
----------------
```

Adding the Partial Products:

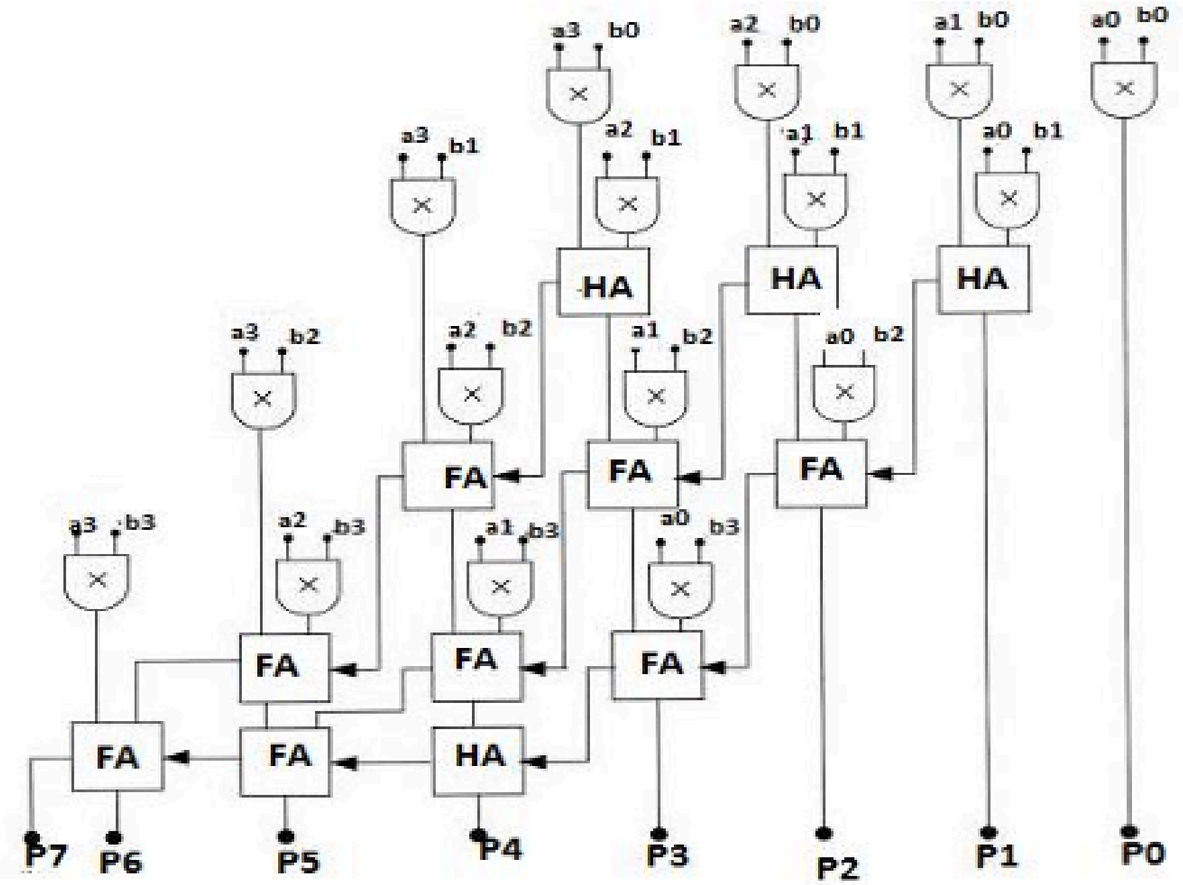
```
  1011   (Binary for 11)
× 1101   (Binary for 13)
---------------
= 10001111  (Binary for 143)
```

```
       1011    (Multiplicand = 11)
     × 1101    (Multiplier = 13)

      1011       (1 × 1011, No Shift)
 +    0000       (0 × 1011, Shift Left 1)
 +   1011       (1 × 1011, Shift Left 2)
 +  1011       (1 × 1011, Shift Left 3)
-----------------
_  10001111 (Binary) = 143 (Decimal)
```

```
      00001011   (First row: 1011 × 1 → No Shift)
 +    00000000   (Second row: 1011 × 0 → Shift Left 1)
 +    00101100   (Third row: 1011 × 1 → Shift Left 2)
 +    10110000   (Fourth row: 1011 × 1 → Shift Left 3)
 -----------------
      10001111   (Final Product in Binary)
```

Gate-Level Implementation of a 4×4 Array Multiplier

- Each bit of the **multiplicand (a3, a2, a1, a0)** is ANDed with each bit of the **multiplier (b3, b2, b1, b0)**.
- This forms **16 AND gates**, producing **4 rows of partial products**.



✓ **Half Adders (HA)** = 4
✓ **Full Adders (FA)** = 8

Figure 14. Block diagram of 4*4 Array Multiplier

Functionality of the 4×4 Array Multiplier

1 **P0 is directly from an AND gate →** It represents the least significant bit (LSB) of the product and does not require any addition.

2 **The first row of Half Adders (HA) processes the second column (P1, P2, P3, P4) →** Each HA adds two bits from the partial products and passes its carry to the next stage.

3 **The carry from Half Adders (HA) is passed into Full Adders (FA) →** Full Adders take three inputs (a partial product, a sum from the previous adder, and a carry from the previous column) and generate a sum and carry.

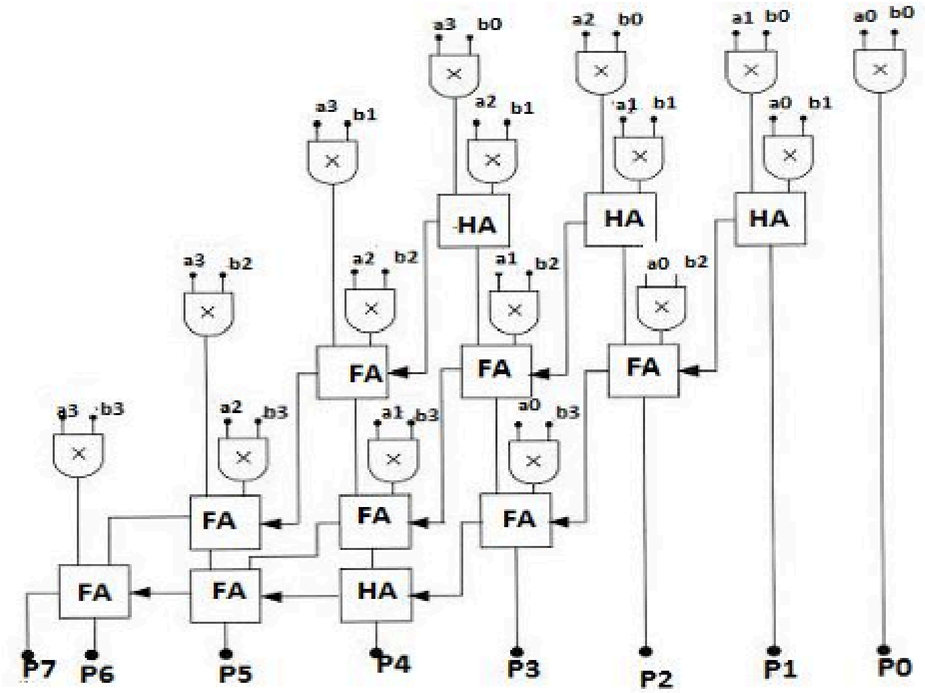


Figure 14. Block diagram of 4*4 Array Multiplier

Functionality of the 4×4 Array Multiplier

4 **Each Full Adder (FA) continues the summation process** → The sum is stored in the respective column, while the carry is passed downward to the next Full Adder.

5 **The process continues until we reach the last columns (P6 and P7)** → In these final stages, the last Full Adders complete the addition of the remaining bits and carry values.

6 **The final Full Adder (FA in P7) generates the most significant bit (MSB) of the product** → This is the last bit of the multiplication result, representing the highest value place in the binary product.

✓ **Final Output**: The product of the 4-bit numbers is obtained as **P7 P6 P5 P4 P3 P2 P1 P0**, representing an **8-bit result**.

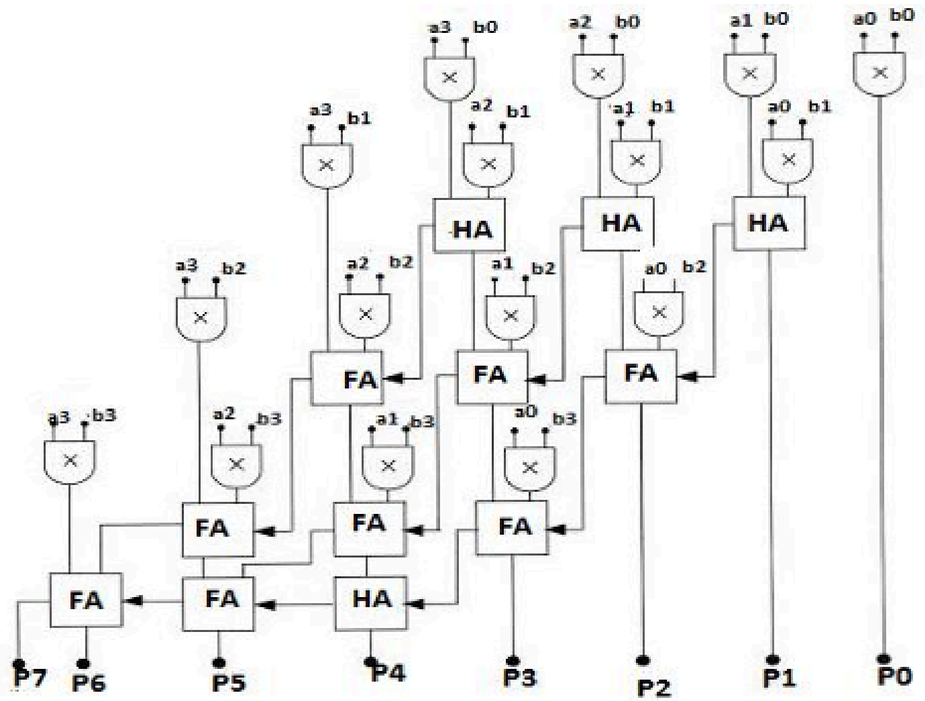


Figure 14. Block diagram of 4*4 Array Multiplier

```v
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

`module array_multiplier` → Defines the **main multiplication module**.

`#(parameter N = 4)` → Defines **N as a parameter** (default `N=4`, meaning a **4×4 multiplier**).

`input wire [N-1:0] a, b` → These are the **two 4-bit inputs** (multiplicand and multiplier).

`output wire [2*N-1:0] product` → This is the **8-bit product output** (`2*N = 8` bits for a 4-bit multiplier).

`partial_products [N-1:0]` → Stores **all the AND gate outputs** (each row of partial products).

`sum [N-1:0]` → Stores **intermediate sum values** while adding partial products.

```verilog
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

`genvar i, j;`

- `genvar` is used for **generating multiple instances** of logic.
- `i` represents the **row index**, and `j` represents the **column index**.

`generate ... endgenerate`

- This is a special Verilog construct that allows the **creation of multiple hardware structures dynamically**.
- Here, it is used to **generate** `N × N` **AND gates**.

`for (i = 0; i < N; i = i + 1)`

- Loops through each **row of partial products**.
- The value of `i` corresponds to a **specific bit of** `b`.

`for (j = 0; j < N; j = j + 1)`

- Loops through each **column of partial products**.
- The value of `j` corresponds to a **specific bit of** `a`.

`assign partial_products[i][j] = a[j] & b[i];`

- Performs **bitwise AND** operation between each bit of `a` and `b`.
- Each result is stored in **partial_products[i][j]**.

```verilog
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

`sum[0]` **stores the first row of partial products.**

`{{N{1'b0}}, partial_products[0]};`

- `{N{1'b0}}` → This **adds** `N` **zeros** to the left of `partial_products[0]` for **proper alignment.**
- **Example:**
  - If `partial_products[0] = 1011`, then `sum[0] = 00001011` (ensuring an 8-bit output).

This is needed to prepare for addition later.

```verilog
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

Each row of partial products is aligned (shifted) and added.

`{ {N{1'b0}}, partial_products[i] } << i`

- This **shifts the partial product left** (`i` times).
- **Example:**
  - `partial_products[1]` → Shift left by 1.
  - `partial_products[2]` → Shift left by 2.

`sum[i] = sum[i-1] + shifted_partial_product;`

- The **new shifted row is added** to the previous sum.

Row	Partial Product	Shifted	Sum
Row 0	`1011`	No Shift	`00001011`
Row 1	`0000`	Shift Left 1	`00000000`
Row 2	`1011`	Shift Left 2	`00101100`
Row 3	`1011`	Shift Left 3	`10110000`

```
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

After all **partial products are generated, shifted, and summed**, the **final multiplication result is stored in** `sum[N-1]`.

`sum[N-1]` is assigned to `product`, which represents the **final 8-bit multiplication result**.

```
// Code your design here
module array_multiplier #(parameter N = 4) (
    input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [2*N-1:0] product
);
    wire [N-1:0] partial_products [N-1:0];  // Array for partial products
    wire [2*N-1:0] sum [N-1:0];  // Array for sum results

    genvar i, j;
    generate
        // Generate partial products
        for (i = 0; i < N; i = i + 1) begin : gen_partial_products
            for (j = 0; j < N; j = j + 1) begin : gen_partial
                assign partial_products[i][j] = a[j] & b[i];
            end
        end
    endgenerate

    // Initialize sum[0] with the first row of partial products
    assign sum[0] = {{N{1'b0}}, partial_products[0]};

    // Add partial products row-wise
    generate
        for (i = 1; i < N; i = i + 1) begin : gen_addition
            assign sum[i] = sum[i-1] + ({ {N{1'b0}}, partial_products[i] } << i);
        end
    endgenerate

    // Final product assignment
    assign product = sum[N-1];

endmodule

// ====================================
// Top-Level Module for Qflow Synthesis
// ====================================
module array_multiplier_top;
    parameter N = 4;
    reg [N-1:0] a, b;
    wire [2*N-1:0] product;

    // Instantiate Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

endmodule
```

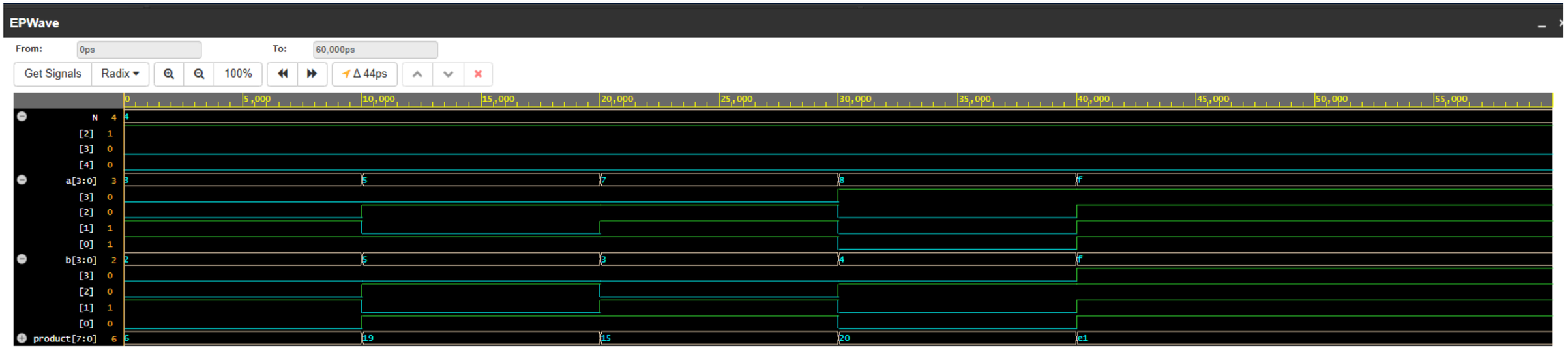
1. **Defines a separate module (array_multiplier_top)** → This is the main module used for testing and synthesis in Qflow.
2. **Uses parameters to set the multiplier size (N=4)** → This makes it flexible for different bit-width multiplications.
3. **Registers (a and b) store input values** → These registers hold the two numbers to be multiplied, which can be changed during simulation.
4. **A wire (product) stores the multiplication result** → This wire connects to the main multiplier module and receives the computed output.
5. **Instantiates the array_multiplier module inside →** The top module creates an instance of the multiplier and connects it to its inputs and outputs.

```
module array_multiplier_tb;

    // Parameter
    parameter N = 4; // Set the bit-width (must match the main module)

    // Inputs
    reg [N-1:0] a, b;

    // Outputs
    wire [2*N-1:0] product;

    // Instantiate the Array Multiplier
    array_multiplier #(.N(N)) uut (
        .a(a),
        .b(b),
        .product(product)
    );

    // Dump waveforms for EDA Playground EPA
    initial begin
        $dumpfile("waveform.vcd"); // VCD file for waveform visualization
        $dumpvars(0, array_multiplier_tb);
    end

    // Test cases
    initial begin
        // Display header
        $display("Time | A    B    | Product");
        $display("-----|----------|---------");

        // Test case 1: 3 x 2 = 6
        a = 4'b0011; b = 4'b0010; #10;
        $display("%4t | %b %b | %b (%d)", $time, a, b, product, product);

        // Test case 2: 5 x 5 = 25
        a = 4'b0101; b = 4'b0101; #10;
        $display("%4t | %b %b | %b (%d)", $time, a, b, product, product);

        // Test case 3: 7 x 3 = 21
        a = 4'b0111; b = 4'b0011; #10;
        $display("%4t | %b %b | %b (%d)", $time, a, b, product, product);

        // Test case 4: 8 x 4 = 32
        a = 4'b1000; b = 4'b0100; #10;
        $display("%4t | %b %b | %b (%d)", $time, a, b, product, product);

        // Test case 5: 15 x 15 = 225
        a = 4'b1111; b = 4'b1111; #10;
        $display("%4t | %b %b | %b (%d)", $time, a, b, product, product);

        // End simulation
        #10;
        $finish;
    end

endmodule
```

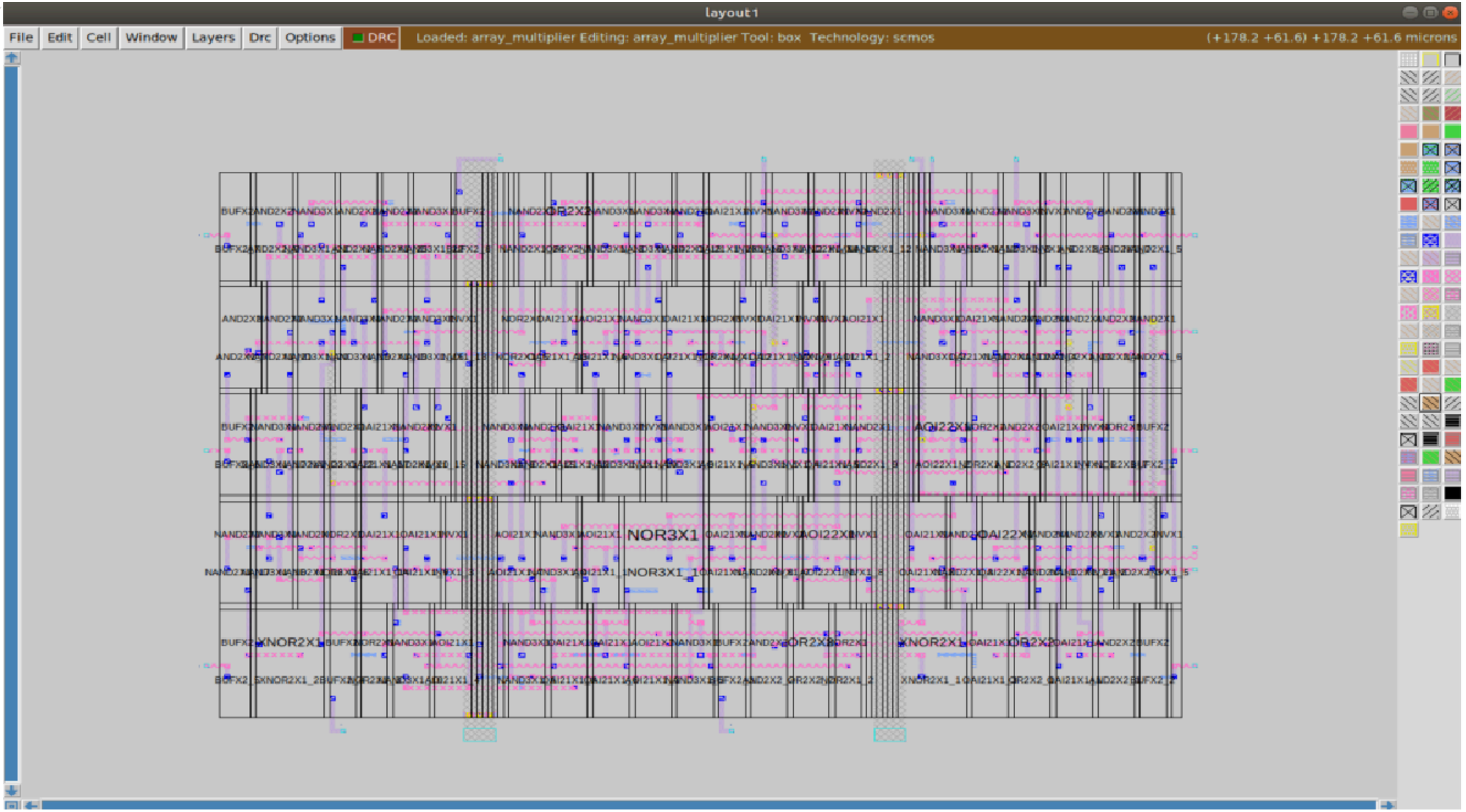
1. **Defines a testbench module (array_multiplier_tb) →** This is used to verify the functionality of the array_multiplier module.
2. **Uses a parameter (N=4) to set the bit-width →** Ensures that the testbench matches the multiplier size.
3. **Registers (a and b) hold input values →** These represent the two numbers being multiplied and are assigned different test values during simulation.
4. **A wire (product) stores the multiplication result →** It connects to the instantiated multiplier module and captures the output.
5. **Instantiates the array_multiplier module inside →** Connects inputs (a, b) and output (product) to the actual multiplier.
6. **Enables waveform dumping ($dumpfile, $dumpvars) →** This allows viewing simulation results in tools like EDA Playground.
7. **Includes multiple test cases →** Different values are assigned to a and b, and the results are displayed using $display.
8. **Ends the simulation using $finish →** Stops the testbench after all test cases are executed.

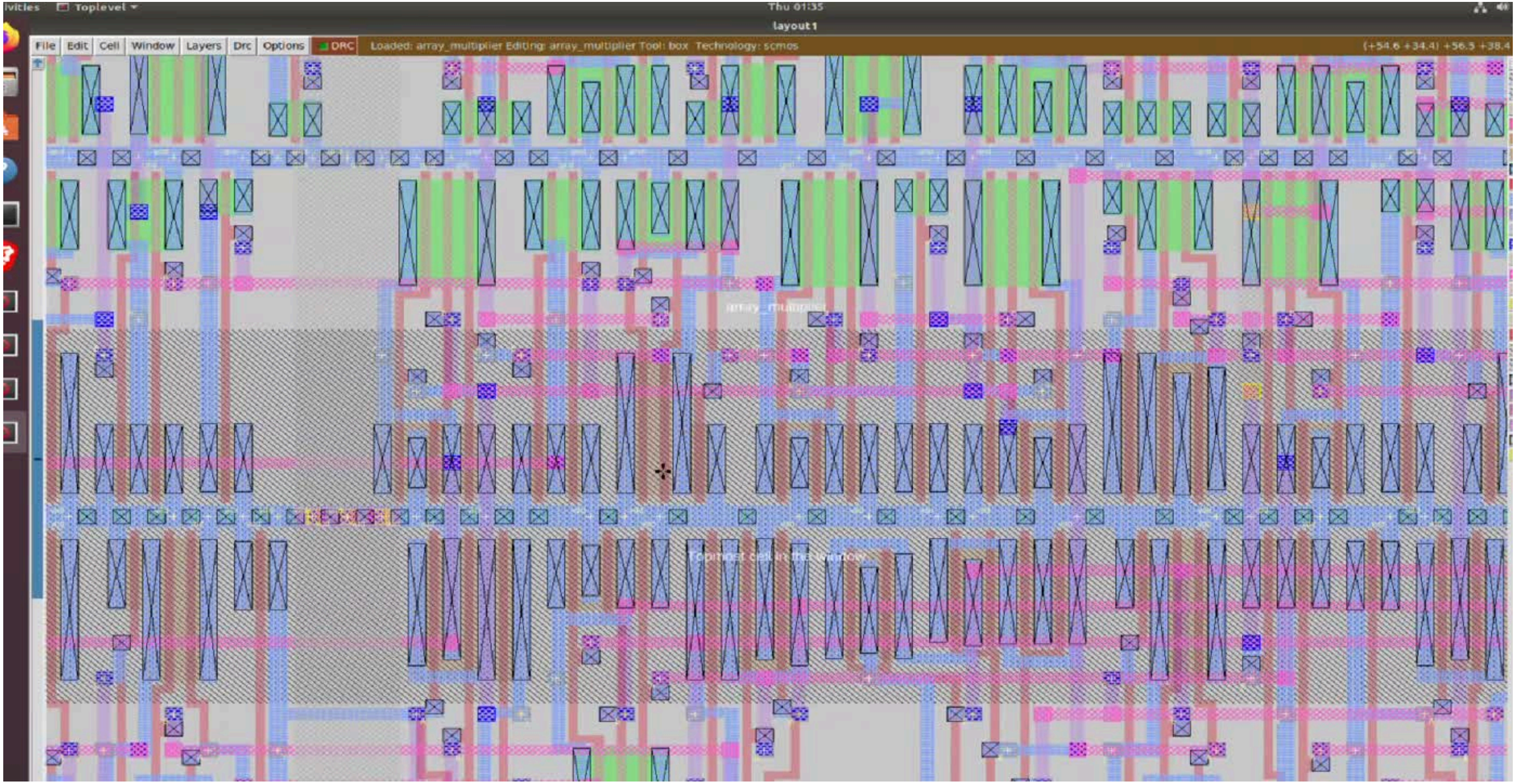
OUTPUT

```
[2025-02-26 20:25:57 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
warning: Some design elements have no explicit time unit and/or
       : time precision. This may cause confusing timing results.
       : Affected design elements are:
       :   -- module array_multiplier declared here: design.sv:2
       :   -- module array_multiplier_top declared here: design.sv:38
VCD info: dumpfile waveform.vcd opened for output.
Time | A    B    | Product
-----|----------|---------
10000 | 0011 0010 | 00000110 (  6)
20000 | 0101 0101 | 00011001 ( 25)
30000 | 0111 0011 | 00010101 ( 21)
40000 | 1000 0100 | 00100000 ( 32)
50000 | 1111 1111 | 11100001 (225)
testbench.sv:57: $finish called at 60000 (1ps)
Finding VCD file...
./waveform.vcd
[2025-02-26 20:25:58 UTC] Opening EPWave...
Done
```



Note: To revert to EPWave opening in a new browser window, set that option on your profile page.


layout1
File
Edit
Cell
Window
Layers
Drc
Options
DRC
Loaded: array_multiplier Editing: array_multiplier Tool: box Technology: scmos
(+178.2 +61.6) +178.2 +61.6 microns
NOR3X1
AOI22X1
OAI22X1
XNOR2X1
OR2X2


Toplevel
Thu 01:35
layout1
File
Edit
Cell
Window
Layers
Drc
Options
DRC
Loaded: array_multiplier Editing: array_multiplier Tool: box Technology: scmos
(+54.6 +34.4) +56.5 +38.4
array_multiplier
Pointer cell in the window

REFERENCES

- https://www.researchgate.net/publication/330652518_Design_and_Implementation_of_Low_Power_Multiplier_Using_Proposed_Two_Phase_Clocked_Adiabatic_Static_CMOS_Logic_Circuit

- https://www.researchgate.net/publication/266890236_Comparative_Analysis_of_4-bit_CMOS_Multipliers

International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 6, December 2018, pp. 4959~4971
ISSN: 2088-8708, DOE: 10.11591/ijece.v8i6.pp4959-4971 ❒ 4959

Design and Implementation of Low Power Multiplier Using Proposed Two Phase Clocked Adiabatic Static CMOS Logic Circuit

Minal Keote[1], P. T. Karule[2]
[1]Department of Electronics and Telecommunication Engineering, Y.C.C.E., India
[2]Department of Electronics Engineering, Y.C.C.E., India

Article Info
Article history:
Received Sep 28, 2017
Revised Jan 1, 2018
Accepted Aug 1, 2018

Keyword:
Adiabtic switching principle
Two phase clocked adiabatic Static CMOS
Conventional CMOS
Energy recovery
Low power

ABSTRACT
This paper presents a design and implementation of 2*2 array and 4*4 array multiplier using proposed Two Phase Clocked Adiabatic Static CMOS logic (2PASCL) circuit. The proposed 2PASCL circuit is based on adiabatic energy recovery principle which consumes less power. The proposed 2PASCL uses two sinusoidal power clocks which are 180° phase shifted with each other. The measurement result of 2*2 array proposed 2PASCL multiplier gives 80.16 % and 97.67 %power reduction relative to reported 2PASCL and conventional CMOS logic and the measurement result of 4*4 array proposed 2PASCL multiplier demonstrate 32.88 % and 82.02 %power reduction compared to reported 2PASCL and conventional CMOS logic. Another advantage of the proposed circuit is that it gives less power though the number of transistors in proposed and reported 2PASCL circuit is same. From the result we conclude that proposed 2PASCL technology is advantageous to application in low power digital systems, pacemakers and sensors. The circuits are simulated at 180nm technology mode.

Corresponding Author:
Minal Keote,
Department of Electronics and Telecommunication Engineering,
Yeshwantrao Chavan College of Engineering, Wanadongri, Hingna,
Maharashtra, India.
Email: klminal@rediffmail.com

1. INTRODUCTION
Power reduction has now become a key concern for the design of portable and wireless devices. In the design of digital circuits which uses complementary metal oxide semiconductor, the power reduction can be achieved by scaling down the transistor threshold voltage in accordance with supply voltage [1]. It will impose limitation as reducing the threshold voltage below a certain limit will increase leakage current. Switching event in CMOS circuits is another cause of power dissipation which can be decrease by reducing the switching activity [2]. Power dissipation in CMOS circuits during charging and discharging of node capacitances is given by the

$$E = C_L V_{DD}^2 \quad (1)$$

from the equation (1) it is apparent that energy can be reduced by reducing supply voltage and decreasing switching activity in CMOS circuits [3]-[4].
Various other techniques are available for low power reduction which includes reversible logic [5], GDI logic [6] and domino logic [7] at circuit level. But taking into account the limitations of power reduction in CMOS circuits. A novel approach called Adiabatic computing have been proposed [8],[9] which uses the

International Conference on VLSI, Communication & Instrumentation (ICVCI) 2011
Proceedings published by International Journal of Computer Applications® (IJCA)

Comparative Analysis of 4-bit CMOS Multipliers

Navdeep Goel
Assistant Professor (ECE)
YCOE, Punjabi University Kashi Campus,
Talwandi Sabo (India)

Lalit Garg
Lecturer (ECE)
Guru Gobind Singh College of Engineering & Technology,
Talwandi Sabo (India)

ABSTRACT
A fast and energy-efficient multiplier is always needed in electronics industry especially digital signal processing (DSP), image processing and arithmetic units in microprocessors. Multiplier is such an important element which contributes substantially to the total power consumption of the system. Multipliers of various bit-widths are frequently required in VLSI from processors to application specific integrated circuits (ASICs). Recently reported logic style comparisons based on full-adder circuits claimed complementary pass transistor logic (CPL) to be much more power-efficient than complementary CMOS. However, new comparisons performed on more efficient CMOS circuit realizations and a wider range of different logic cells, as well as the use of realistic circuit arrangements demonstrate CMOS to be superior to CPL in most cases with respect to speed, area, power dissipation, and power-delay products. The most important and widely accepted metrics for measuring the quality of multiplier designs propagation delay, power dissipation and area. This paper describes the comparative performance of 4-bit multipliers designed using TANNER EDA, using different logic design styles.

Keywords: Multiplier, CMOS Logic Design Style.

1. INTRODUCTION
The increasing demand for low-power very large scale integration (VLSI) can be addressed at different design levels, such as the architectural, circuit, layout, and the process technology level [1]. At the circuit design level, considerable potential for power savings exists by means of proper choice of a logic style for implementing combinational circuits. This is because all the important parameters governing power dissipation—switching capacitance, transition activity, and short-circuit currents—are strongly influenced by the chosen logic style. Depending on the application, the kind of circuit to be implemented, and the design technique used, different performance aspects become important. In the past, the parameters like high speed, small area and low cost were the major areas of concern, whereas power considerations are now gaining the attention of the scientific community associated with VLSI design. In recent years, the growth of personal computing devices (portable computers and real time audio and video based multimedia applications) and wireless communication systems has made power dissipation a most critical design parameter [1]. In the absence of low-power design techniques such applications generally suffer from very short battery life, while packaging and cooling them would be very difficult and this is leading to an unavoidable increase in the cost of the product. In multiplication, reliability is strongly affected by power consumption. Usually, high power dissipation implies high temperature operation, which, in turn, has a tendency to induce several failure mechanisms in the system.

Power dissipation is the most critical parameter for portability & mobility and it is classified in to dynamic and static power dissipation. Dynamic power dissipation occurs when the circuit is operational, while static power dissipation becomes an issue when the circuit is inactive or is in a power-down mode. There are three major sources of power dissipation in digital CMOS circuits, which are summarized in equation (1) [2].

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{leakage} = \left(\alpha_{0\to 1} \times C_L \times V_{dd}^2 \times f_{clk}\right) + \left(I_{sc} \times V_{dd}\right) + \left(I_{leakage} \times V_{dd}\right) \quad (1)$$

The first term represents the switching component of power, where C_L is the load capacitance, f_{clk} is the clock frequency and α is the probability that a power consuming transition occurs (the activity factor). The second term is due to the direct-path short circuit current, I_{sc}, which arises when both the NMOS and PMOS transistors are simultaneously active, conducting current directly from supply to ground. Finally, leakage current, $I_{leakage}$ which can arise from substrate injection and sub-threshold effects, is primarily determined by fabrication technology considerations.

The switching power dissipation in CMOS digital integrated circuits is a strong function of the power supply voltage. Therefore, reduction of V_{dd} emerges as a very effective means of limiting the power consumption. However, the saving in power dissipation comes at a significant cost in terms of increased circuit delay. Since the exact analysis of propagation delay is quite complex, a simple first order derivation [3] can be used to show the relation between power supply and delay time

$$T_d \propto \frac{C_L V_{DD}}{K(V_{DD} - V_{TH})^{\alpha}} \quad (2)$$

K - Transistor's aspect ratio (W /L)
V_{TH} - Transistor threshold voltage
α - Velocity saturation index which varies between 1 and 2

Unfortunately, reducing the supply voltage reduces power, but when the supply voltage is near to threshold voltage (from equation 2), the delay increases drastically [4].

Section II gives a short introduction to the most important existing static logic styles and compares them qualitatively. Section III gives the two important multiplier architectures, designed in this paper. Results of quantitative comparisons based on simulations

THANK YOU

By: KEVAL BHAVSAR