VELLORE INSTITUTE OF TECHNOLOGY

Vellore, Tamil Nadu - 632014

A Project Report on

"MATRIX MULTIPLICATION USING SYSTOLIC ARRAY ALGORITHM"

Submitted in Fulfilment of the Requirements for the Award of Degree

BACHELOR OF ENGINEERING
IN
ELECTRONICS AND COMMUNICATION ENGINEERING

SUBMITTED BY -

SHRUTI BHANDARI	21BEC0811
ANANYA VERMA	21BEC0868
SARTHAK VARSHNEY	21BEC0927
SHUBHAM SHARMA	21BEC0954

Under the Guidance of

Mr. Sunil Kumar Yadav
Scientist 'B', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka- 560093

Mr. Himanshu Singh
Scientist 'B', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka- 560093

Shri Paramananda Jena
Scientist 'G', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka- 560093

VELLORE INSTITUTE OF TECHNOLOGY

Vellore, Tamil Nadu - 632014

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

<u>CERTIFICATE</u>

Certified that the project entitled "MATRIX MULTIPLICATION USING SYSTOLIC ARRAY ALGORITHM", carried out by **SHRUTI BHANDARI (21BEC0811), ANANYA VERMA (21BEC0868), SARTHAK VARSHNEY (21BEC0927), SHUBHAM SHARMA (21BEC0954)** respectively, is a bonafide work carried out in partial fulfilment for the award of degree of BACHELOR of ENGINEERING in ELECTRONICS and COMMUNICATION ENGINEERING of the VELLORE INSTITUTE OF TECHNOLOGY, VELLORE during academic year 2023. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report deposited to the Departmental library. The Project report has been approved as it satisfies academic requirements in respect of Project report work prescribe for the Bachelor of Engineering Degree.

Mr. Sunil Yadav
Scientist 'B', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka-
560093

Shri Paramananda Jena
Scientist 'G', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka-
560093

Mr. Himanshu Singh
Scientist 'B', D-PSR Division
LRDE, DRDO
Bangalore, Karnataka-
560093

VELLORE INSTITUTE OF TECHNOLOGY

Vellore, Tamil Nadu - 632014

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

DECLARATION

We hereby declare that the entire project work embodied in this project has been carried out by us in LRDE, DRDO under the supervision of
Shri PARAMANANDA JENA. This report has not been submitted in the part or full for the award of any diploma and degree of this or any other University.

SHRUTI BHANDARI
21BEC0811

ANANYA VERMA
21BEC0868

SARTHAK VARSHNEY
21BEC0927

SHUBHAM SHARMA
21BEC0954

ACKNOWLEDGEMENT

We consider our privilege to express our gratitude and thanks to the following persons for the help and encouragement in completing this Project successfully.

We would like to express our gratitude to **Dr. G. VISWANATHAN**, chancellor of Vellore Institute of Technology and the staff of Electronics and Communication Engineering department, for providing the necessary facilities and requirements to complete Project work within the allotted time.

We wish to express our sincere thanks to Prof. **NOOR MOHAMMAD V** Department of Electronics and Communication Engineering, Vellore Institute of Technology, for his support and encouragement.

We are extremely thankful to the divisional officer **Mrs. ANURADHA D.** (Scientist "G", Radar-I, LRDE) who gave us the permission to work in the prestigious department.

We are very grateful to our guide **Shri PARAMANANDA JENA (**Scientist "G", Radar-I, LRDE) who has shown immense patience toward us and guided us. He was there to help us at every point of time when we needed him.

We also express our thanks to our project coordinator **Mr. SUNIL YADAV and Mr. HIMANSHU SINGH**, for their valuable guidance in completion of the Project.

Finally, we wish to thank and acknowledge the help given by our parents and friends

SHRUTI BHANDARI	**21BEC0811**
ANANYA VERMA	**21BEC0868**
SARTHAK VARSHNEY	**21BEC0927**
SHUBHAM SHARMA	**21BEC0954**

CONTENTS

ABSTRACT

This project titled “Matrix Multiplication using Systolic Array Algorithm” and its implementation in FPGA presents a comprehensive exploration of an advanced approach to matrix multiplication, leveraging the power of systolic arrays, and implements this algorithm on an FPGA.

This project delves into the principles of systolic arrays which offer a highly parallelised architecture for matrix operations. This algorithm breaks down matrix multiplication into a sequence of smaller computations, enabling concurrent processing and minimising data movement, thereby reducing both latency and computational time.

This project provides demonstration of how systolic algorithm can significantly enhance matrix multiplication performance and how such algorithms can be efficiently implemented in hardware using FPGA technology.

Chapter 1

INTRODUCTION

In the realm of digital design and high-performance computing, the efficient multiplication of large numbers is a fundamental operation with a wide range of applications, from signal processing to scientific simulations. Systolic arrays, originally conceptualized by H.T. Kung and Charles E. Leiserson in the early 1980s, have emerged as a powerful and versatile solution for accelerating the multiplication of numbers, particularly in the context of hardware acceleration.

This project report delves into the realm of systolic array multiplication using Verilog, a hardware description language widely employed in the design and verification of digital circuits. Systolic arrays are parallel computing architectures that optimize data flow, allowing for the rapid execution of matrix and vector multiplications. By exploiting parallelism at both the algorithmic and architectural levels, systolic arrays can significantly enhance the computational throughput, making them an attractive choice for applications demanding high-speed numerical operations.

In this report, we will explore the principles behind systolic array multiplication, dissecting its architectural components and detailing the Verilog implementation of a systolic array-based multiplies. We will discuss the advantages and trade-offs associated with systolic array multiplication, highlighting its suitability for various application domains and implementation of the same on Intel FPGA.

By the end of this report, readers will gain a comprehensive understanding of systolic array multiplication using Verilog, equipping them with valuable insights and knowledge to harness the power of this innovative technique in their own digital design projects.

1.1 RTL (Register Transfer Level) Design: It is used in the logic design phase of the integrated circuit design cycle. RTL abstraction is used in HDLs to create high level representation of a circuit. Lower level representations and actual wiring can be derived. It helps in area and power consumption estimation.

1.2 Verilog data types:

In Verilog, the data types are of 2 types:

- Nets-wires which carry forward the signals.
- Registers-storage elements to hold the value.

1.2.1 Nets

- It represents the physical connection between structural entities.
- Except for the trireg, they don't store values.
- In Verilog, net type is defined by assignments like wire, tri, wor, trior, wand and trireg.
- Wire is the most frequently used type.
- The net data type is used when :
  1. The output of some devices drives the signal.
  2. The signal is declared as an input or in-out port.

1.2.2 Registers:

- They are the variables that can store values.
- They are used in the procedural assignment when the signal is on the left-hand side.
- An assignment statement changes the value of the data storage element.
- Reg, integer, time and real are few of the register types, where reg is more frequently used.
- We use *reg* to describe logic, *integer* for variables and arithmetic calculations, *real* for decimal points calculations and *time* data types for storing simulation times.

1.3 Values set in Verilog:

VALUE	DESCRIPTION
0	It means logic zero or false
1	It means logic one or true
X	It represents unknown logical values or don't care
Z	It represents a high impedance of the tri-state gate

1.4 Blocking and Non-blocking assignments:

1.4.1 Blocking assignment

- In blocking assignments the compiler executes the statements, one after another.
- In coding, an "=" operator is used for the assignments.
- The statements written in parallel block are not prevented by the blocking assignments.

1.4.2 Non-blocking assignment

- It is specified using "<=" symbol.
- In non-blocking assignments we can schedule the assignments without blocking the execution of the next statements.
- All assignments can be processed in parallel.

1.5 Counters

- Counters are used to count various operations and events.
- They are used to store and display the number of occurrences of repetitions.
- Depending on the type of clock pulse applied, counters can be synchronous or asynchronous.
- They are mostly designed using JK Flip flop.

1.5.1 Asynchronous Counter

- Also known as Ripple Counter.
- The counter is not synchronized with one external clock.
- Different flip flops used in the counter can be triggered by different pulses on clocks.
- This can slow down the counting speed and also generate wrong counts.

1.5.2 Synchronous Counter

- All the flip-flops are triggered with a single clock.
- Since, all the flip-flops are synchronized the counter is known as synchronous counter.
- The speed of operation is uniform.
- The chances of wrong count generation are minimum.
- The design is challenging, as we must wire all the flip-flops to a single clock.

1.6 Up-Counter

- Counter in upward direction from 0 to 2^n-1.
- Revert 0 after the last number.
- A 4-bit up- counter will count from 0000 to 1111 in repeated order.

1.7 Data width in Verilog: Data width refers to the number of bits in a data signal. It determines the range and precision of data that can be represented in Verilog, and it's a crucial aspect of determining data types.

1.8 2-D array in Verilog: It is a data structure in Verilog that stores data in a grid or matrix format with rows and columns. It allows us to represent and manipulate data in a structured manner.

1.9 Parametric analysis in Verilog: Parametric analysis involves creating Verilog modules or designs with parameters that can be customized. These parameters allow us to create reusable and flexible designs by changing values without rewriting the code.

1.10 Using 'Generate' statement: In Verilog, the 'generate' statement is used to conditionally generate or instantiate hardware elements, such as modules, blocks or other RTL constructs, based on compile-time conditions or parameters. It is often used in conjunction with 'if', 'else if', 'else', and 'for' statements to control the instantiation of modules, logic or other hardware components. It's particularly useful for parameterizing designs or implementing different configurations of a module.

1.11 Generating an IP (Intellectual Property): First we will define the specifications and requirements for the IP core, including its inputs, outputs, functionality and performance criteria. Then we will implements the IP core's logic and behaviour in Verilog, which we can verify by simulating it with test-benches. After that validate the IP core on hardware or n FPGA environment to ensure it functions correctly in a real-world setting.

Chapter 2

SOFTWARE and HARDWARE USED

1. **Quartus (quartus prime pro 21.3):**
   It is a software tool developed by Intel (formerly Altera) and is primarily used for the design and programming of FPGA's. It supports hardware description languages like VHDL and Verilog and offers a comprehensive suite of tools for FPGA development.
   Compilation flow:
   1.1 Design entry: Start by creating or importing the FPGA design using hardware description languages, VHDL or Verilog.
   1.2 Synthesis: Quartus performs synthesis, which translates the RTL description of the design into a gate-level netlist. This optimized netlist represents how to design will be implemented on the FPGA.
   1.3 Place and route: The design undergoes a place and route process to map the logical elements onto the physical resources of the target FPGA. This includes configuring the look-up tables, flip flops and interconnects.
   1.4 Compilation: During compilation, Quartus optimizes the design for the target FPGA architecture, taking into account timing constraints and resource utilization.
   1.5 Timing analysis: Quartus performs static timing analysis to ensure that the design meets timing requirements, such as setup and hold times for flip flops.
   1.6 Configuration Generation: Quartus generates a configuration bitstream file that the programs the FPGA to implement the desired design.
   1.7 Testing and Debugging: After programming the FPGA, engineers can test the design on the real hardware environment and debug any issues or unexpected behaviour.

2. **Questa Intel FPGA edition 2021.2 (quartus prime pro 21.3):**
   It is a software tool used for functional verification and simulation of digital designs, particularly in the context of VHDL and Verilog-based designs. It offers a range of features for design and verification engineers.

3. **<u>FPGA (Field Programmable Gate Array)</u>**

   An FPGA is a reconfigurable digital logic device that provides a highly flexible and programmable platform for implementing a wide range of digital circuits and systems.

   Intel Agilex FPGA’s are a part of Intel’s FPGA product lineup, known for their versatility, performance, and power efficiency. These FPGA’s are used in applications like data centre acceleration, artificial intelligence, networking and more.

Chapter 3

LITERATURE SURVEY

This chapter deals with the papers and books prepared by various authors and their studies on various parameters.

[1] Ming-Bo Lin, "Digital System Designs and Practices using Verilog HDL and FPGA's".
This book gives us basic features and capabilities of Verilog HDL, FPGA System Designs and contains basic Combinational and Sequential modules. It discusses three-closely related issues of Hierarchical Structural Modelling, which include instantiations, generate statements and configurations. The instantiation is the mechanism through which the hierarchical structure is formed by modules being embedded into other modules. The generate statements can conditionally generate declaration and instantiations into a design.
Also, we studied that for modelling a ROM device without timing checks, it only needs to declare a memory with the required number of words and the word width. In general, the contents of ROMs can be initialized by using procedure assignment statements or reading from a file through the use of system tasks: $readmemb or $readmemh.

[2] Henry Y. H. Chuang and Guo He presented a paper on "A Versatile Systolic Array for Matrix Computations".
This paper deals with the factors affecting the utility of a VLSI processor array, Feedback Systolic Array, VLSI processor array based on Faddeev's algorithm and sparsity in matrix. A feedback system is also introduced in which we can remove the excess subarrays and at the same time achieve problem-size independence. They also provide a better way to solve a matrix which is by using Gaussian Elimination and Faddeev's algorithm. We studied that if any problem is larger than the array then it is generally decomposed into sub problems which needs to be solved in the array one at a time, then the result of each computation is stored in the host for further processing. To make an array system size independent we just need to replace the column of q 's' arrays with an 's' array with feedback. So that the processing time for the first phase will be q times as much because all data strips, except for the first one, have to be processed by single 's' array q times. Whereas, processing time for 2nd phase is not affected because column of q's' arrays are not used in the phase.

[3] Mark A. Richards, James A. Scheer, William A. Holm, “Principles of Modern Radar Basic Principles”.

This book focuses on basic concepts, radar signal phenomenology such as clutter, atmospheric effects, and Doppler effects, description of all major subsystems of modern radars such as the antenna, transmitter, receiver, including modern architectural elements. We studied about some radar cultural information such as the “band” terminology (e.g., L-band, X-band), basic radar configurations and waveforms, basic radar measurements and radar applications.

Chapter 4

Systolic Array Algorithm

A systolic array is an arrangement of processors in an array in which the data flows synchronously across the array between neighbours, usually with different data flowing in different directions. It is a specialized form of parallel computing, where multiple processors are connected by short wires.

Systolic arrays have proven themselves as an efficient hardware acceleration technique for matrix multiplication, as they can produce very high throughput due to the high degree of concurrent processing possible in them.

The array system is capable of solving many matrix problems involving arbitrarily large matrices. It can also process sparse matrices efficiently by skipping the blocks of zeros.

Each processor at each step takes in data from one or more neighbours (e.g. North and West), processes it and, in the next step, outputs results in the opposite direction (South and East).

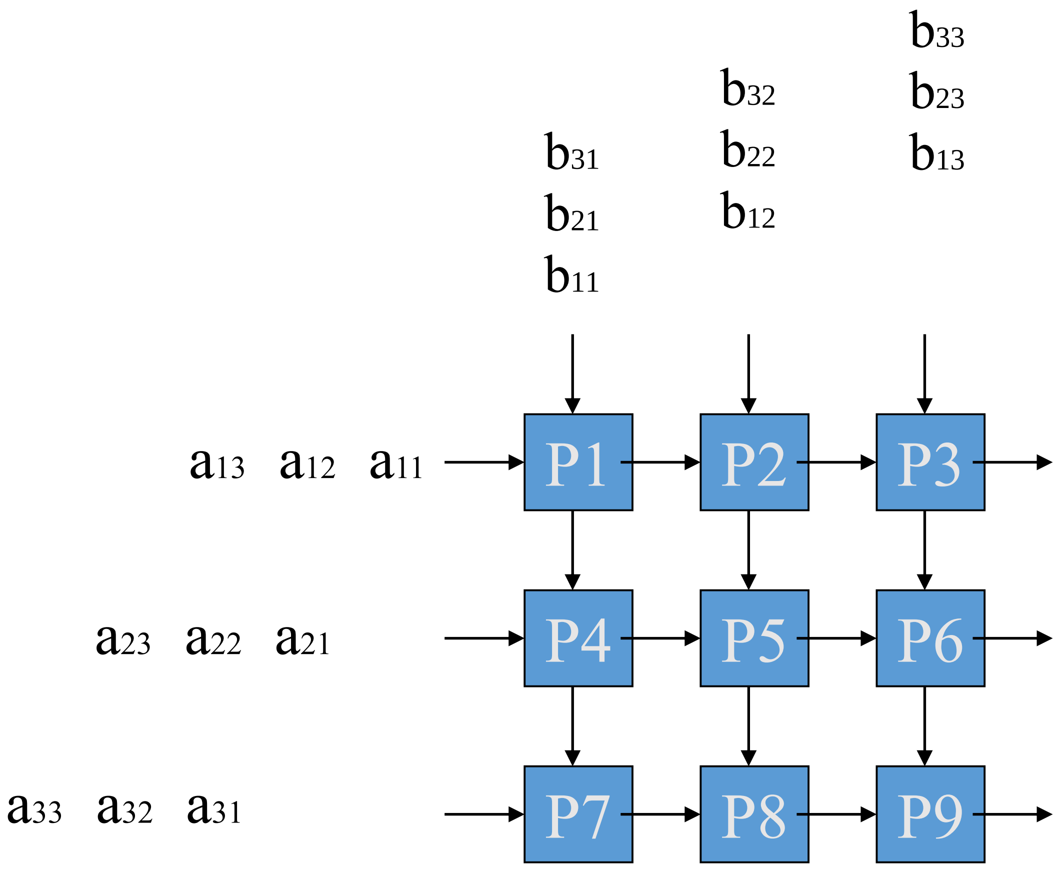
After performing the required operations, the cells share information with their neighbors.

ADVANTAGES:

- It is extremely fast.
- It has easily scalable architecture.
- It can do various tasks that single processor machines cannot attain.
- It can turn some exponential problems into linear or polynomial time.

DISADVANTAGES:

- Implementation is expensive.
- As they are a highly specialized processor type so they are not needed on most applications.
- Difficult to implement and build.


b_{33} b_{23} b_{13}
b_{32} b_{22} b_{12}
b_{31} b_{21} b_{11}
a_{13} a_{12} a_{11}
a_{23} a_{22} a_{21}
a_{33} a_{32} a_{31}
P1
P2
P3
P4
P5
P6
P7
P8
P9

At every tick of the global system clock data is passed to each processor from two different directions, then it is multiplied and the result is saved in a register.

We need to modify the input data, like so:

Flip columns 1 & 3 $\longrightarrow$

$$\begin{matrix} a_{13} & a_{12} & a_{11} \\ a_{23} & a_{22} & a_{21} \\ a_{33} & a_{32} & a_{31} \end{matrix}$$

Flip rows 1 & 3 $\longrightarrow$

$$\begin{matrix} b_{31} & b_{32} & b_{33} \\ b_{21} & b_{22} & b_{23} \\ b_{11} & b_{12} & b_{13} \end{matrix}$$

and it finally staggers the data sets for input.

Chapter-5

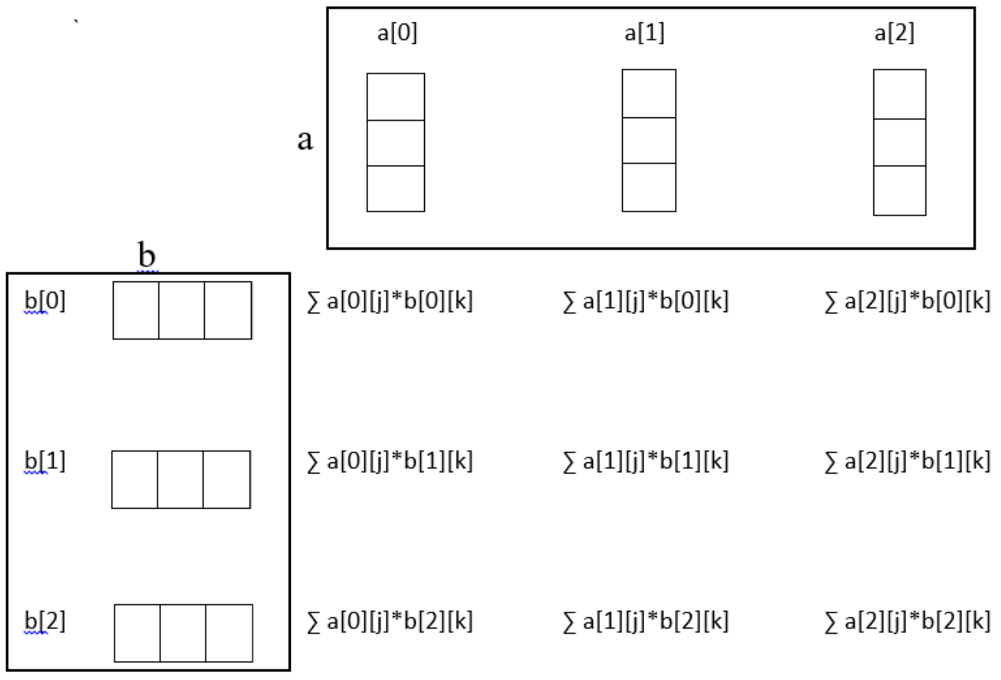
1st cell code with reset pin

We created a module of a cell which represents each cell of the resultant matrix, which is the result of the multiplication of 2 matrices using systolic array algorithm. Later we instantiated this module for implementing 3x3 systolic array multiplication.
'a' and 'b' are the values of the input matrices, s represents the sum of the multiplication of input values from a and b.

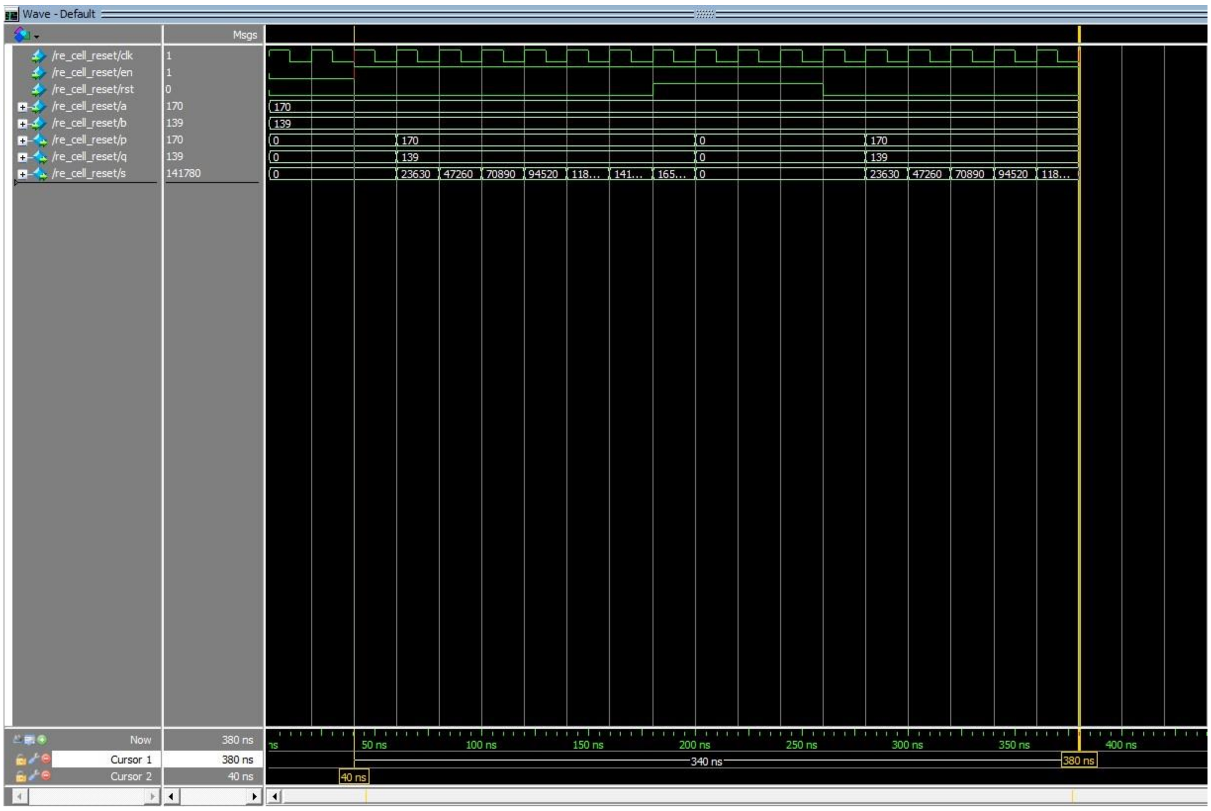
When reset pin is 1 or HIGH, the system resets and sets to 0.

When reset pin is 0 or LOW, the system depends upon the enable pin's state.

When enable is 1 or HIGH, the system starts reading the data from the ROM and implements the systolic array algorithm and stores the value of the summation of the multiplication of the used input values in 's', simultaneously it stores the value of a and b in p and q respectively so that they can be used as input in the adjacent cells (s=s+a*b).



a and b are representing collective ROMs, where j and k are the indices of each ROM in a and b respectively.

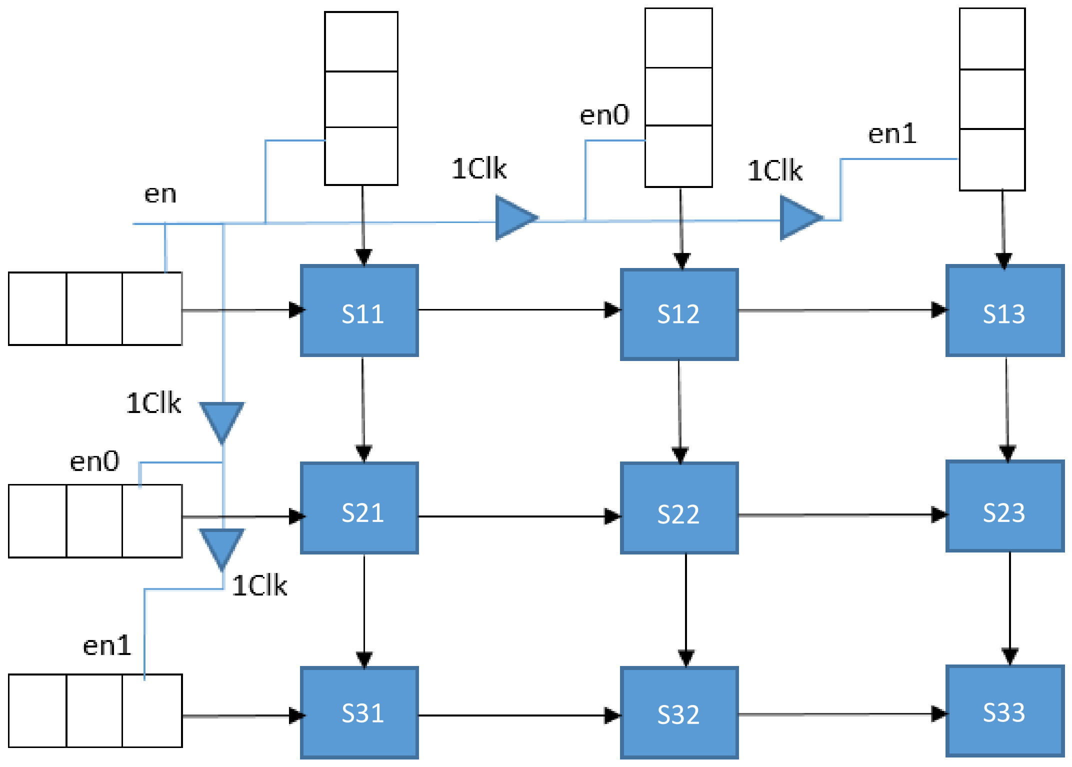


From the above simulation we can observe that the above simulation represents the working of a single cell.

Chapter-6

Reusable code with reset pin

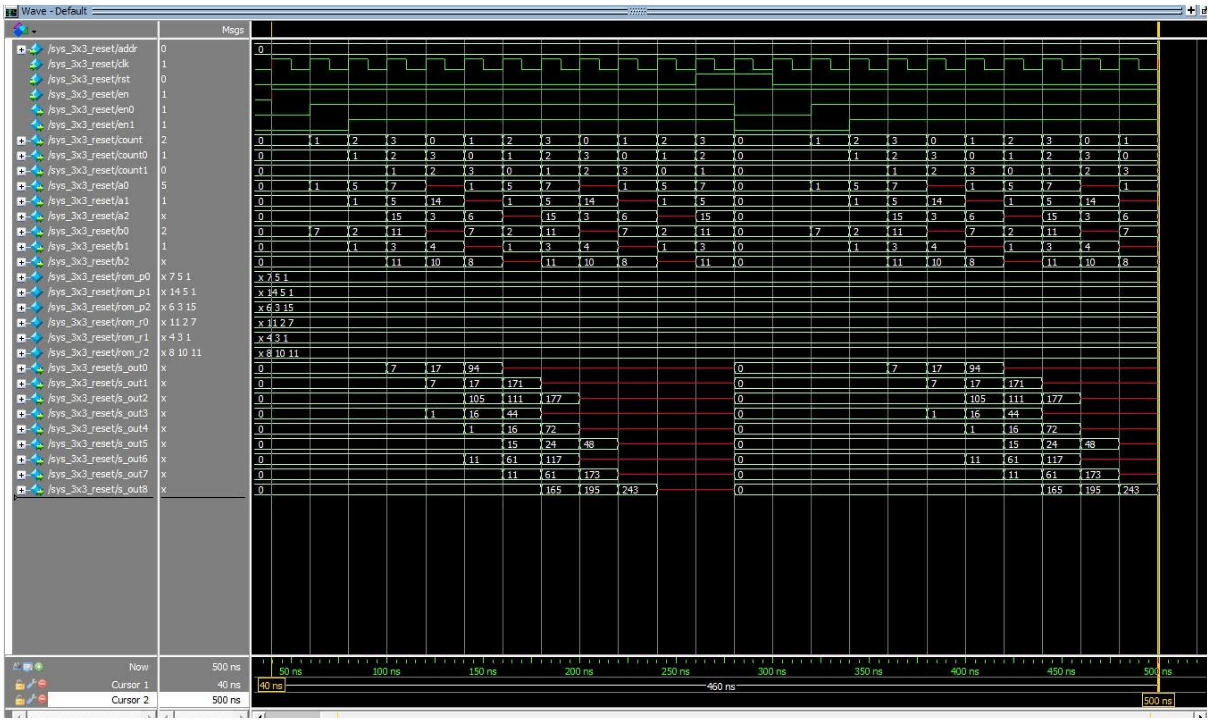
In this module we created 6-txt files for representing ROMs, which contain 8-bit binary data that are being used as inputs a and b in each cell as per the pattern of systolic array algorithm.
The S11, S12, S13, S21 and S31 are the cells which are connected to the ROMs and are termed as boundary cells as they are directly using the ROM data. S11 is connected to 2 ROMs simultaneously while the others are only connected to 1 ROM each. The rest of the cells are using the ‘p’ and ‘q’ values of their adjacent cells as the input.



The above diagram represents the methodology that we used for implementing the systolic array algorithm. The blue triangles represent buffers, which are being used for creating a delay by 1 clock cycle to adhere by the pattern of the systolic algorithm.
Because of the presence of the buffer the en0 pin attached to 2nd ROM takes the value of the en pin from the previous clock cycle. Similarly, en1 pin attached to

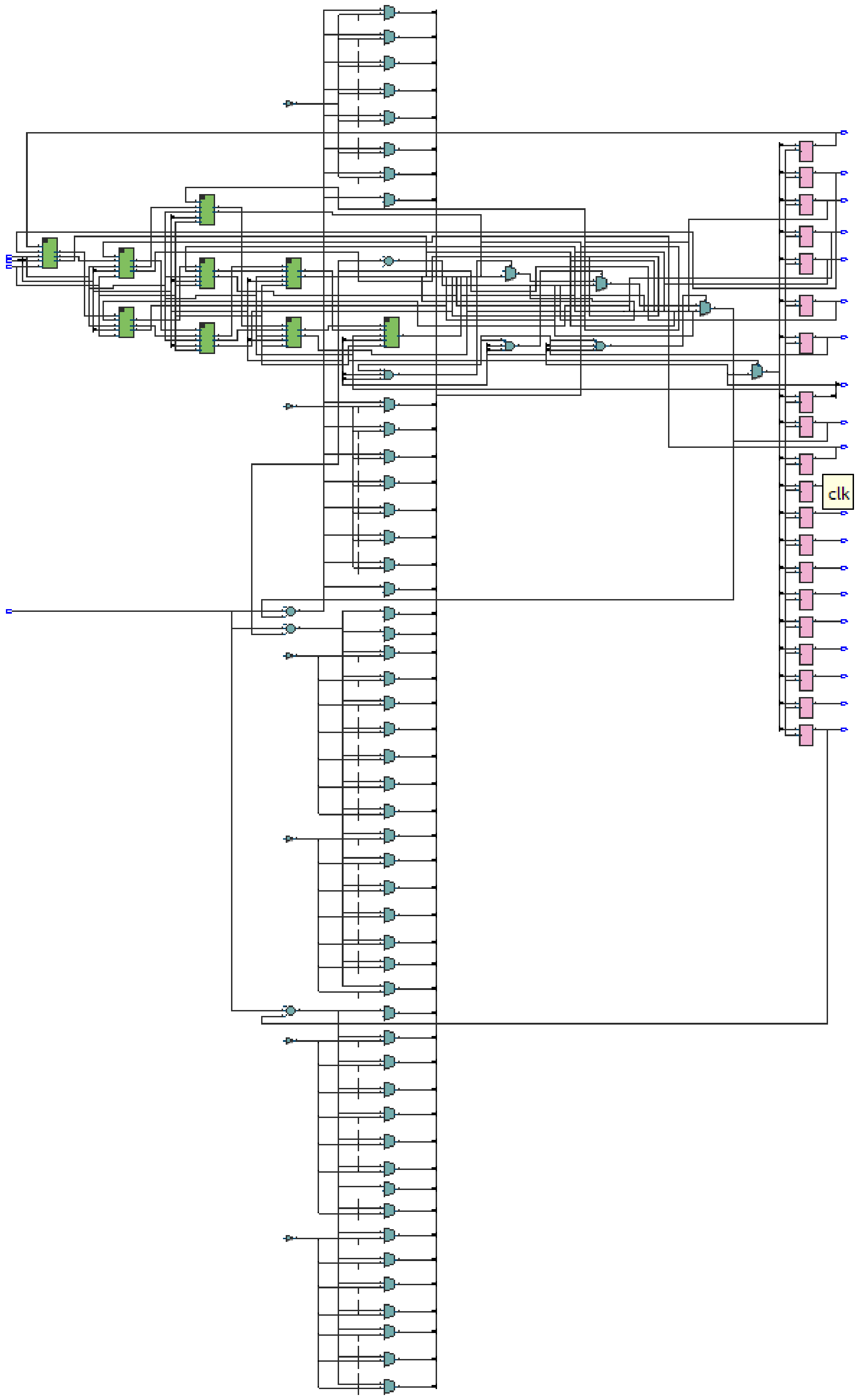
3rd ROM takes the value of the en0 pin from the previous clock cycle. After simulation, in the first clock we will start getting the value of S11, in the 2nd clock we will start getting the values of S12 and S21, in the 3rd clock we will start getting the values of S13, S22 and S31, in the 4th clock we will start getting the values of S23 and S32, in the 5th clock we will start getting the value of S33. The whole implementation of the algorithm is done in 7 clock ticks.

<u>OUTPUT:</u>



```
Wave - Default
                          Msgs
/sys_3x3_reset/addr       0
/sys_3x3_reset/clk        1
/sys_3x3_reset/rst        0
/sys_3x3_reset/en         1
/sys_3x3_reset/en0        1
/sys_3x3_reset/en1        1
/sys_3x3_reset/count      2
/sys_3x3_reset/count0     1
/sys_3x3_reset/count1     0
/sys_3x3_reset/a0         5
/sys_3x3_reset/a1         1
/sys_3x3_reset/a2         x
/sys_3x3_reset/b0         2
/sys_3x3_reset/b1         1
/sys_3x3_reset/b2         x
/sys_3x3_reset/rom_p0     x 7 5 1
/sys_3x3_reset/rom_p1     x 14 5 1
/sys_3x3_reset/rom_p2     x 6 3 15
/sys_3x3_reset/rom_r0     x 11 2 7
/sys_3x3_reset/rom_r1     x 4 3 1
/sys_3x3_reset/rom_r2     x 8 10 11
/sys_3x3_reset/s_out0     x
/sys_3x3_reset/s_out1     x
/sys_3x3_reset/s_out2     x
/sys_3x3_reset/s_out3     x
/sys_3x3_reset/s_out4     x
/sys_3x3_reset/s_out5     x
/sys_3x3_reset/s_out6     x
/sys_3x3_reset/s_out7     x
/sys_3x3_reset/s_out8     x

Now        500 ns
Cursor 1   40 ns
Cursor 2   500 ns
```

RTL view:



Chapter-7

Systolic Implementation on FPGA

FPGA (Field-Programmable Gate Array), is a reconfigurable digital logic device that provides a highly flexible and programmable platform for implementing a wide range of digital circuits and systems.

FPGA Architecture:

FPGAs consists of an array of programmable logic blocks, interconnect resources, and input/output blocks. Here's a breakdown of these components:

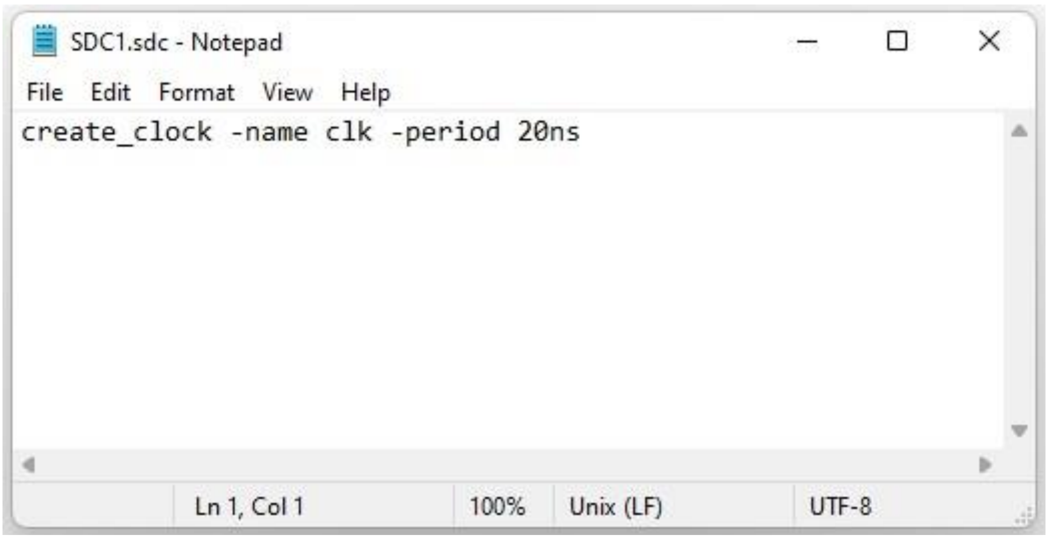
1. **Programmable logic blocks (PLBs):** PLBs are the heart of an FPGA. They are composed of look-up tables, flip-flops and multiplexers.
2. **Interconnect Resources:** Interconnect resources facilitate connections between PLBs and other components. The routing fabric consists of programmable switches, wires, and routing channels. This flexibility allows designers to create custom connections for data flow.
3. **Input/output Blocks:** IOBs are used to interface the FPGA with the external world. They provide pins for connecting to sensors, displays, memory and other external devices.

FPGA Programming:

FPGAs are programmed using hardware description languages (HDLs) like Verilog or VHDL. The programming involves these key steps:

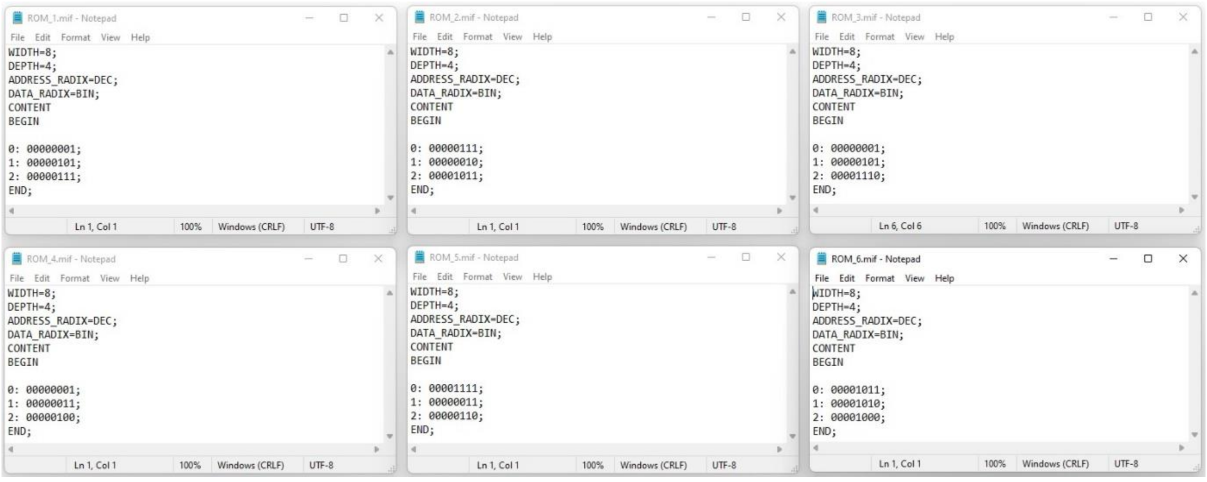
1. **Design entry:** Engineers create a high-level design using HDLs. This design describes the desired functionality of the FPGA.
2. **Synthesis:** The HDL code is synthesized into a netlist, which represents the logical functions and connections within the FPGA.
3. **Place and Route:** The synthesis output is mapped onto the physical FPGA, determining how the logic functions are placed within the PLBs and interconnected.
4. **Configuration:** The final bitstream file is generated and loaded onto the FPGA to configure its logic elements. The bitstream configures the PLBs, interconnects, and other FPGA resources to implement the desired functionality.

SDC File:



```
create_clock -name clk -period 20ns
```

MIF Files:



ROM_1.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00000001;
1: 00000101;
2: 00000111;
END;
```

ROM_2.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00000111;
1: 00000010;
2: 00001011;
END;
```

ROM_3.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00000001;
1: 00000101;
2: 00001110;
END;
```

ROM_4.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00000001;
1: 00000011;
2: 00000100;
END;
```

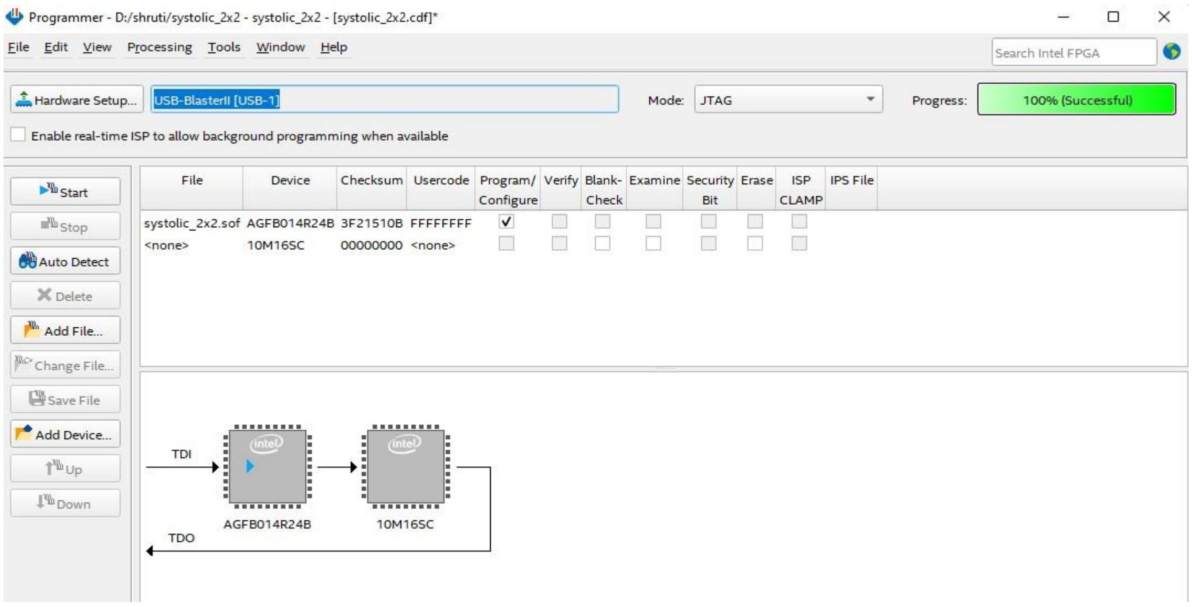
ROM_5.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00001111;
1: 00000011;
2: 00000110;
END;
```

ROM_6.mif

```
WIDTH=8;
DEPTH=4;
ADDRESS_RADIX=DEC;
DATA_RADIX=BIN;
CONTENT
BEGIN

0: 00001011;
1: 00001010;
2: 00001000;
END;
```

FPGA Programmer:



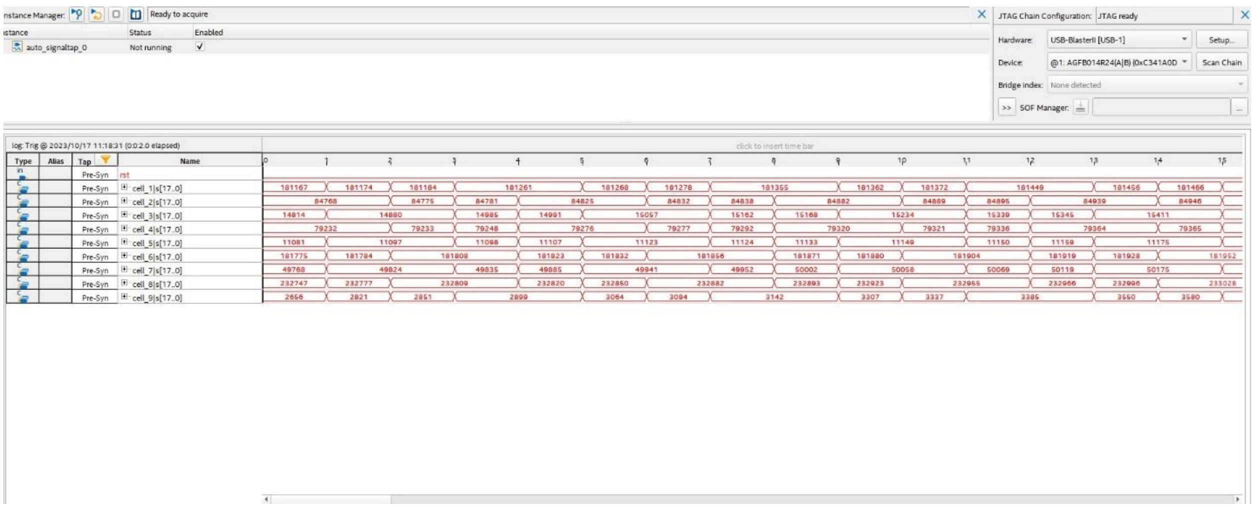
Programmer - D:/shruti/systolic_2x2 - systolic_2x2 - [systolic_2x2.cdf]*

Hardware Setup... USB-BlasterII [USB-1]   Mode: JTAG   Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase	ISP CLAMP	IPS File
systolic_2x2.sof	AGFB014R24B	3F21510B	FFFFFFFF	✓							
<none>	10M16SC	00000000	<none>								

FPGA Simulation:



Instance Manager: Ready to acquire

Instance	Status	Enabled
auto_signaltap_0	Not running	✓

JTAG Chain Configuration: JTAG ready

Hardware: USB-BlasterII [USB-1]

Device: @1: AGFB014R24(A|B) (0xC341A0D

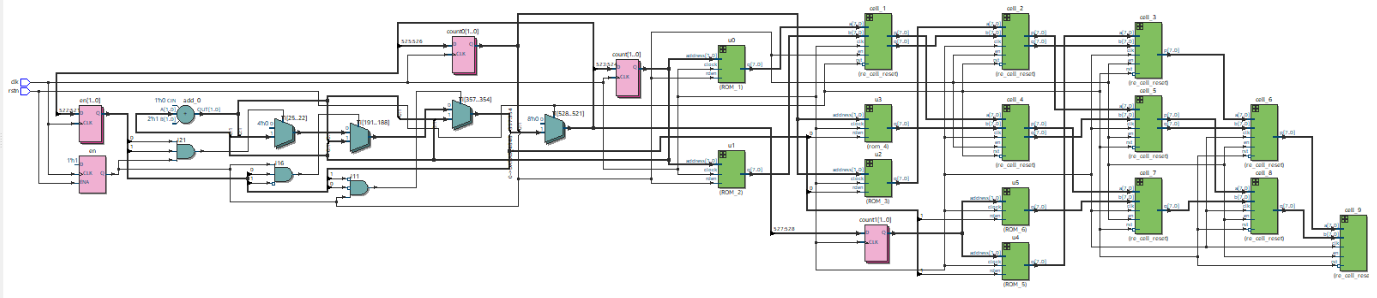
Bridge index: None detected

SOF Manager:

log: Trig @ 2023/10/17 11:18:31 (0:0:2.0 elapsed)

Type	Alias	Tap	Name	0 – 15
in		Pre-Syn	rst	
		Pre-Syn	cell_1|s[17..0]	181167, 181174, 181184, 181261, 181268, 181278, 181355, 181362, 181372, 181449, 181456, 181466
		Pre-Syn	cell_2|s[17..0]	84768, 84775, 84781, 84825, 84832, 84838, 84882, 84889, 84895, 84939, 84946
		Pre-Syn	cell_3|s[17..0]	14814, 14880, 14985, 14991, 15057, 15162, 15168, 15234, 15339, 15345, 15411
		Pre-Syn	cell_4|s[17..0]	79232, 79233, 79248, 79276, 79277, 79292, 79320, 79321, 79336, 79364, 79365
		Pre-Syn	cell_5|s[17..0]	11081, 11097, 11098, 11107, 11123, 11124, 11133, 11149, 11150, 11159, 11175
		Pre-Syn	cell_6|s[17..0]	181775, 181784, 181808, 181823, 181832, 181856, 181871, 181880, 181904, 181919, 181928, 181952
		Pre-Syn	cell_7|s[17..0]	49768, 49824, 49835, 49885, 49941, 49952, 50002, 50058, 50069, 50119, 50175
		Pre-Syn	cell_8|s[17..0]	232747, 232777, 232809, 232820, 232850, 232882, 232893, 232923, 232955, 232966, 232996, 233028
		Pre-Syn	cell_9|s[17..0]	2656, 2821, 2851, 2899, 3064, 3094, 3142, 3307, 3337, 3385, 3550, 3580

RTL View:



Chapter 8

RADAR

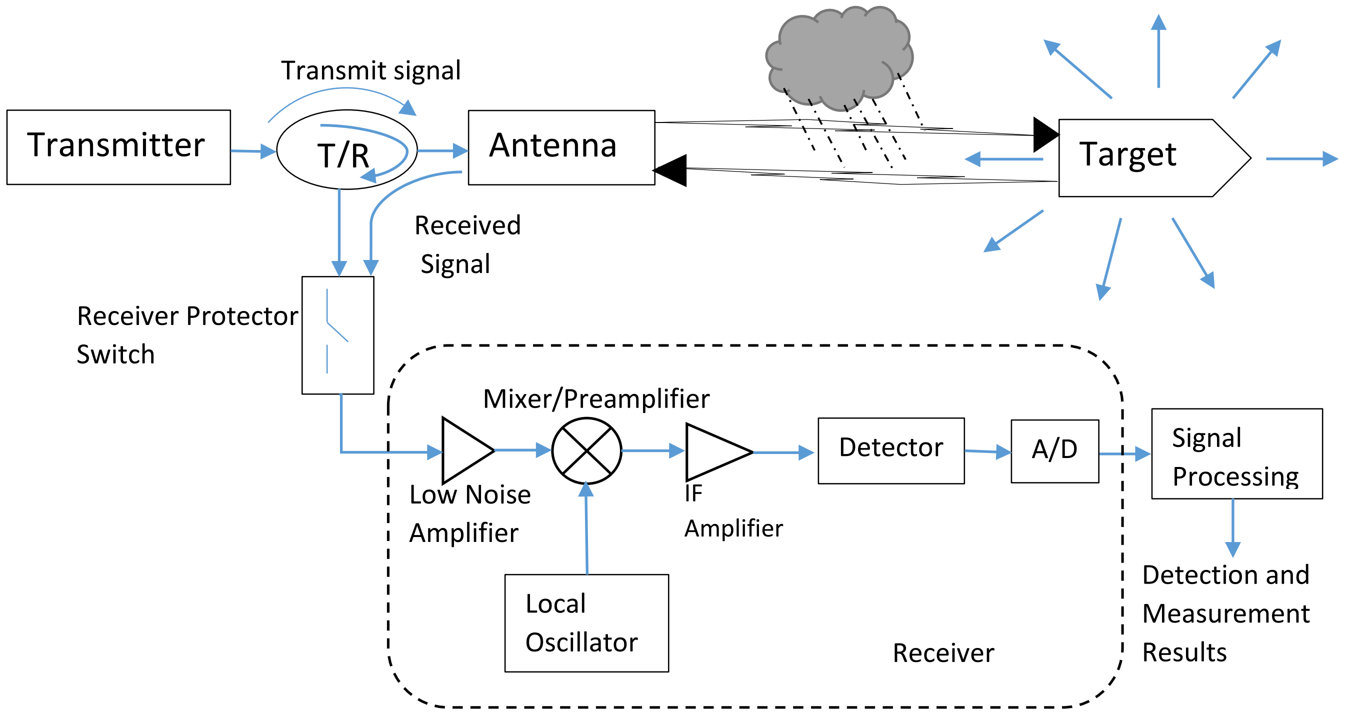
1. INTRODUCTION :

Radar (Radio Detection and Ranging) technology has revolutionized the way we perceive and interact with our surroundings, playing a pivotal role in applications ranging from weather forecasting to military defence and autonomous navigation systems. At the heart of radar systems lies the need for efficient and high speed signal processing, particularly in tasks like target detection and range estimation.

Radar is an electrical system that transmits radio frequency (RF) electromagnetic (EM) waves towards a region of interest and capture their reflections to discern the location, speed, and characteristics of objects in their vicinity.

Modern radars, however, are sophisticated transducer/computer systems that not only detect targets and determine target range but also track, identify, image and classify targets while suppressing strong unwanted interference such as echoes from the environment (known as clutter) and countermeasures (jamming).

2. BLOCK DIAGRAM:



2.1 RADAR COMPONENTS:

Radar systems consists of several components:

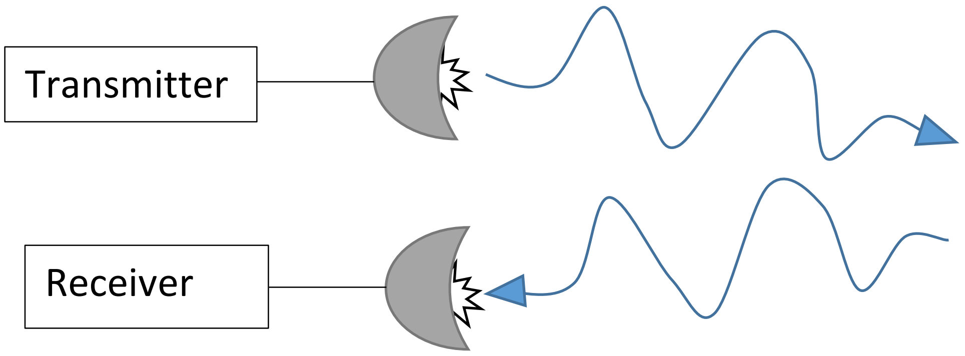
1. **Transmitter**: It generates EM waves, which are then radiated into space through an antenna.
2. **T/R Device:** The transmitter is connected to the antenna through a transmitter/receiver (T/R) device (usually a circulator or a switch). It provides isolation between the transmitter and receiver to protect the sensitive receiver components from the high-powered transmit signal.
3. **Antenna**: It is a subsystem that takes these EM waves an input from the transmitter and introduces them into the propagation medium (normally the atmosphere).
4. **Receiver unit**: It captures the radar echoes reflected by objects, amplifies and converts RF signal to an intermediate frequency (IF) and subsequently applies the signal to an analog-to-digital (A/D) converter after removing the carrier from the modulated target signal and then sends the target data to a Signal Processor for further analysis.
5. **Signal Processor:** It uses Signal processing techniques, such as filtering, pulse compression, and Doppler processing, to extract the useful information from the received signals.

3. RF and Radar Bands:

Band	Frequency Range
High frequency (HF)	3 – 30 MHz
Very High frequency(VHF)	30 – 300 MHz
Ultra High Frequency(UHF)	300 – 1 GHz
L	1 – 2 GHz
S	2 – 4 GHz
C	4 – 8 GHz
X	8 – 12 GHz
Ku(“under” K-band)	12 – 18 GHz
K	18 – 27 GHz
Ka(“above” K-band)	27 – 40 GHz
V	40 – 75 GHz
W	75 – 110 GHz
mm	100 – 300 GHz

4. Monostatic and Bistatic configuration:

4.1 Bistatic



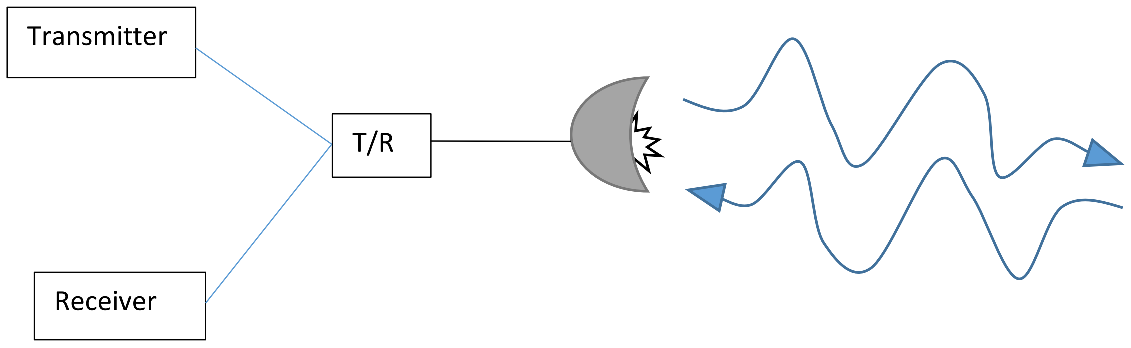
In the bistatic configuration, there are separate antennas for the transmit and receive radar functions. Use of two antennas alone does not determine whether a system is monostatic or bistatic. The system is considered to be bistatic only if there is sufficient separation between the two antennas such that “the angles or ranges to the target are sufficiently different”.

The transmitter is a high-powered device that transmits EM waves with power levels in the range of hundreds of kilowatts or megawatts. The receiver is a power sensitive device that can respond to EM waves in the range of milliwatts to nanowatts. If directly introduced to the receiver, high power EM waves from the transmitter could severely damage the receiver’s sensitive components and can lead to self-jamming (unable to detect the target).

This can be prevented by bistatic radar configuration which provides significant isolation by physically separating the transmitter and receiver antennas.

There are some applications for which the bistatic system has significant separation between the transmitter and receiver. For example, a *semi active missile* has only the receiver portion on board. The transmitter is on another platform. The transmitter “illuminates” the target while the missile “homes” in on the signal reflected from the target.

4.2 Monostatic



In the Monostatic configuration, one antenna serves both the transmitter and receiver. If the two antennas are very close together, say, on the same structure, then the system is considered to be monostatic.

Most of the radars in the modern day are monostatic- since only one antenna is required which is a more practical design. It is more difficult to provide isolation between the transmitter and receiver since both subsystems must be attached to the antenna. A T/R device provides this isolation.

5. Radar Waveforms:

5.1 CW Waveform

With the CW waveform, the transmitter continually transmits a signal, usually without interruption, while both the radar transmitter and receiver are operating. Continuous wave radars often employ the bistatic configuration to effect transmitter/receiver isolation. Since this isolation is not perfect, there is some competing signal due to the leakage, relegating CW systems to relatively low power and hence short-range applications. Since the CW radar is continuously transmitting, determination of the transmitted EM wave's round- trip time and, thus, target range, must be accomplished by changing the characteristics of the wave(e.g., changing the wave's frequency overtime).

CW radars tend to be simple radars and are used for such applications as police speed-timing radars, altimeters, and proximity fuses.

5.2 Pulse Waveform

Pulse radar transmit finite duration EM wave pulses, separated by times during which the transmitter is off, or pulse width "τ". During this time, the receiver is isolated from the antenna, or blanked, thus protecting its sensitive components from transmitter's high-power EM waves. No received signals can be detected during this time and the receiver is connected to the antenna, allowing it to receive any EM waves (echoes) that may have been reflected from objects in the environment. This "listening" time plus the pulse width represents one pulsed radar cycle time, normally called the *interpulse period* (IPP) or *pulse repetition interval* (PRI).

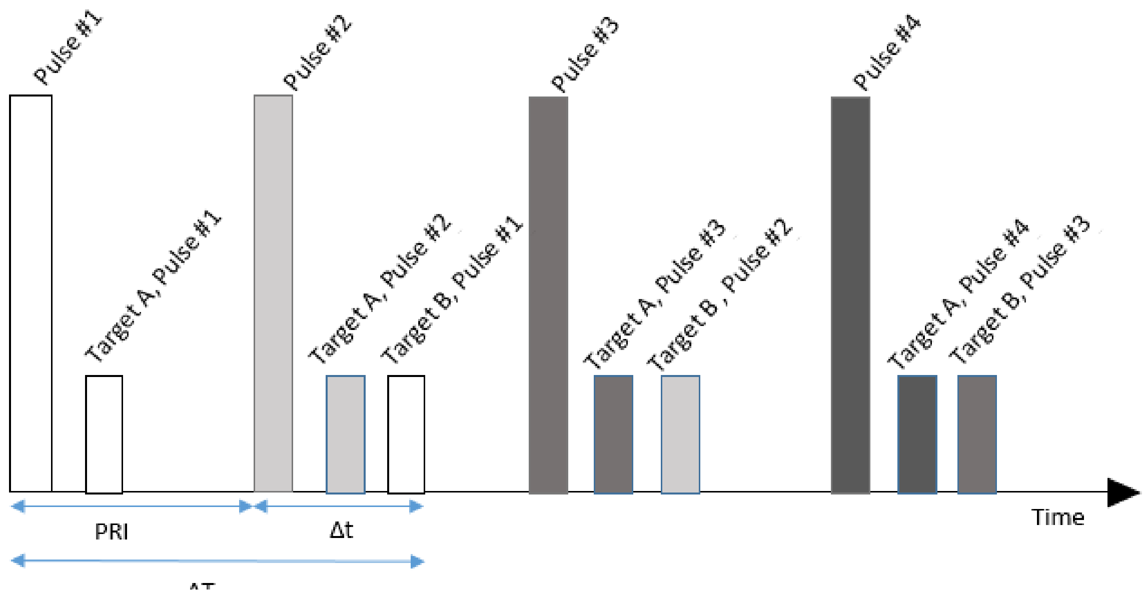
5.2.1 Pulse Repetition Frequency (PRF)

The number of transmit/receive cycles the radar completes per second is called the *pulse repetition frequency* (PRF), which is properly measured in *pulses per second* (PPS) but is often expressed in hertz. The PRF and PRI are related according to

$$\mathrm{PRF} = \frac{1}{\mathrm{PRI}}$$

5.2.2 Unambigous Range Measurement

Problems can occur in pulsed radar when determining the range to targets if the pulse round trip travel time, ΔT, between the radar and the distant target is greater than the interpulse period, IPP. In this case, the EM wave in a given pulse will not return to the radar's receiver before the next pulse is transmitted, resulting in a time ambiguity and related range ambiguity.



This situation is illustrated in the figure above. The tall rectangles represent transmitted pulses; the shorter rectangles represent the received echoes from two targets. The shading of the target echoes matches the shading of the pulse from which they originated. The time delay to target A and back is less than the interpulse period, so the echo from target A from a given pulse is received before the next pulse is transmitted. The time delay ΔT to target B is greater than the PRI.

Range ambiguities can be avoided by ensuring that the interpulse period, PRI, is long enough or, equivalent, the project repetition frequency PRF is low enough, such that all echoes of interest from a given pulse return to the radar receiver before the next pulse is transmitted. The round trip time is given by

$$\Delta T = \frac{2R}{c}$$

The unambiguous range $R_{ua,}$ is the maximum range at which the range to a target can be measured unambiguosly by the radar. It is given by

$$R_{ua} = \frac{c}{2PRF}$$

6. Applications:

1. Military applications

- Search radars
- Air defense system
- Over-the-horizon search radars
- Ballistic Missile Defense radars
- Radar seekers and fire control radars
- Instrumentation/Tracking test range radars
- Tracking, fire control, and missile support radars
- Multifunction radars
- Artillery locating radars
- Target identification radars

2. Commercial applications

- Process control radars
- Airport surveillance radars
- Weather radars

- Wake Vortex radars
- Marine navigation radars
- Satellite mapping radars
- Police speed measuring radars
- Automotive collision avoidance radars
- Ground penetration radars
- Radar altimeters

APPENDIX

CODES

CODE:

1. reusable cell code with RESET pin

```
//reusable cell code with RESET pin

module re_cell_reset
//ADDR_WIDTH is the width of the address of the data stored in the memory
//DATA_WIDTH is the width of the data stored in the memory
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=1)
(
        input clk, en, rst,
        input [DATA_WIDTH-1:0]a,
        input  [DATA_WIDTH-1:0]b,

        output reg [2*(DATA_WIDTH)+1:0] s,
        output reg [DATA_WIDTH-1:0]p,
        output reg [DATA_WIDTH-1:0]q
        );
        initial
        begin
        s=0;
        p=0;
        q=0;
        end
        always @ (posedge clk)
        begin
        if(rst==0)
        begin
         if(en==1)
                begin
                s<=s+a*b;       //formulation of systolic

                p<=a;
                q<=b;
                end
                end
        else
        begin
                s<=0;   //when rst pin is 1 or HIGH , the system resets
                p<=0;
                q<=0;
        end
        end
```

```
endmodule
```

2. 3x3 systolic code without RESET pin

```
//3x3 systolic code without RESET pin

module sys_3x3_reset
//ADDR_WIDTH is the width of the address of the data stored in the memory
//DATA_WIDTH is the width of the data stored in the memory

#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=2)
(
            input [(ADDR_WIDTH-1):0] addr,
            input clk, en, rst,

            output reg [DATA_WIDTH-1:0]a0,
            output reg  [DATA_WIDTH-1:0]b0,
            output reg [DATA_WIDTH-1:0]a1,
            output reg  [DATA_WIDTH-1:0]b1,
            output reg [DATA_WIDTH-1:0]a2,
            output reg  [DATA_WIDTH-1:0]b2,
            output reg [ADDR_WIDTH-1:0]count,
            output reg [ADDR_WIDTH-1:0]count0,
            output reg [ADDR_WIDTH-1:0]count1,
            output reg [2*(DATA_WIDTH)+1:0] s_out0,
            output reg [2*(DATA_WIDTH)+1:0] s_out1,
             output reg [2*(DATA_WIDTH)+1:0] s_out2,
            output reg [2*(DATA_WIDTH)+1:0] s_out3,
            output reg [2*(DATA_WIDTH)+1:0] s_out4,
            output reg [2*(DATA_WIDTH)+1:0] s_out5,
            output reg [2*(DATA_WIDTH)+1:0] s_out6,
            output reg [2*(DATA_WIDTH)+1:0] s_out7,
            output reg [2*(DATA_WIDTH)+1:0] s_out8,
            output reg en0,en1

);

//declaring output wires
 wire [2*(DATA_WIDTH)+1:0] s0;
 wire [2*(DATA_WIDTH)+1:0] s1;
 wire [2*(DATA_WIDTH)+1:0] s2;
 wire [2*(DATA_WIDTH)+1:0] s3;
 wire [2*(DATA_WIDTH)+1:0] s4;
 wire [2*(DATA_WIDTH)+1:0] s5;
 wire [2*(DATA_WIDTH)+1:0] s6;
 wire [2*(DATA_WIDTH)+1:0] s7;
wire [2*(DATA_WIDTH)+1:0] s8;

//declaring cell interconnecting wires
wire  [DATA_WIDTH-1:0]p0;
wire  [DATA_WIDTH-1:0]p1;
wire  [DATA_WIDTH-1:0]p2;
wire  [DATA_WIDTH-1:0]p3;
wire  [DATA_WIDTH-1:0]p4;
wire  [DATA_WIDTH-1:0]p5;
wire  [DATA_WIDTH-1:0]p6;
wire  [DATA_WIDTH-1:0]p7;
wire  [DATA_WIDTH-1:0]p8;

wire  [DATA_WIDTH-1:0]q0;
wire  [DATA_WIDTH-1:0]q1;
wire  [DATA_WIDTH-1:0]q2;
wire  [DATA_WIDTH-1:0]q3;
wire  [DATA_WIDTH-1:0]q4;
wire  [DATA_WIDTH-1:0]q5;
wire  [DATA_WIDTH-1:0]q6;
wire  [DATA_WIDTH-1:0]q7;
wire  [DATA_WIDTH-1:0]q8;

	// Declare the ROM variable
	reg [DATA_WIDTH-1:0] rom_p0[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r0[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_p1[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r1[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_p2[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r2[2**ADDR_WIDTH-1:0];

	initial
	begin
	count=0;
	count0=0;
	count1=0;
	en1=0;
	en0=0;
		//Reading data from the ROM
		$readmemb("single_port_rom_init_0.txt", rom_p0);
		$readmemb("single_port_rom_init_1.txt", rom_r0);
		$readmemb("single_port_rom_init_2.txt", rom_p1);
		$readmemb("single_port_rom_init_3.txt", rom_r1);
		$readmemb("single_port_rom_init_4.txt", rom_p2);
		$readmemb("single_port_rom_init_5.txt", rom_r2);

	end
```

```
always @ (posedge clk)
begin
en0<=en;          //creating buffer of 1 clock cycle
en1<=en0;         //creating buffer of 2 clock cycle
if(rst==0)
begin
if(en==1 && en0==0 && en1==0)
begin
//retrieving data from ROMs
        a0 <= rom_p0[addr+count];
        b0 <= rom_r0[addr+count];

        //up counters
        count<=count+1;
        count0<=count;
        count1<=count0;

        // connecting the output wires to reg to store the values.
        s_out0<=s0;
        s_out1<=s1;
        s_out2<=s2;
        s_out3<=s3;
        s_out4<=s4;
        s_out5<=s5;
        s_out6<=s6;
        s_out7<=s7;
        s_out8<=s8;
        end

else if(en1==0 && en0==1 && en==1)
begin
//retrieving data from ROMs
        a0 <= rom_p0[addr+count];
        b0 <= rom_r0[addr+count];
        a1 <= rom_p1[addr+count0];
        b1 <= rom_r1[addr+count0];

        //up counters
        count<=count+1;
        count0<=count;          //establishing a buffer
        count1<=count0;         //establishing a buffer

        // connecting the output wires to reg to store the values.
        s_out0<=s0;
        s_out1<=s1;
        s_out2<=s2;
        s_out3<=s3;
        s_out4<=s4;
```

```
		s_out5<=s5;
		s_out6<=s6;
		s_out7<=s7;
		s_out8<=s8;
	end

	else if(en1==1 && en==1 && en0==1)
	begin
	//retrieving data from ROMs
		a0 <= rom_p0[addr+count];
		b0 <= rom_r0[addr+count];
		a1 <= rom_p1[addr+count0];
		b1 <= rom_r1[addr+count0];
		a2 <= rom_p2[addr+count1];
		b2 <= rom_r2[addr+count1];

		//up counters
		count<=count+1;
		 count0<=count;		//establishing a buffer
		count1<=count0;		//establishing a buffer

		// connecting the output wires to reg to store the values.
		s_out0<=s0;
		s_out1<=s1;
		s_out2<=s2;
		s_out3<=s3;
		s_out4<=s4;
		s_out5<=s5;
		s_out6<=s6;
		s_out7<=s7;
		s_out8<=s8;
	end

		else
		//the other cases of en pin, did all the things zero to prevent the 'X' don't care or
undefined condition.
		begin
		count=0;
		s_out0<=0;
		s_out1<=0;
		s_out2<=0;
		s_out3<=0;
		s_out4<=0;
		s_out5<=0;
		s_out6<=0;
		s_out7<=0;
		s_out8<=0;
```

```
                count0<=count;          //establishing a buffer
                count1<=count0;         //establishing a buffer
                a0 <= 0;
                b0 <= 0;
                a1 <= 0;
                b1 <= 0;
                a2 <= 0;
                b2 <= 0;


        end
        end
else

begin
                a0 <= 0;
                b0 <= 0;
                a1 <= 0;
                b1 <= 0;
                a2 <= 0;
                b2 <= 0;

                count<=0;
                count0<=0;
                count1<=0;

                en0<=0;
                en1<=0;

                s_out0<=0;
                s_out1<=0;
                s_out2<=0;
                s_out3<=0;
                s_out4<=0;
                s_out5<=0;
                s_out6<=0;
                s_out7<=0;
                s_out8<=0;

end
        end


        // module instantiation using generate statement
        generate
        re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_1(.clk(clk),.en(en),.rst(rst),.a(a0),.b(b0),.s(s0),.p(p0),.q(q0));
```

```
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_2(.clk(clk),.en(en),.rst(rst),.a(a1),.b(q0),.s(s1),.p(p1),.q(q1));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_3(.clk(clk),.en(en),.rst(rst),.a(a2),.b(q1),.s(s2),.p(p2),.q(q2));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_4(.clk(clk),.en(en),.rst(rst),.a(p0),.b(b1),.s(s3),.p(p3),.q(q3));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_5(.clk(clk),.en(en),.rst(rst),.a(p1),.b(q3),.s(s4),.p(p4),.q(q4));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_6(.clk(clk),.en(en),.rst(rst),.a(p2),.b(q4),.s(s5),.p(p5),.q(q5));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_7(.clk(clk),.en(en),.rst(rst),.a(p3),.b(b2),.s(s6),.p(p6),.q(q6));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_8(.clk(clk),.en(en),.rst(rst),.a(p4),.b(q6),.s(s7),.p(p7),.q(q7));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_9(.clk(clk),.en(en),.rst(rst),.a(p5),.b(q7),.s(s8),.p(p8),.q(q8));
            endgenerate


endmodule
```

3. Systolic Implementation on FPGA

```
// systolic implementation with RESET pin (FPGA)
//addr stands for address
//clk stands for clock and en stands for enable pin

module sys_3x3_reset
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=2)
//ADDR_WIDTH is the width of the address of the data stored in the memory
//DATA_WIDTH is the width of the data stored in the memory

(input clk, rstn);

            reg [(ADDR_WIDTH-1):0] addr;
            wire [DATA_WIDTH-1:0]a0;
            wire  [DATA_WIDTH-1:0]b0;
            wire [DATA_WIDTH-1:0]a1;
            wire  [DATA_WIDTH-1:0]b1;
            wire [DATA_WIDTH-1:0]a2;
            wire  [DATA_WIDTH-1:0]b2;
            reg [ADDR_WIDTH-1:0]count;
            reg [ADDR_WIDTH-1:0]count0;
            reg [ADDR_WIDTH-1:0]count1;
            reg [2*(DATA_WIDTH)+1:0] s_out0, s_out1, s_out2, s_out3, s_out4, s_out5, s_out6, s_out7,
s_out8;
            reg  en,en0,en1;
```

```
//declaring output wires
 wire [2*(DATA_WIDTH)+1:0] s0;
 wire [2*(DATA_WIDTH)+1:0] s1;
 wire [2*(DATA_WIDTH)+1:0] s2;
 wire [2*(DATA_WIDTH)+1:0] s3;
 wire [2*(DATA_WIDTH)+1:0] s4;
 wire [2*(DATA_WIDTH)+1:0] s5;
 wire [2*(DATA_WIDTH)+1:0] s6;
 wire [2*(DATA_WIDTH)+1:0] s7;
 wire [2*(DATA_WIDTH)+1:0] s8;

 //declaring cell interconnecting wires
 wire  [DATA_WIDTH-1:0]p0;
 wire  [DATA_WIDTH-1:0]p1;
 wire  [DATA_WIDTH-1:0]p2;
 wire  [DATA_WIDTH-1:0]p3;
 wire  [DATA_WIDTH-1:0]p4;
 wire  [DATA_WIDTH-1:0]p5;
 wire  [DATA_WIDTH-1:0]p6;
 wire  [DATA_WIDTH-1:0]p7;
 wire  [DATA_WIDTH-1:0]p8;

 wire  [DATA_WIDTH-1:0]q0;
 wire  [DATA_WIDTH-1:0]q1;
 wire  [DATA_WIDTH-1:0]q2;
 wire  [DATA_WIDTH-1:0]q3;
 wire  [DATA_WIDTH-1:0]q4;
 wire  [DATA_WIDTH-1:0]q5;
 wire  [DATA_WIDTH-1:0]q6;
 wire  [DATA_WIDTH-1:0]q7;
 wire  [DATA_WIDTH-1:0]q8;

	// Declare the ROM variable
	reg [DATA_WIDTH-1:0] rom_p0[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r0[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_p1[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r1[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_p2[2**ADDR_WIDTH-1:0];
	reg [DATA_WIDTH-1:0] rom_r2[2**ADDR_WIDTH-1:0];

	wire rst;

	assign rst = ~rstn;
	always @ (posedge clk)
	begin
	addr=0;
	if(rst==0)
```

```
begin
en<=1;

en0<=en;        //creating buffer of 1 clock cycle
en1<=en0;       //creating buffer of 2 clock cycle

if(en==1 && en0==0 && en1==0)
begin


        count<=count+1;
        count0<=count;
        count1<=count0;
        s_out0<=s0;
        s_out1<=s1;
        s_out2<=s2;
        s_out3<=s3;
        s_out4<=s4;
        s_out5<=s5;
        s_out6<=s6;
        s_out7<=s7;
        s_out8<=s8;
        end

else if(en1==0 && en0==1 && en==1)
begin


        //up counters
        count<=count+1;
        count0<=count;          //establishing a buffer
        count1<=count0;         //establishing a buffer

        // connecting the output wires to reg to store the values.
        s_out0<=s0;
        s_out1<=s1;
        s_out2<=s2;
        s_out3<=s3;
        s_out4<=s4;
        s_out5<=s5;
        s_out6<=s6;
        s_out7<=s7;
        s_out8<=s8;
end

else if(en1==1 && en==1 && en0==1)
begin
```

```
                    //up counters
                    count<=count+1;
            count0<=count;                  //establishing a buffer
                    count1<=count0;         //establishing a buffer



// connecting the output wires to reg to store the values.
                    s_out0<=s0;
                    s_out1<=s1;
                    s_out2<=s2;
                    s_out3<=s3;
                    s_out4<=s4;
                    s_out5<=s5;
                    s_out6<=s6;
                    s_out7<=s7;
                    s_out8<=s8;
          end

                    else

                    //the other cases of en pin, did all the things zero to prevent the 'X' don't care or
undefined condition.
                    begin
                    count=0;
                    s_out0<=0;
                    s_out1<=0;
                    s_out2<=0;
                    s_out3<=0;
                    s_out4<=0;
                    s_out5<=0;
                    s_out6<=0;
                    s_out7<=0;
                    s_out8<=0;

                    count0<=count;
                    count1<=count0;
                    end
                    end

                    else                //when rst pin is 1 , everything should be null
                    begin
                    count<=0;
                    count0<=0;
                    count1<=0;

                    en0<=0;
                    en1<=0;
```

```
                        s_out0<=0;
                        s_out1<=0;
                        s_out2<=0;
                        s_out3<=0;
                        s_out4<=0;
                        s_out5<=0;
                        s_out6<=0;
                        s_out7<=0;
                        s_out8<=0;
                        end
            end

            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_1(.clk(clk),.en(en),.rst(rst),.a(a0),.b(b0),.s(s0),.p(p0),.q(q0));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_2(.clk(clk),.en(en),.rst(rst),.a(a1),.b(q0),.s(s1),.p(p1),.q(q1));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_3(.clk(clk),.en(en),.rst(rst),.a(a2),.b(q1),.s(s2),.p(p2),.q(q2));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_4(.clk(clk),.en(en),.rst(rst),.a(p0),.b(b1),.s(s3),.p(p3),.q(q3));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_5(.clk(clk),.en(en),.rst(rst),.a(p1),.b(q3),.s(s4),.p(p4),.q(q4));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_6(.clk(clk),.en(en),.rst(rst),.a(p2),.b(q4),.s(s5),.p(p5),.q(q5));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_7(.clk(clk),.en(en),.rst(rst),.a(p3),.b(b2),.s(s6),.p(p6),.q(q6));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_8(.clk(clk),.en(en),.rst(rst),.a(p4),.b(q6),.s(s7),.p(p7),.q(q7));
            re_cell_reset #(.DATA_WIDTH(DATA_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
cell_9(.clk(clk),.en(en),.rst(rst),.a(p5),.b(q7),.s(s8),.p(p8),.q(q8));

  ROM_1 u0 (.q(a0),               //  output,  width = 8,      q.dataout
     .address (count),            //  input,  width = 2, address.address
     .clock   (clk),              //  input,  width = 1,   clock.clk
     .rden    (en)                //  input,  width = 1,    rden.rden
  );
  ROM_2 u1 (.q(b0),               //  output,  width = 8,      q.dataout
     .address (count),            //  input,  width = 2, address.address
     .clock   (clk),              //  input,  width = 1,   clock.clk
     .rden    (en)                //  input,  width = 1,    rden.rden
  );

  ROM_3 u2 (
     .q      (a1),                //  output,  width = 8,      q.dataout
     .address (count0),           //  input,  width = 2, address.address
     .clock   (clk),              //  input,  width = 1,   clock.clk
     .rden    (en0)               //  input,  width = 1,    rden.rden
```

```
    );

    rom_4 u3 (
      .q      (b1),                    //  output,  width = 8,      q.dataout
      .address (count0),              //   input,  width = 2, address.address
      .clock   (clk),                 //   input,  width = 1,   clock.clk
      .rden    (en0)                  //   input,  width = 1,    rden.rden
    );

    ROM_5 u4 (
      .q      (a2),                   //  output,  width = 8,      q.dataout
      .address (count1),              //   input,  width = 2, address.address
      .clock   (clk),                 //   input,  width = 1,   clock.clk
      .rden    (en1)                  //   input,  width = 1,    rden.rden
    );

    ROM_6 u5 (
      .q      (b2),                   //  output,  width = 8,      q.dataout
      .address (count1),              //   input,  width = 2, address.address
      .clock   (clk),                  //   input,  width = 1,   clock.clk
      .rden    (en1)                   //   input,  width = 1,    rden.rden
    );

endmodule
```

REFERENCES

[1] Ming-Bo Lin, “Digital System Designs and Practices using Verilog HDL and FPGA’s”.

[2] Henry Y. H. Chuang and Guo He presented a paper on “A Versatile Systolic Array for Matrix Computations”.

[3] Mark A. Richards, James A. Scheer, William A. Holm, “Principles of Modern Radar Basic Principles”.