Systolic Array Matrix – Matrix Multiplication

RTL Design using verilog

Created By: Ahmed Mostafa Gomaa

Table of Contents

1. Introduction

Matrix-matrix multiplication is a fundamental operation in numerous scientific, engineering, and machine learning applications. As the scale of data and computational demands grow, traditional sequential processing methods often fall short in terms of performance and efficiency. To address this challenge, systolic arrays have emerged as a powerful architectural paradigm for accelerating matrix-matrix multiplication, particularly in hardware implementations such as GPUs, TPUs, and FPGAs.

A systolic array is a specialized, highly parallel computing structure composed of a grid of interconnected processing elements (PEs). Each PE is a simple computational unit designed to perform a specific task, such as multiplying and accumulating values. The array operates in a synchronized, pipelined manner, where data flows rhythmically (like a "systolic" heartbeat) between neighboring PEs, enabling efficient data reuse and minimizing memory bandwidth requirements.

In the context of matrix-matrix multiplication, the systolic array is particularly well-suited due to its ability to exploit the inherent parallelism of the operation. Each PE is responsible for computing a portion of the result matrix, and the data flows through the array in a carefully orchestrated manner, ensuring that each PE receives the necessary input values at the right time. This approach not only maximizes computational throughput but also reduces the need for frequent data transfers to and from external memory, which is often a bottleneck in traditional architectures.

The process element (PE) is the core building block of a systolic array. It typically consists of a multiplier, an adder, and a small amount of local storage (e.g., registers). The PE performs multiply-accumulate (MAC) operations, which are the cornerstone of matrix multiplication. By organizing PEs in a two-dimensional grid and carefully controlling the flow of data, systolic arrays can achieve high computational density and energy efficiency, making them ideal for modern high-performance computing tasks.

2. Systolic Array Matrix Multiplication

2.1. Introduction

Matrix-matrix multiplication is a critical operation in many computational domains, including scientific computing, machine learning, computer graphics, and signal processing. As the size of matrices grows and the demand for faster computations increases, traditional CPU-based approaches often struggle to meet performance and efficiency requirements. Systolic arrays have emerged as a highly efficient hardware architecture for accelerating matrix-matrix multiplication, particularly in specialized hardware like GPUs, TPUs (Tensor Processing Units), and FPGAs.

2.2. What is a Systolic Array?

A systolic array is a network of interconnected processing elements (PEs) arranged in a grid-like structure. Each PE is a simple computational unit that performs basic operations, such as multiply-accumulate (MAC). The array operates in a synchronized, pipelined fashion, where data flows rhythmically between neighboring PEs, much like the systolic pumping of blood in the human body. This dataflow architecture minimizes memory access overhead and maximizes parallelism, making it ideal for compute-intensive tasks like matrix-matrix multiplication.

2.3. How Systolic Arrays Perform Matrix-Matrix Multiplication

Consider two matrices **A** (of size M×K) and **B** (of size K×N) being multiplied to produce a result matrix **C** (of size M×N). In a systolic array:

1. **Dataflow Organization**:
   - The matrices **A** and **B** are fed into the systolic array in a carefully orchestrated manner.
   - Each PE in the array is responsible for computing a partial result of one element in the output matrix **C**.
   - The rows of matrix **A** flow horizontally through the array, while the columns of matrix **B** flow vertically.
2. **Processing Element (PE) Operation**:
   - Each PE performs a MAC operation: it multiplies an element of **A** with an element of **B** and adds the result to a locally stored partial sum.
   - The partial sums are accumulated over multiple cycles until the final result for each element of **C** is computed.
3. **Pipelining and Parallelism**:
   - The systolic array exploits both spatial and temporal parallelism. Multiple PEs work simultaneously on different parts of the matrices, and data flows through the array in a pipelined manner, ensuring high utilization of computational resources.
4. **Data Reuse**:
   - One of the key advantages of systolic arrays is their ability to reuse data locally within the array. Once data is loaded into a PE, it can be used multiple times without needing to fetch it again from external memory, significantly reducing memory bandwidth requirements.

2.4. Why Use Systolic Arrays for Matrix-Matrix Multiplication

1. **High Performance**:
   - Systolic arrays achieve high computational throughput by leveraging massive parallelism. The regular structure of matrix-matrix multiplication maps naturally to the grid-like organization of PEs, enabling efficient computation.
2. **Energy Efficiency**:

- By minimizing data movement and reusing data locally, systolic arrays reduce power consumption compared to traditional architectures that rely heavily on frequent memory accesses.

3. **Scalability**:
   - Systolic arrays can be scaled up by adding more PEs, making them suitable for handling larger matrices and more complex computations.
4. **Low Latency**:
   - The pipelined nature of systolic arrays ensures that data is processed with minimal delay, making them ideal for real-time applications.

2.5. Performance of Systolic Arrays

- **Throughput**: Systolic arrays can achieve near-peak performance for matrix-matrix multiplication, often reaching close to the theoretical maximum number of operations per second (e.g., teraflops or petaflops in modern hardware).
- **Latency**: The pipelined design ensures low latency, as data flows continuously through the array without stalling.
- **Memory Bandwidth**: Systolic arrays significantly reduce the need for external memory access, as data is reused locally within the array. This is particularly beneficial for large-scale computations where memory bandwidth is a bottleneck.

2.6. Power Consumption

Systolic arrays are designed to be energy-efficient for several reasons:

1. **Data Reuse**: By reusing data within the array, they minimize the energy-intensive process of fetching data from external memory.
2. **Localized Computation**: Each PE performs simple operations (e.g., MAC) with minimal control overhead, reducing dynamic power consumption.
3. **Regular Structure**: The predictable and regular dataflow reduces the need for complex control logic, further lowering power consumption.

Compared to general-purpose CPUs and even GPUs, systolic arrays can achieve significantly higher performance per watt, making them ideal for power-constrained environments like mobile devices and data centers.

2.7. Applications of Systolic Arrays

1. **Machine Learning and Deep Learning**:
   - Systolic arrays are widely used in accelerators like Google's TPUs for training and inference of deep neural networks, where matrix multiplications are a core operation.
2. **Scientific Computing**:
   - Applications such as finite element analysis, fluid dynamics, and quantum simulations often involve large-scale matrix operations that benefit from the parallelism of systolic arrays.
3. **Signal Processing**:

    - Tasks like Fourier transforms, filtering, and beamforming rely on matrix operations that can be accelerated using systolic arrays.
4. **Computer Graphics**:
    - Rendering and image processing algorithms often involve matrix transformations, which can be efficiently computed using systolic arrays.
5. **Cryptography**:
    - Cryptographic algorithms, such as those based on linear algebra, can be accelerated using systolic arrays.
6. **High-Performance Computing (HPC)**:
    - Systolic arrays are increasingly being integrated into HPC systems to accelerate large-scale simulations and data analysis tasks.

3. Processing Element (PE)

3.1. Introduction

The **Processing Element (PE)** is the fundamental building block of a systolic array. It is a simple, modular computational unit designed to perform specific tasks, such as multiply-accumulate (MAC) operations, which are critical for matrix-matrix multiplication and other linear algebra operations. The efficiency and performance of a systolic array largely depend on the design and functionality of its PEs. Below is a detailed exploration of PEs, including their structure, functionality, role in systolic arrays, and design considerations.

3.2. Structure of a Processing Element (PE)

A typical PE in a systolic array consists of the following components:

1. **Multiplier**:
    - Performs the multiplication of two input values (e.g., elements from matrices **A** and **B**).
    - The multiplier is often optimized for low latency and high throughput.
2. **Adder/Accumulator**:
    - Adds the result of the multiplication to a running partial sum.
    - The accumulator stores intermediate results, which are updated iteratively during the computation.
3. **Local Storage (Registers)**:
    - Small, fast memory units (registers) are used to store input values, intermediate results, and partial sums.
    - Local storage minimizes the need to access external memory, reducing latency and power consumption.
4. **Control Logic**:
    - Manages the flow of data within the PE and coordinates with neighboring PEs.
    - Ensures that the PE operates in sync with the rest of the systolic array.
5. **Input/Output Interfaces**:
    - Connects the PE to its neighboring PEs in the systolic array.
    - Facilitates the flow of data into and out of the PE.

3.3. Functionality of a PE

The primary function of a PE is to perform **multiply-accumulate (MAC)** operations, which are the core of matrix-matrix multiplication. Here's how a PE operates:

1. **Data Input**:
   - The PE receives input values from its neighboring PEs or from external memory.
   - For matrix-matrix multiplication, one input might be an element from matrix **A**, and the other might be an element from matrix **B**.
2. **Multiplication**:
   - The multiplier computes the product of the two input values.
3. **Accumulation**:
   - The adder adds the product to a partial sum stored in the accumulator.
   - This partial sum represents the intermediate result for a specific element of the output matrix **C**.
4. **Data Output**:
   - The updated partial sum is stored locally, and the input values are passed to neighboring PEs for further processing.
5. **Iteration**:
   - The PE repeats the above steps for multiple cycles until the final result for its assigned portion of the computation is computed.

3.4. Role of PEs in Systolic Arrays

In a systolic array, PEs are organized in a grid-like structure, and each PE is responsible for computing a portion of the overall result. The key roles of PEs in a systolic array include:

1. **Parallel Computation**:
   - Each PE works independently on its portion of the computation, enabling massive parallelism.
2. **Data Reuse**:
   - PEs reuse data locally, reducing the need to fetch data from external memory repeatedly.
3. **Pipelined Dataflow**:
   - PEs operate in a pipelined manner, ensuring that data flows smoothly through the array without bottlenecks.
4. **Modularity**:
   - The simple and modular design of PEs allows systolic arrays to be scaled up or down easily by adding or removing PEs.

3.5. Design Considerations for PEs

When designing PEs for systolic arrays, several factors must be considered to optimize performance, power consumption, and area efficiency:

1. **Precision**:
   - The precision of the multiplier and adder (e.g., 8-bit, 16-bit, 32-bit) must be chosen based on the application requirements.
   - Lower precision can improve performance and reduce power consumption but may affect accuracy.

2. **Local Storage Size**:
    - The size of the local registers must be sufficient to store intermediate results and partial sums without requiring frequent access to external memory.
3. **Interconnect Design**:
    - The input/output interfaces must be designed to ensure efficient data transfer between PEs with minimal latency.
4. **Power Efficiency**:
    - PEs should be optimized for low power consumption, especially for applications in mobile devices or data centers.
5. **Scalability**:
    - The design of PEs should allow for easy scaling to larger arrays without significant overhead.
6. **Flexibility**:
    - Some PEs are designed to support multiple operations (e.g., addition, subtraction, or logical operations) to increase versatility.

3.6. Advantages of PEs in Systolic Arrays

1. **High Throughput**:
    - PEs enable massive parallelism, allowing systolic arrays to achieve high computational throughput.
2. **Low Latency**:
    - The pipelined operation of PEs ensures that data is processed with minimal delay.
3. **Energy Efficiency**:
    - By reusing data locally and minimizing memory access, PEs reduce power consumption.
4. **Scalability**:
    - The modular design of PEs allows systolic arrays to be scaled to handle larger problems.
5. **Simplicity**:
    - The simple and regular structure of PEs makes them easy to design and implement in hardware.

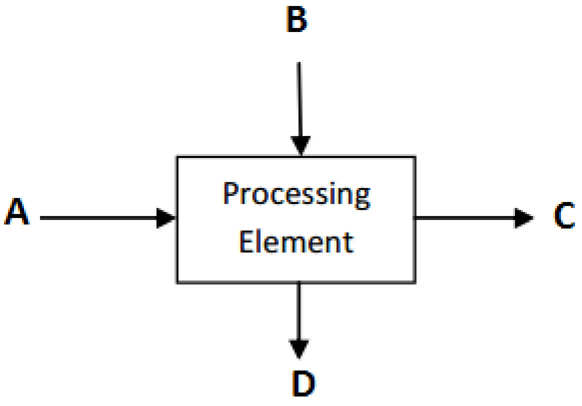
3.7. Applications of PEs

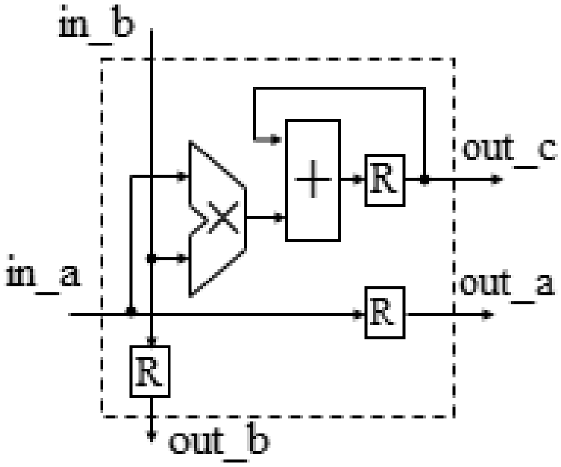
PEs are used in a wide range of applications where high-performance and energy-efficient computation is required:

1. **Machine Learning and Deep Learning**:
    - PEs are used in accelerators like TPUs for training and inference of neural networks.
2. **Scientific Computing**:
    - PEs accelerate matrix operations in simulations, numerical analysis, and linear algebra.
3. **Signal Processing**:
    - PEs are used in applications like FFT, filtering, and beamforming.
4. **Computer Graphics**:
    - PEs accelerate rendering and image processing tasks.

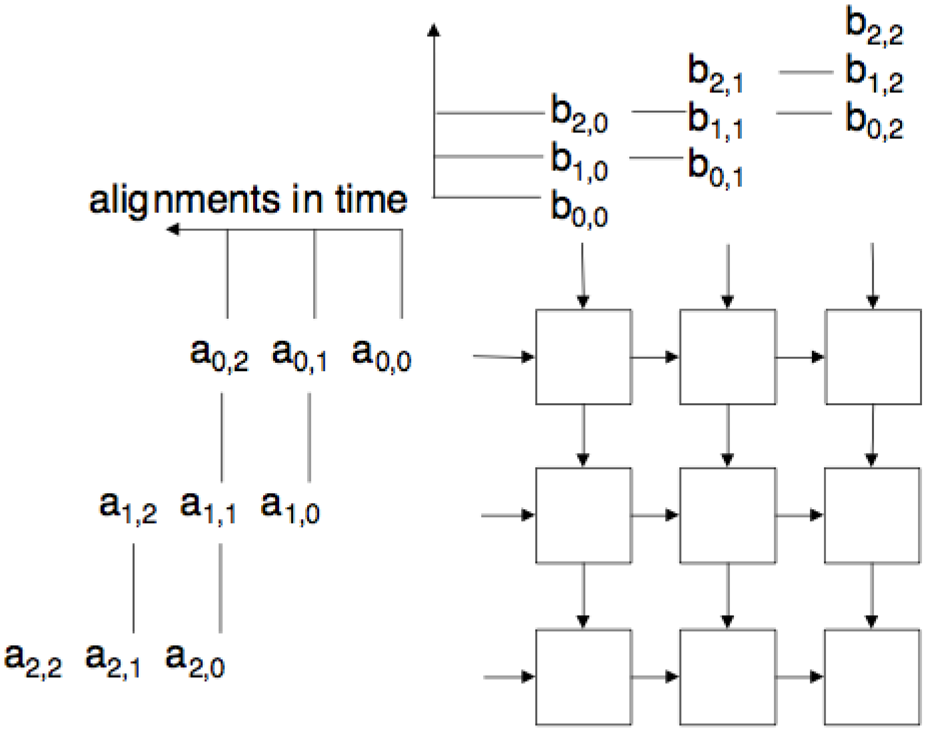
5. **Cryptography**:
   - PEs are used in cryptographic algorithms that involve matrix operations.

3.8. Structure View of PEs





3.9. View of PEs Array

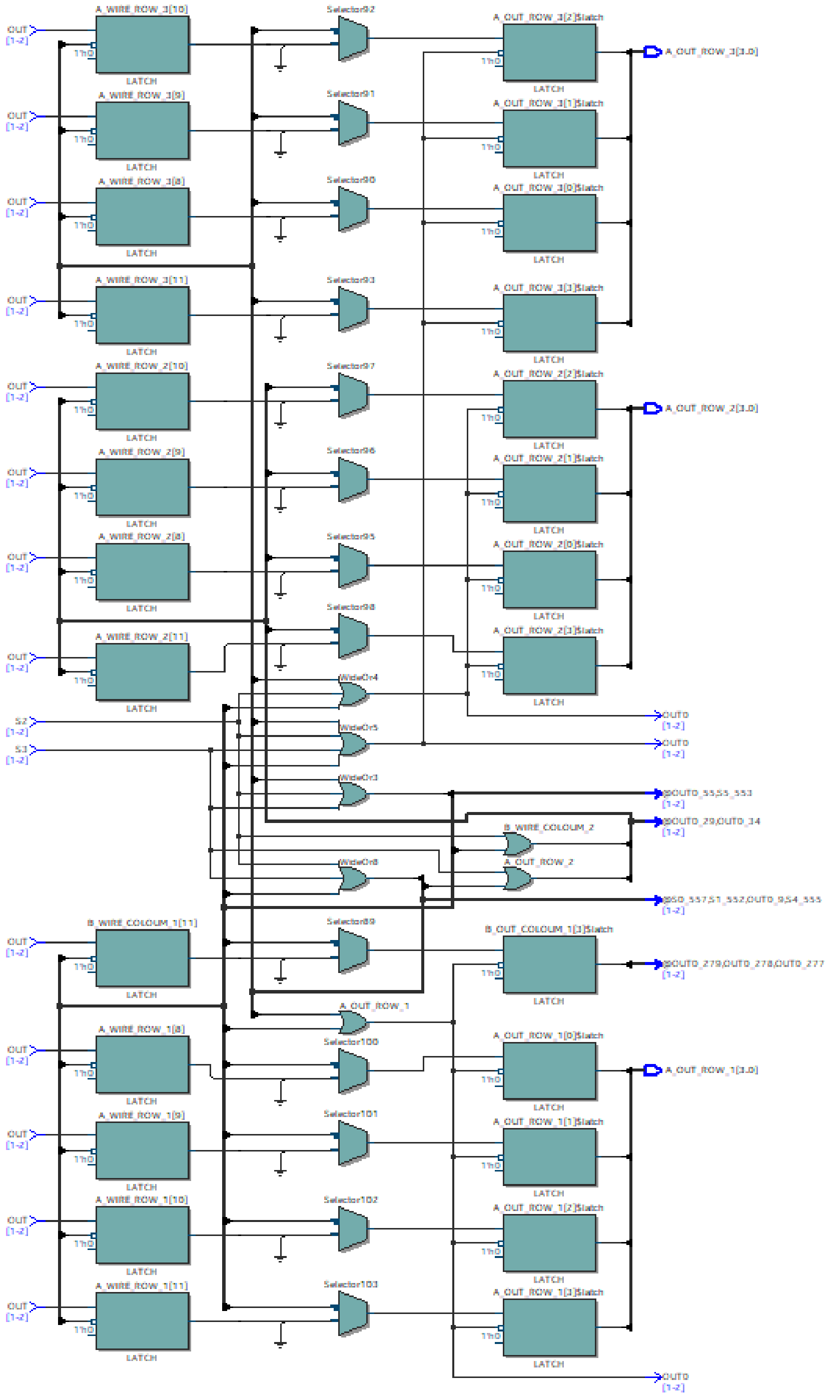


4. Systolic Array Multi. Structure

4.1. Components of Systolic Array

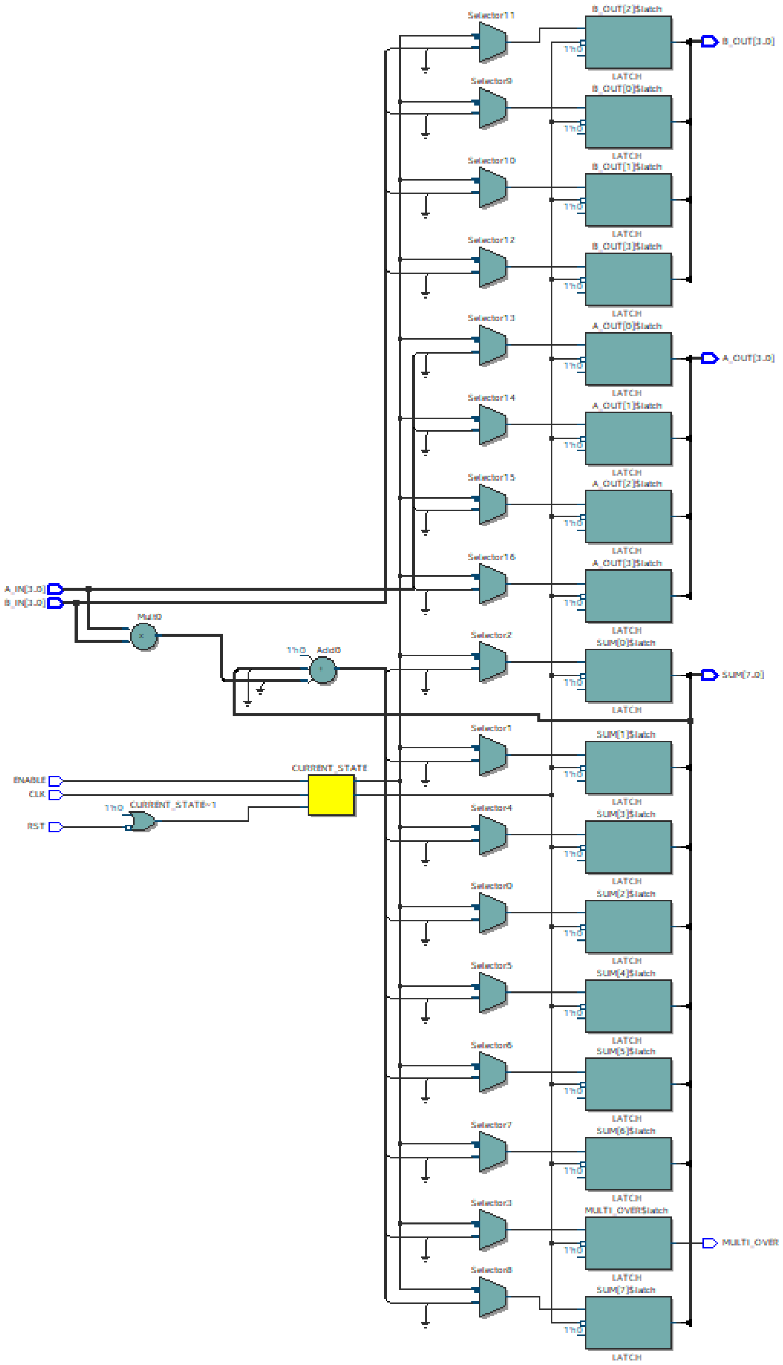
1. **Shift Register**
2. **Array of Processing Element (PEs)**

4.2. Shift Register RTL Viewer



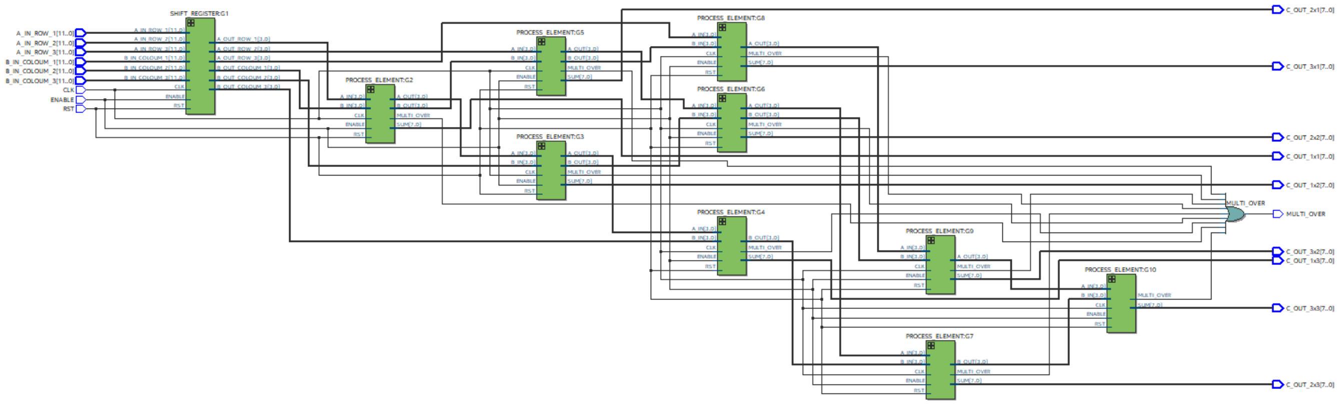
We use the shift register to handle the input of process Elements.

4.3. PEs RTL Viewer

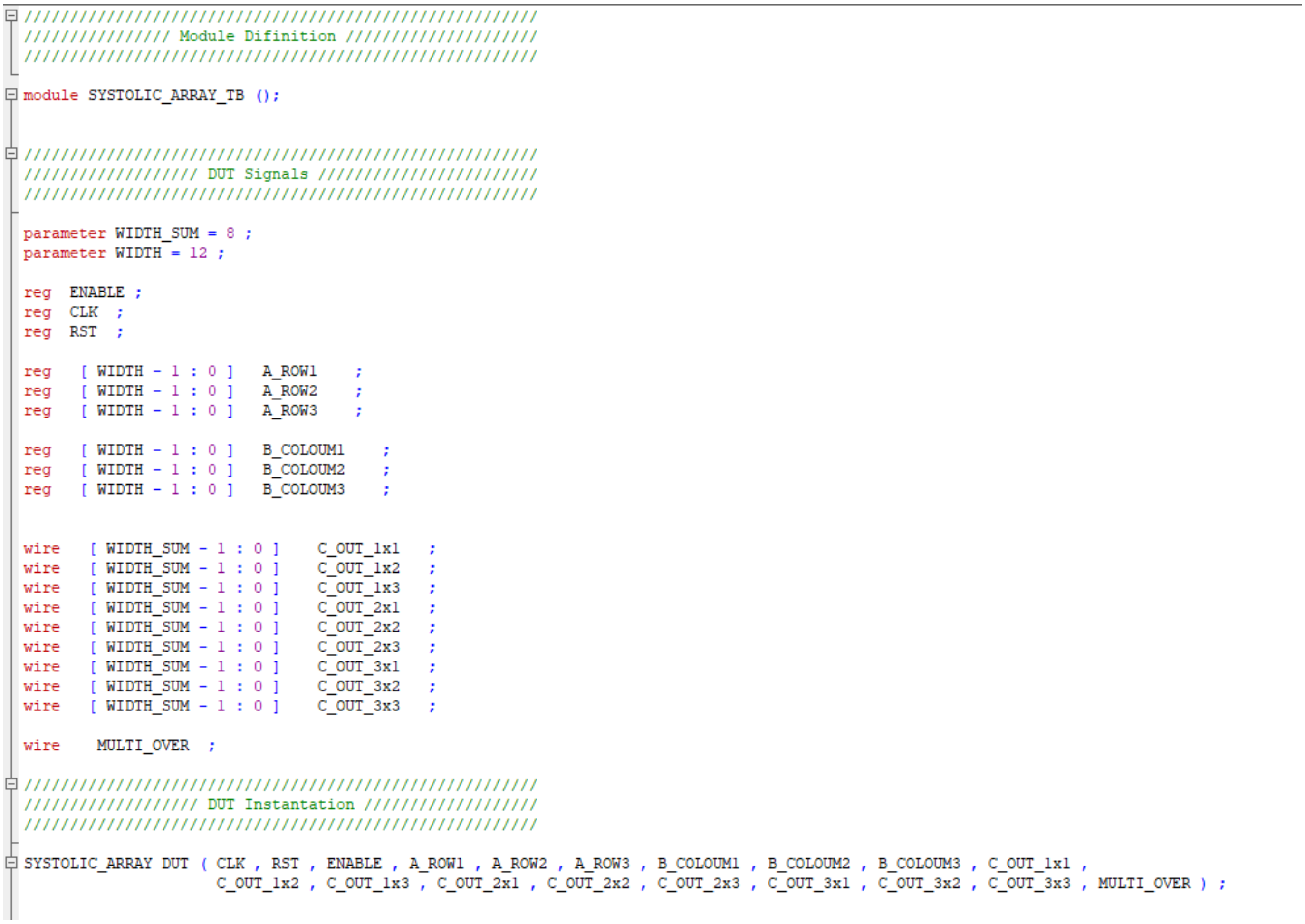


We multiply Matrix-Matrix 3x3 and each cell at the matrix consists of 4 bits.

4.4. Systolic Array RTL Viewer

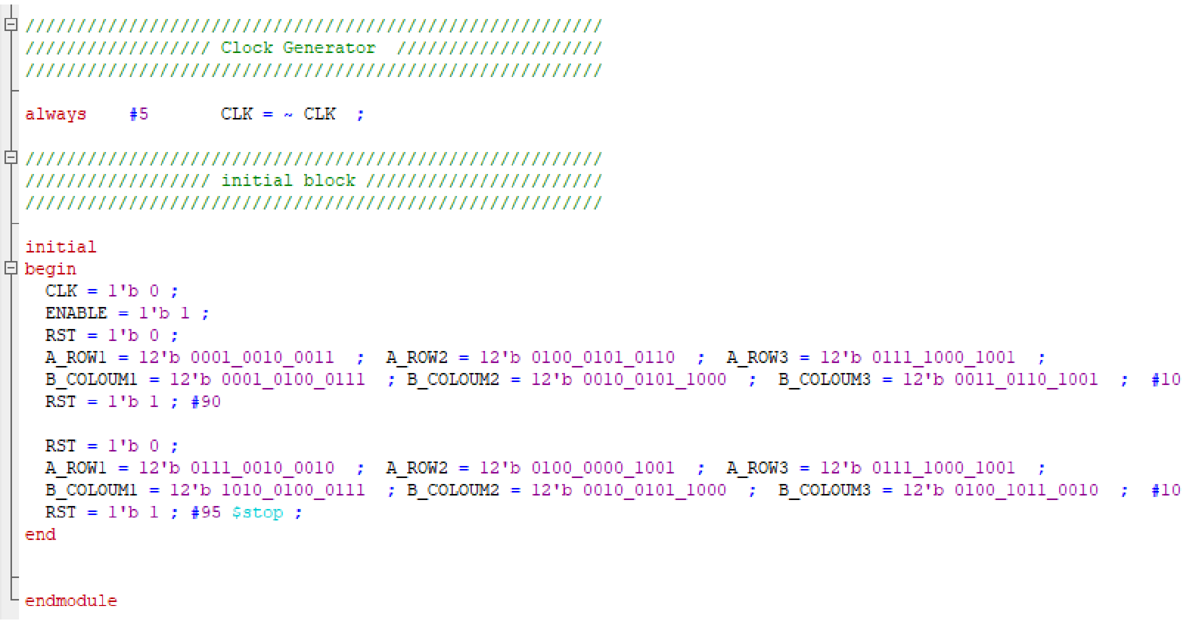


4.5. Testbench Code



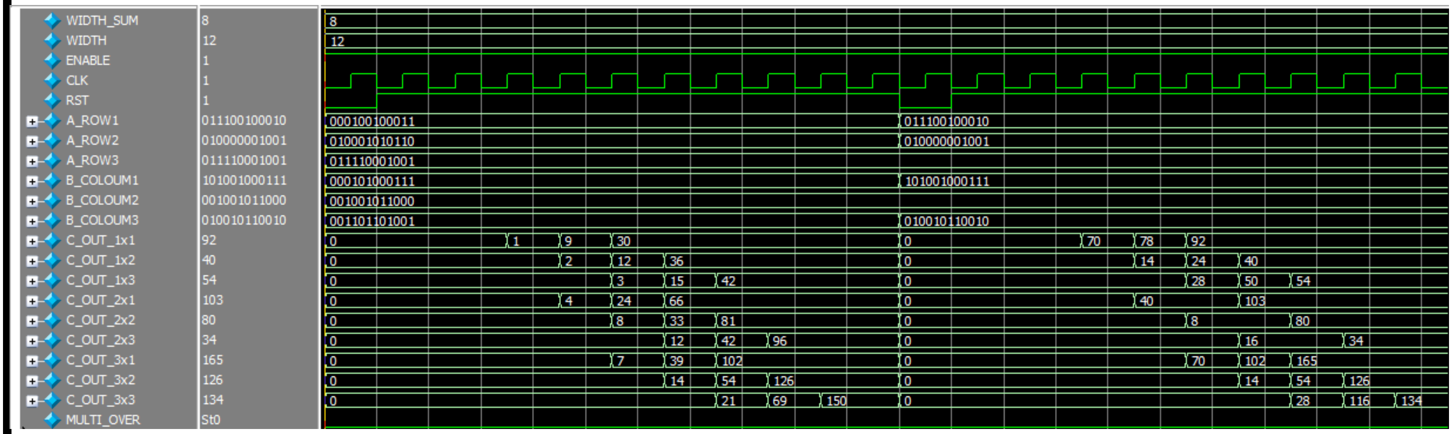
```verilog
//////////////////////////////////////////////////////////
///////////////// Module Difinition /////////////////////
//////////////////////////////////////////////////////////

module SYSTOLIC_ARRAY_TB ();


//////////////////////////////////////////////////////////
//////////////////// DUT Signals ////////////////////////
//////////////////////////////////////////////////////////

parameter WIDTH_SUM = 8 ;
parameter WIDTH = 12 ;

reg  ENABLE ;
reg  CLK   ;
reg  RST   ;

reg   [ WIDTH - 1 : 0 ]    A_ROW1    ;
reg   [ WIDTH - 1 : 0 ]    A_ROW2    ;
reg   [ WIDTH - 1 : 0 ]    A_ROW3    ;

reg   [ WIDTH - 1 : 0 ]    B_COLOUM1    ;
reg   [ WIDTH - 1 : 0 ]    B_COLOUM2    ;
reg   [ WIDTH - 1 : 0 ]    B_COLOUM3    ;


wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_1x1    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_1x2    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_1x3    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_2x1    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_2x2    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_2x3    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_3x1    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_3x2    ;
wire    [ WIDTH_SUM - 1 : 0 ]    C_OUT_3x3    ;

wire    MULTI_OVER  ;

//////////////////////////////////////////////////////////
//////////////////// DUT Instantation ///////////////////
//////////////////////////////////////////////////////////

SYSTOLIC_ARRAY DUT ( CLK , RST , ENABLE , A_ROW1 , A_ROW2 , A_ROW3 , B_COLOUM1 , B_COLOUM2 , B_COLOUM3 , C_OUT_1x1 ,
                     C_OUT_1x2 , C_OUT_1x3 , C_OUT_2x1 , C_OUT_2x2 , C_OUT_2x3 , C_OUT_3x1 , C_OUT_3x2 , C_OUT_3x3 , MULTI_OVER ) ;
```

•



```
/////////////////////////////////////////////////////////
////////////////// Clock Generator  ////////////////////
/////////////////////////////////////////////////////////

always    #5       CLK = ~ CLK  ;

/////////////////////////////////////////////////////////
///////////////// initial block /////////////////////
/////////////////////////////////////////////////////////

initial
begin
  CLK = 1'b 0 ;
  ENABLE = 1'b 1 ;
  RST = 1'b 0 ;
  A_ROW1 = 12'b 0001_0010_0011  ;  A_ROW2 = 12'b 0100_0101_0110  ;  A_ROW3 = 12'b 0111_1000_1001  ;
  B_COLOUM1 = 12'b 0001_0100_0111  ; B_COLOUM2 = 12'b 0010_0101_1000  ;  B_COLOUM3 = 12'b 0011_0110_1001  ;  #10
  RST = 1'b 1 ; #90

  RST = 1'b 0 ;
  A_ROW1 = 12'b 0111_0010_0010  ;  A_ROW2 = 12'b 0100_0000_1001  ;  A_ROW3 = 12'b 0111_1000_1001  ;
  B_COLOUM1 = 12'b 1010_0100_0111  ; B_COLOUM2 = 12'b 0010_0101_1000  ;  B_COLOUM3 = 12'b 0100_1011_0010  ;  #10
  RST = 1'b 1 ; #95 $stop ;
end


endmodule
```

4.6. Wave Form



WIDTH_SUM	8
WIDTH	12
ENABLE	1
CLK	1
RST	1
A_ROW1	011100100010
A_ROW2	010000001001
A_ROW3	011110001001
B_COLOUM1	101001000111
B_COLOUM2	001001011000
B_COLOUM3	010010110010
C_OUT_1x1	92
C_OUT_1x2	40
C_OUT_1x3	54
C_OUT_2x1	103
C_OUT_2x2	80
C_OUT_2x3	34
C_OUT_3x1	165
C_OUT_3x2	126
C_OUT_3x3	134
MULTI_OVER	St0