Optimizing Chip Performance: The Role of MAC Arrays in Semiconductor Design

Enhancing Computational Efficiency: The Power of Multiply-Accumulate Arrays

In chip and semiconductor manufacturing, MAC (Multiply-Accumulate) arrays in RTL (Register Transfer Level) are crucial components, especially in the context of processing high-speed mathematical computations, which are fundamental in fields such as AI/ML, signal processing, and more.

Importance of MAC Arrays in Chip Design:

1. Efficient Computational Performance:
    - MAC arrays are often used in operations involving large datasets, such as matrix multiplications, convolutions in neural networks, and digital signal processing (DSP). These operations require multiplying numbers and then adding the results, which MAC arrays handle very efficiently.
    - Having a specialized array of MAC units significantly improves the speed and performance of the chip for tasks that require frequent multiplication and accumulation, such as in AI/ML inference and training.
2. Parallelism:
    - A MAC array in RTL allows for parallel execution of multiple multiply-accumulate operations. This parallelism boosts throughput and reduces latency, which is essential for applications like real-time data processing and complex simulations.

- The design of MAC arrays is optimized to perform multiple operations simultaneously, making them a key building block for high-performance chips in areas such as image processing, radar systems, and cryptocurrency mining.

3. Energy Efficiency:

   - MAC operations, being computationally intensive, are often implemented in hardware for energy efficiency. By creating a dedicated MAC array, the system can handle numerous operations per clock cycle, reducing power consumption when compared to software-based solutions.

   - Optimizing these arrays in RTL allows designers to reduce the chip's power requirements, a key consideration for mobile and embedded devices.

4. Customizable in VLSI Design:

   - In custom silicon solutions, like those from your startup Yarok14, designing efficient MAC arrays in RTL allows tailoring the hardware to specific applications, whether it's for low-power IoT devices or high-performance AI accelerators.

   - Custom MAC array implementations in VLSI can be highly optimized for size, speed, and power, providing a significant advantage over general-purpose processors.

5. Scalability:

   - As semiconductor designs become more complex, the scalability of MAC arrays ensures that the chip can handle increasingly large datasets. RTL design allows these arrays to be scaled efficiently, supporting larger AI models, bigger signal-processing tasks, and more sophisticated computational workloads.

6. Integration with AI/ML Acceleration:

    - Since AI/ML workloads often involve repeated matrix multiplications and additions (e.g., in deep learning models), MAC arrays in RTL provide the foundation for dedicated hardware accelerators, such as Tensor Processing Units (TPUs) or custom AI chips.

    - In your case, integrating MAC arrays in the design of custom silicon chips would significantly optimize the performance of the Biomethane Optimization System or any other AI-driven product you are developing.

7. Cost-Effectiveness in Production:

    - Efficient RTL designs allow for cost-effective manufacturing of MAC arrays at scale. Using standard design libraries and toolchains for MAC array design can reduce both design time and costs in semiconductor production, making it easier to bring custom products to market.

In Summary:

In semiconductor manufacturing, **MAC arrays in RTL** are essential for optimizing mathematical computation efficiency, energy use, and overall performance. By allowing multiple operations to happen in parallel and providing a way to tailor hardware to specific applications, MAC arrays play a key role in creating high-performance chips, especially for AI, signal processing, and similar domains.

Usually We entirely possible to design an **N×N MAC array RTL block** using **Verilog** (rather than SystemVerilog, which is an extension of Verilog with additional features). Verilog provides the essential constructs to describe hardware at the RTL level, and you can implement the multiply-accumulate operations effectively using the language's basic features. However, SystemVerilog does offer additional constructs and features that might make the design process easier or more efficient, but Verilog alone can be fully sufficient for most tasks.

Verilog Code for an N×N MAC Array

Here's a simple way to implement an N×N MAC array RTL block using Verilog:

1. MAC Unit Module (1×1)

This module performs a basic **Multiply-Accumulate (MAC)** operation for a single input.

Verilog

```
module mac_unit(
    input [15:0] A, // First input operand (16-bit)
    input [15:0] B, // Second input operand (16-bit)
    input [31:0] C, // Accumulator (32-bit)
    output reg [31:0] result // Result of (A * B + C)
);
    always @(*) begin
        result = (A * B) + C; // Multiply and accumulate
    end
endmodule
```

2. N×N MAC Array Module

Now, we can expand this to an **N×N MAC array**, where each MAC unit will perform a multiply-accumulate operation. The MAC array will process multiple inputs in parallel.

Verilog

```
module mac_array #(
    parameter N = 4 // Size of the array (4x4 as default, but can be
any size)
)(
    input [15:0] A [0:N-1], // First input array (N elements)
    input [15:0] B [0:N-1], // Second input array (N elements)
    input [31:0] C [0:N-1], // Accumulator array (N elements)
    output [31:0] result [0:N-1] // Result array (N elements)
);
    genvar i; // Generate variable for loop iteration

    // Generate N MAC units in parallel
    generate
```

```
        for (i = 0; i < N; i = i + 1) begin : mac_unit_gen
            mac_unit mac_inst (
                .A(A[i]),
                .B(B[i]),
                .C(C[i]),
                .result(result[i])
            );
        end
    endgenerate
endmodule
```

Explanation of the Code:

1. `mac_unit` **Module**:
    - This is a single multiply-accumulate unit. It takes two operands (A and B), performs the multiplication, and adds the result to an accumulator (C).
    - It outputs the accumulated result as `result`.
2. `mac_array` **Module**:
    - This is the N×N MAC array. The parameter `N` allows you to specify the size of the array (defaulted to 4x4, but it can be set to any size).
    - It uses a **generate loop** to instantiate multiple MAC units (one for each index of the array). This loop iterates over the range of N and generates an individual MAC unit for each one.
    - The inputs `A`, `B`, and `C` are all arrays of size N, meaning the array will have N separate inputs, each corresponding to one MAC unit.
    - The `result` output is also an array of N elements, where each element holds the result from each MAC unit.

Key Points:

- **Array Inputs and Outputs**: The inputs `A`, `B`, and `C` are all arrays, which means that for an N×N MAC array, you'll be passing N elements for each input. Similarly, the `result` is an array that collects the N results.

- **Generate Loop**: The `generate` block in Verilog allows for creating multiple instances of the `mac_unit` block. In this case, we instantiate N MAC units in parallel, which performs the multiply-accumulate operation for each pair of inputs from `A` and `B`.

- **Verilog RTL Design**: This design uses **combinational logic** (`always @(*)`) for the multiplication and accumulation operation. The `result` for each MAC unit is computed as the product of `A[i] * B[i]` added to the corresponding accumulator `C[i]`.

Benefits of Using Verilog:

- **Simplicity**: Verilog is simpler and has fewer features than SystemVerilog, which might be beneficial for straightforward designs where you don't need advanced features like classes or assertions.

- **Compatibility**: Verilog is widely supported across all synthesis and simulation tools, making it a good choice for portable designs.

- **Readability**: The Verilog code above is relatively easy to understand and implement for anyone familiar with RTL design.

Conclusion:

We can definitely design an N×N MAC array RTL block using Verilog. While **SystemVerilog** offers additional capabilities for verification, design, and abstraction, **Verilog** alone is more than capable of implementing an N×N MAC array in a simple and effective manner. The example provided shows how you can use **generate loops** in Verilog to efficiently instantiate and parallelize multiple MAC units.

✅ Modifications from SystemVerilog to Verilog:

SystemVerilog offers more advanced data structures and syntax that are not supported in standard Verilog. Here are the key modifications made in the transition:

1. Array Port Declarations:

SystemVerilog Syntax:

systemverilog

```
input wire [7:0] a[7:0],
input wire [7:0] b[7:0],
output reg [15:0] mac_out[7:0]
```

-
    - This uses a 2D array of 8-bit inputs and 16-bit outputs.

Verilog Modification:
Verilog does **not** support array ports directly, so we must flatten the arrays into individual ports:

Verilog

```
input [7:0] a0, a1, a2, a3, a4, a5, a6, a7,
input [7:0] b0, b1, b2, b3, b4, b5, b6, b7,
output reg [15:0] mac_out0, mac_out1, mac_out2, mac_out3,
output reg [15:0] mac_out4, mac_out5, mac_out6, mac_out7
```

2. Array Access in Testbench:

SystemVerilog Syntax:

systemverilog

```
a[0] = 8'd10;
b[0] = 8'd3;
```

- This is valid in SystemVerilog but not in Verilog.

Verilog Modification:
Since the ports are now individual signals, we access them directly:

verilog

```
a0 = 8'd10;
b0 = 8'd3;
```

3. Dumping Signals to VCD:

- No change was necessary for this, as both Verilog and SystemVerilog use the same `$dumpfile` and `$dumpvars` syntax.

```
initial begin
    $dumpfile("mac_array_8x8_tb.vcd");
    $dumpvars(0, mac_array_8x8_tb);
end
```

4. Module Instantiation:

SystemVerilog:
Instantiation would use array-based port connections:

systemverilog

```
.a(a),
.b(b),
.mac_out(mac_out)
```

- **Verilog Modification:**
  Since we flattened the ports, we need to connect each signal individually:

verilog

```
.a0(a0), .a1(a1), .a2(a2), .a3(a3), .a4(a4), .a5(a5), .a6(a6),
.a7(a7),

.b0(b0), .b1(b1), .b2(b2), .b3(b3), .b4(b4), .b5(b5), .b6(b6),
.b7(b7),
.mac_out0(mac_out0), .mac_out1(mac_out1), .mac_out2(mac_out2),
.mac_out3(mac_out3),
.mac_out4(mac_out4), .mac_out5(mac_out5), .mac_out6(mac_out6),
.mac_out7(mac_out7)
```

5. Data Types:

- **SystemVerilog:** Allows `logic` and `wire` to be declared as arrays and structs.
- **Verilog:** Only supports simple `wire`, `reg`, etc., without array-based declarations.

✅ Summary of the Conversion:

- Replaced array ports with individual ports.
- Modified the testbench to handle individual signals.
- Adjusted signal connections during module instantiation.
- No structural changes in the core logic — only syntactical adjustments.

In the NVIDIA GH100 Streaming Multiprocessor (SM) architecture shown in the image, the **MAC arrays** are primarily located within the **Tensor Cores**.

✅ Where are the MAC Arrays?

1. **Tensor Cores (4th Generation):**
    - The Tensor Cores are specialized processing units designed specifically for matrix operations, including **matrix multiplication and accumulation (MAC)**.
    - Each Tensor Core can perform multiple MAC operations in parallel, making it highly efficient for deep learning workloads, AI inference, and matrix computations.
    - Tensor Cores are optimized for mixed-precision operations (e.g., FP16, FP32, INT8) to maximize throughput while maintaining accuracy.
2. **ALU Units (INT32, FP32, FP64):**
    - The general-purpose arithmetic units (INT32, FP32, FP64) can also perform basic MAC operations. However, they are less specialized than Tensor Cores and are used for more general computations rather than dense matrix operations.

✅ Why Tensor Cores for MAC?

- Tensor Cores are specifically optimized to handle large-scale matrix multiplications, which are essentially dense MAC operations.

- They leverage **SIMD (Single Instruction, Multiple Data)** and **SIMT (Single Instruction, Multiple Threads)** architectures to perform multiple MAC operations in parallel, significantly boosting throughput for AI and HPC tasks.

The expression **(A × B) + C** represents the **Multiply-Accumulate (MAC)** operation, which is a fundamental computational building block commonly used in many digital signal processing (DSP), machine learning (ML), and hardware acceleration tasks.

To break it down:

- **A** and **B**: These are the two operands that will be multiplied.
- **C**: This is the **accumulator**, a register or memory location that stores the ongoing result of previous computations.
- **(A × B) + C**: This is the MAC operation. It computes the product of **A** and **B**, then adds the result to the accumulator **C**.

Understanding the Process:

1. **Multiplication**: The MAC unit multiplies two values, **A** and **B**.
2. **Accumulation**: The result of the multiplication is added to the value stored in the accumulator **C**. This means that after each operation, **C** is updated with the new sum.

This operation is **repeated** many times for different sets of inputs (values for **A**, **B**, and **C**), often in parallel, making the MAC unit very efficient for certain types of algorithms, such as:

- **Neural networks**: Where it computes the weighted sum of inputs.
- **Signal processing**: For filtering, convolution, etc.
- **Cryptography**: In specific algorithms that require heavy multiplication and accumulation.

Example in the Context of Hardware (MAC Unit):

In a hardware context, **each MAC unit** is designed to perform this operation (A × B) + C for a given set of input values. For example:

- In a **neural network** accelerator, A might be the input data (e.g., pixel values of an image), B might be the weights (e.g., parameters learned by the model), and C could be the current accumulated sum of previous operations (e.g., previous layers' outputs or a

bias term).

Working in Parallel:

When multiple MAC units are arranged in a **MAC array**, the computations can be performed **in parallel**:

- **A1 × B1 + C1**, **A2 × B2 + C2**, ..., **An × Bn + Cn** are computed simultaneously, greatly increasing throughput and speeding up the computation.

In such arrays, **each MAC unit** operates independently, allowing for simultaneous processing of multiple data points, which is extremely useful in fields like AI, machine learning, and signal processing where large amounts of data are processed concurrently.

Summary:

- **MAC Unit**: A hardware block that performs **multiplication** and **accumulation** of the form **(A × B) + C**.

- **Parallel MAC Array**: An array of such units working in parallel to process multiple sets of inputs simultaneously.