

**CENTER FOR DEVELOPMENT OF**

**ADVANCED COMPUTING**

**SUMMER INTERNSHIP REPORT**

**SUBMITTED BY**

**AKASH SONOWAL**

**CHAYANDEEP CHAKRABORTY**

**SHOUVIK PAUL**

**UNDER THE SUPERVISION OF**

**PRASHANTHI PATHIPATI**

**DECLARATION**

I hereby declare that the contents in this report are correct up to our knowledge and have been put down after research analysis.

Date:

Place: (Signature)

**ACKNOWLEDGEMENT**

I would like to express my deepest appreciation to all those who provided us the possibility to complete this report. I am highly indebted to\_\_\_\_\_\_\_\_\_\_\_\_\_, Center for Development of Advanced Computing (C-DAC), Silchar for building the foundation of the project and also for her/his full effort in guiding the team in achieving the goal as well as encouragement to maintain our progress in track.

Date:

Place: (Signature)

**CERTIFICATE**

This is to certify that \_\_\_\_\_\_\_\_\_\_\_\_has successfully completed the project title “\_\_\_\_\_\_\_\_\_\_\_\_” under the guidance of \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ at Center for Development of Advanced Computing, Silchar.

This Project Report is the proof and record the bonafide research work carried out by her/his.

(Signature)

Project Engineer

Center for Development of Advanced Computing, Silchar.

**CONTENTS**

Chapter 1. VERILOG 1-8

* 1. Introduction of Verilog.
  2. Design flow of Verilog.
  3. Vivado Software.

1.4 PYNQ-Z2 Board.

Chapter 2. PROJECT OBJECTIVE 9

2.1 Code dumping in pynq-z2 board.

2.2 RAMAN Algorithm.

Chapter 3. CODE DUMP 10-24

3.1 Full Adder

3.2 N-bit parallel Adder.

3.3 Binary to Gray converter.

3.4 J-K flip flop.

Chapter 4. RAMAN ALGORITHM 25-50

4.1 Introduction.

4.2 Algorithm.

4.3 Module of Raman Algorithm.

4.4 Code.

4.5 Output.

Chapter 5. Results And Conclusion. 51 -56

5.1 Project snapshot

5.2 Conclusion.

Chapter 6. References. 57

| **VERILOG** |
| --- |

**Verilog is a hardware description language (HDL) used in digital hardware design and verification. It was originally developed by Gateway Design Automation in the early 1980s and has since become one of the most widely used HDLs in the electronic design industry. Verilog provides engineers with a powerful means to model, simulate, and implement digital circuits and systems, making it an essential tool for hardware designers and verification engineers.**

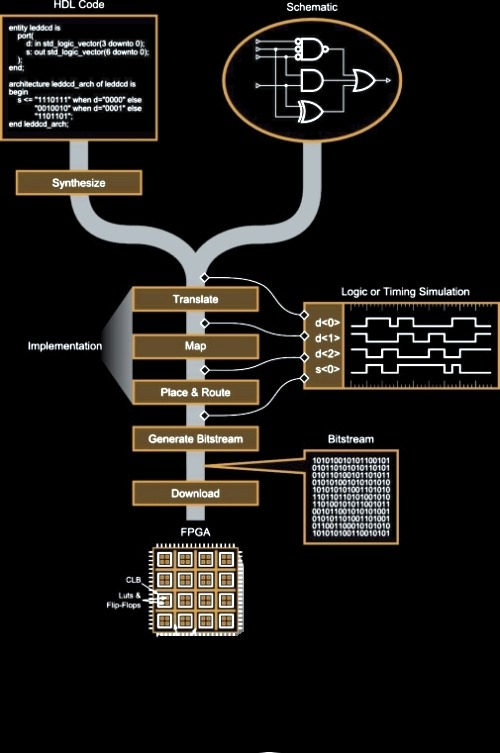
**Key Features and Applications:**

* **Behavioral Modelling :** Verilog allows engineers to describe the behavior of a digital system at a high level of abstraction, enabling them to focus on functionality and logic rather than the specific hardware implementation.
* **Modularity :** Verilog promotes a modular design approach, wherein complex systems are built by assembling smaller, reusable modules. This modularity enhances design efficiency and facilitates design reuse.
* **Simulation :** Verilog is extensively used for functional simulation of digital designs. Engineers can verify the correctness and performance of their circuits by simulating them using testbench environments.
* **Synthesis :** Verilog is not only a simulation language but also a hardware synthesis language. Designers can synthesize Verilog code to obtain the corresponding gate-level representation, which can be physically implemented in hardware.
* **FPGA and ASIC Design :** Verilog plays a crucial role in designing Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). FPGA designs can be realized through synthesis, while ASICs can be fabricated from the gate-level netlist.
* **Verification :** One of the essential aspects of hardware design is verification, ensuring that the circuit behaves as intended. Verilog enables engineers to create detailed testbenches to verify the functionality and performance of their designs.
* **Digital Systems :** Verilog is applied in various digital systems, including microprocessors, graphics processing units (GPUs), digital signal processors (DSPs), networking hardware, and other complex digital circuits.

| DESIGN FLOW OF VERILOG |
| --- |

The typical Verilog design flow involves the following steps:

* **Design Specification:** Define the functional behavior and architecture of the digital system.
* **Verilog Coding:** Write the Verilog code to model the design's behavior and structure using modules and logic gates.
* **Functional Simulation:** Verify the correctness of the design by simulating it using testbenches.
* **Synthesis:** Convert the Verilog code into a gate-level representation suitable for implementation in hardware.
* **Implementation:** Physical implementation of the design using FPGAs or ASICs.
* **Verification and Testing:** Verify the functionality of the hardware using various verification methodologies and test vectors.



**FIGURE \_1 Verilog Design Flow**

| **APPLICATIONS AND CONCLUSION** |
| --- |

**Verilog has significantly contributed to the advancement of digital hardware design by providing a flexible and efficient means to model and implement complex digital systems. Its adoption in FPGA and ASIC design, along with its simulation capabilities, has made Verilog a fundamental language in the field of digital hardware engineering. As the semiconductor industry continues to evolve, Verilog remains a crucial tool for designing innovative and cutting-edge electronic systems.**

| **VIVADO SOFTWARE** |
| --- |

**Vivado is a powerful and comprehensive design environment developed by Xilinx, a leading FPGA (Field-Programmable Gate Array) manufacturer. It serves as an essential tool for FPGA and SoC (System on Chip) design, providing hardware engineers with a complete platform for design entry, simulation, synthesis, implementation, and verification.**

**Key Features:**

* **Design Entry :** Vivado offers multiple design entry methods, including graphical and text-based options, allowing users to describe their circuits using either block diagrams or Hardware Description Languages (HDLs) like Verilog or VHDL.
* **High-Level Synthesis :** Vivado supports high-level synthesis, enabling designers to express algorithms in C, C++, or SystemC and automatically convert them into hardware implementations, increasing productivity and accelerating design time.
* **IP Integrator :** With the IP Integrator tool, Vivado simplifies the process of integrating and configuring IP cores into FPGA designs. This drag-and-drop feature streamlines the development of complex systems.
* **System-Level Debugging :** Vivado provides advanced debugging features at both the RTL (Register Transfer Level) and system level. This helps engineers identify and resolve design issues more efficiently.
* **Timing Analysis and Closure :** The software includes powerful timing analysis and optimization tools to ensure that the design meets the required performance and timing constraints.
* **Partial Reconfiguration :** Vivado supports partial reconfiguration, allowing specific regions of the FPGA to be reconfigured while the rest of the system remains operational. This feature is particularly useful in dynamic and resource-constrained applications.
* **Programming and Debugging :** Vivado supports programming and debugging of Xilinx FPGAs through various methods, including JTAG, USB, and Ethernet.

| **APPLICATIONS** |
| --- |

**Vivado is extensively used in a wide range of applications, including but not limited to:**

* **Digital signal processing (DSP)**
* **Video and image processing**
* **Aerospace and defense systems**
* **High-performance computing (HPC)**
* **Communications and networking**
* **Embedded systems and IoT (Internet of Things) devices**



**FIGURE\_2 Vivado Software.**

| **PYNQ\_Z2 BOARD** |
| --- |

**The PYNQ-Z2 board is an open-source development platform designed for embedded systems and digital design projects. It is part of the PYNQ (Python Productivity for Zynq) ecosystem developed by Xilinx, which combines the flexibility of Python programming with the power of Zynq All-Programmable SoCs (System on Chips).**

**Key Features :**

* **Zynq SoC :** The PYNQ-Z2 board features a Xilinx Zynq-7000 SoC, combining a dual-core ARM Cortex-A9 processor with an FPGA fabric. This integration allows users to leverage the capabilities of both processors and programmable logic.
* **Python Productivity :** PYNQ-Z2 enables developers to work with Python, a high-level and user-friendly programming language. Python libraries facilitate the configuration and control of the FPGA, making hardware design more accessible to software developers.
* **Open-Source Framework :** PYNQ-Z2 is built on an open-source framework, providing access to the board's hardware specifications, reference designs, and community-driven support. This fosters collaboration and encourages innovation among developers.
* **Rich Set of I/O :** The board offers various I/O interfaces, including HDMI, USB, Ethernet, audio ports, and multiple PMOD (Peripherals Module) connectors, enabling easy integration with a wide range of peripherals.
* **On-Board Memory :** PYNQ-Z2 features DDR3 memory, Flash memory for booting, and SD card slots, providing ample space for storing applications and data.
* **Jupyter Notebooks :** PYNQ-Z2 leverages Jupyter Notebooks, an interactive web-based environment, to develop and document projects. This enables users to mix code, visualizations, and text explanations for better project understanding and collaboration.

| **APPLICATIONS** |
| --- |

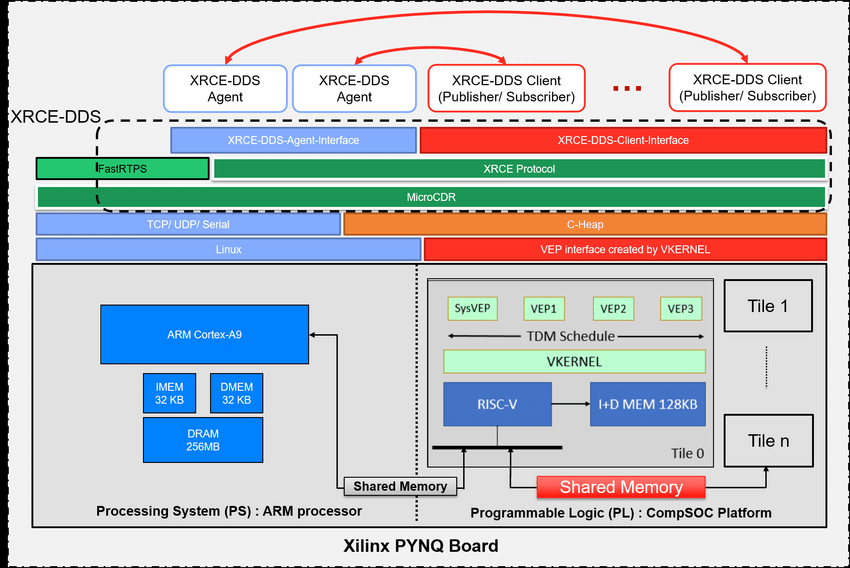
**The PYNQ-Z2 board is popular for a variety of applications, including:**

* **Rapid prototyping of digital systems and hardware accelerators.**
* **Machine learning and artificial intelligence (AI) applications with the FPGA's parallel processing capabilities.**
* **Image and video processing.**
* **IoT (Internet of Things) projects.**
* **Robotics and automation.**

|  |
| --- |

**FIGURE\_3 PYNQ Z2 BOARD.**

| **ARCHITECTURE OF PYNQ-Z2 BOARD** |
| --- |



**FIGURE\_4 ARCHITECTURE OF PYNQ-Z2 BOARD .**

| **CODE DUMP IN FPGA** |
| --- |

**"Code Dump" typically refers to the process of loading a digital design (such as a hardware description written in a Hardware Description Language like Verilog or VHDL) onto an FPGA (Field-Programmable Gate Array) board to configure it and make it perform the desired functionality.**

**To perform a "Code Dump" on an FPGA using Verilog, we need to follow these general steps :**

* **Write Verilog Code :** First, write the hardware description code in Verilog that describes the functionality of your digital design. This code will specify the behavior and structure of your circuit.
* **Compile the Verilog Code :** Use the FPGA vendor's design software (e.g., Vivado for Xilinx FPGAs or Quartus for Intel FPGAs) to compile the Verilog code. This process synthesizes the Verilog code into a hardware netlist, which represents the actual gates and flip-flops used to implement your design on the FPGA.
* **Create a Constraints File :** Create a constraints file (e.g., XDC for Xilinx FPGAs or SDC for Intel FPGAs) that defines the physical mapping of your design to the FPGA's pins and other constraints such as clock frequencies and timing requirements.
* **Connect the FPGA Board :** Connect your FPGA board to the host computer using the appropriate programming cable (e.g., JTAG or USB).
* **Load the Configuration :** Use the FPGA vendor's programming software to load the generated bitstream (binary configuration file) onto the FPGA. This process configures the FPGA with your design.
* **Configuration Verification :** After the configuration is loaded, the FPGA will start executing the design according to the Verilog code that was dumped. Verify that the FPGA behaves as expected and performs the desired functionality.

| **CODE DUMP OF FULL ADDER IN PYNQ\_Z2 BOARD** |
| --- |

**A Full Adder is a combinational logic circuit that performs addition on three binary inputs: A, B, and an input carry (C\_in). It produces two outputs: the sum (S) and the output carry (C\_out). A Full Adder can be used to add two binary numbers of any length by cascading multiple Full Adders together.**

**The truth table of a Full Adder is as follows:**

| **A** | **B** | **C\_in** | **Sum (S)** | **Carry Out (C\_out)** |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**The Sum output (S) is the result of the binary addition of A, B, and C\_in, while the Carry Out (C\_out) represents the carry generated when adding A, B, and C\_in.**

|  |
| --- |

**FIGURE\_5 BLOCK DIAGRAM OF FULL ADDER** .

|  |
| --- |

**FIGURE\_6 LOGICAL GATE DIAGRAM OF FULL ADDER.**

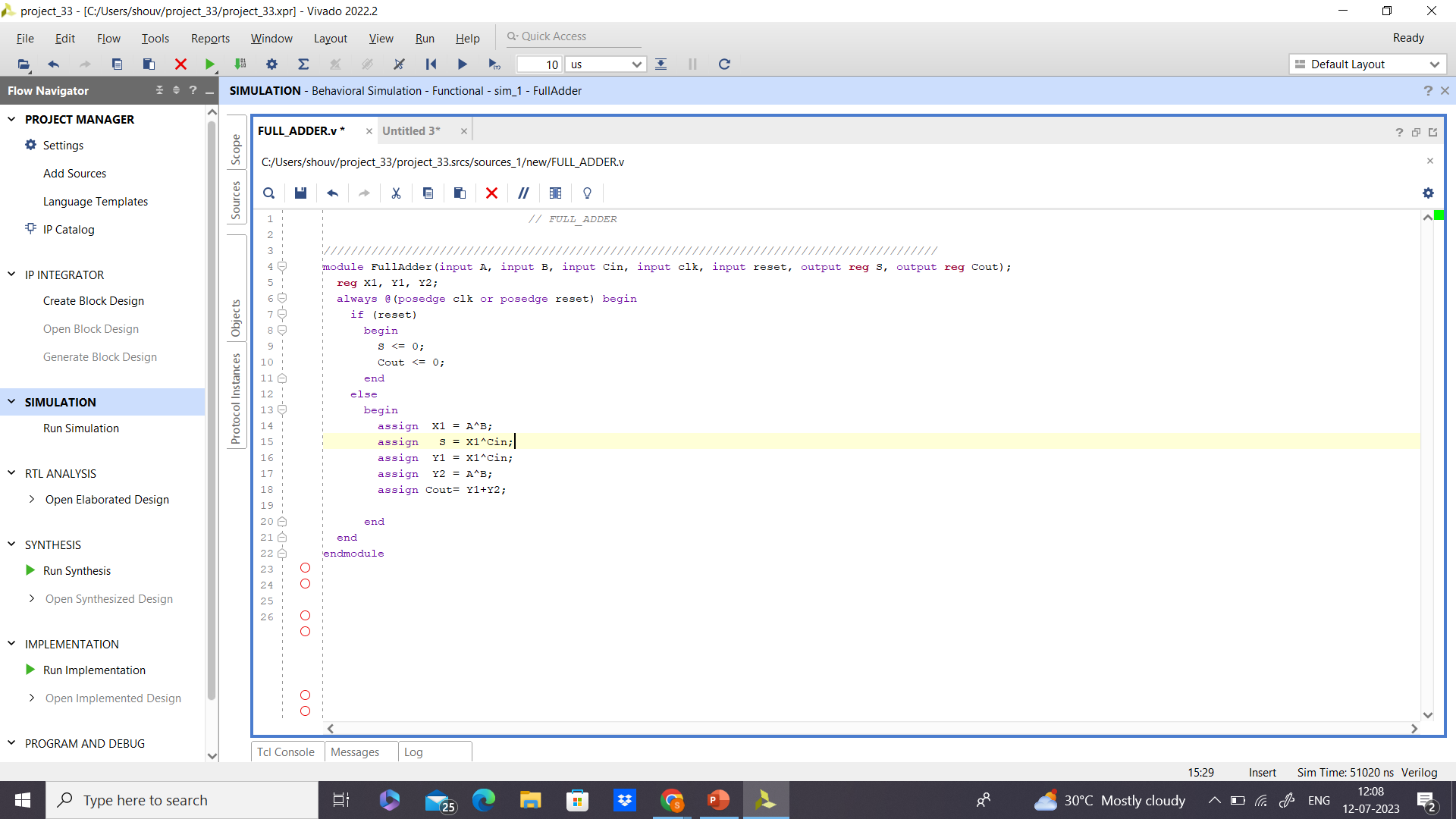
| **VERILOG CODE OF FULL ADDER** |
| --- |

**CODE :**

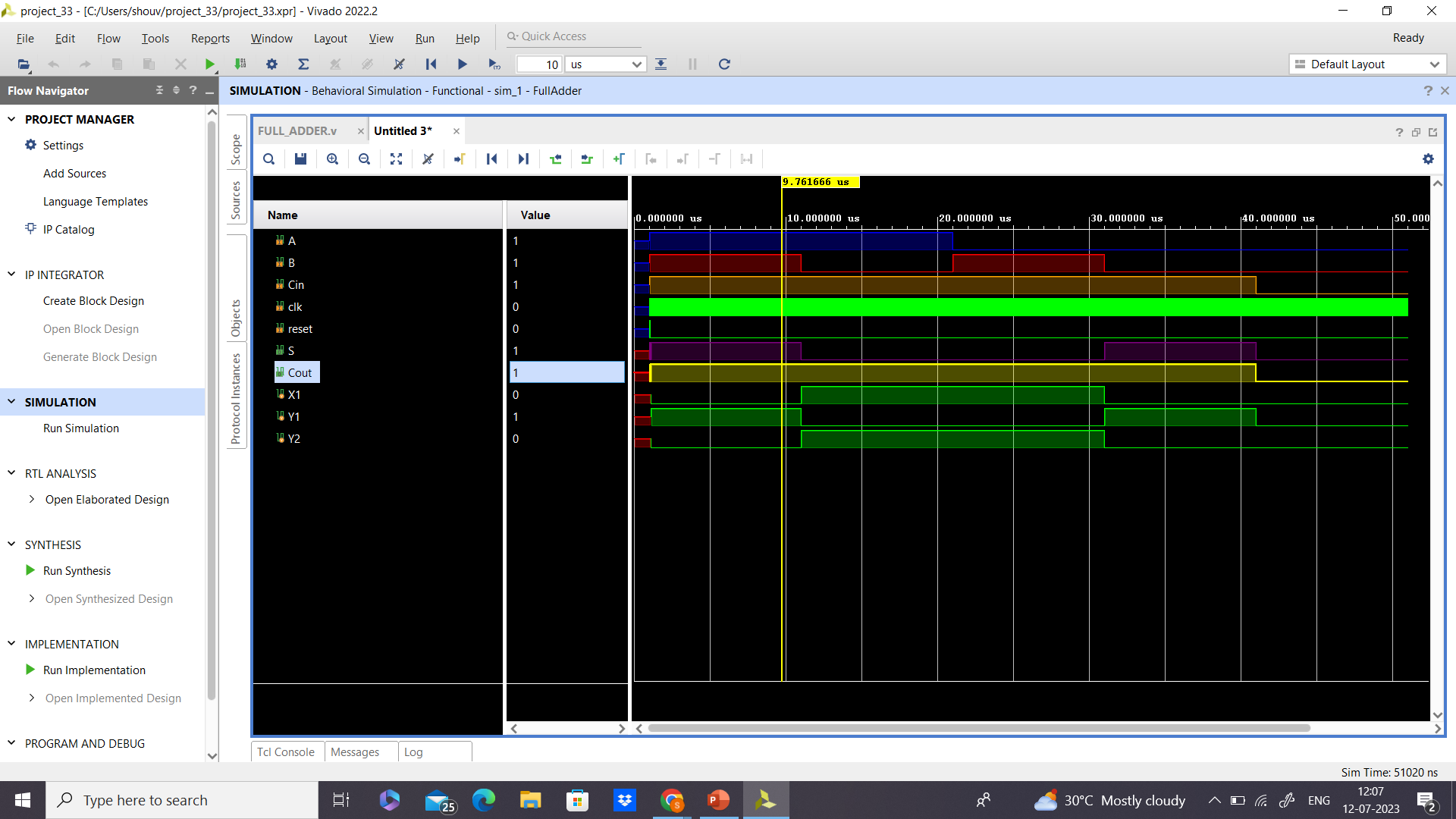
| **module FullAdder(input A, input B, input Cin, input clk, input reset, output reg S, output reg Cout);**  **reg X1, Y1, Y2;**  **always @(posedge clk or posedge reset) begin**  **if (reset)**  **begin**  **S <= 0;**  **Cout <= 0;**  **end**  **else**  **begin**  **assign X1 = A^B;**  **assign S = X1^Cin;**  **assign Y1 = X1^Cin;**  **assign Y2 = A^B;**  **assign Cout= Y1+Y2;**    **end**  **end**  **endmodule** |
| --- |

| **SNAPSHOT OF FULL\_ADDER** |
| --- |

**CODE :**



**OUTPUT :**



| **CONCLUSION** |
| --- |

**A Full Adder can be realized using various logic gates like AND, OR, and XOR gates. It is a fundamental building block in digital circuits, and multiple Full Adders can be cascaded to create adders for multi-bit binary numbers, such as 4-bit, 8-bit, or larger. The Carry Out (C\_out) of one Full Adder becomes the Carry In (C\_in) for the next adder in the cascade, allowing for the addition of multi-bit numbers.**

| **CODE DUMP OF N-BIT\_FULL\_ADDER** |
| --- |

In digital arithmetic, adding two binary numbers requires the use of a Full Adder. When dealing with multi-bit binary numbers, we use an N-Bit Full Adder, which can perform addition on N-bit binary inputs, producing N-bit sums and carry outputs.

**Functionality :**  An N-Bit Full Adder takes N binary inputs from two N-bit numbers (A and B) and an input carry (C\_in), and produces two outputs: the N-bit sum (S) and the output carry (C\_out). The carry output (C\_out) represents the carry generated when adding the N-bit numbers.

**Architecture :** The N-Bit Full Adder is constructed by cascading multiple Full Adders together. Each Full Adder handles a single bit addition and produces a sum bit (S) and a carry-out (C\_out). The carry-out from each Full Adder becomes the carry-in for the next Full Adder in the cascade. This process propagates the carry through the N-Bit adder, enabling the addition of multi-bit binary numbers.

**Truth Table :** The truth table for an N-Bit Full Adder is more extensive than that of a single Full Adder, as it needs to account for all possible combinations of N-bit inputs.

For example, for a 4-Bit Full Adder, the truth table would include 16 rows, covering all possible combinations of 4-bit inputs (A, B, and C\_in), and the corresponding 4-bit sums (S) and the final carry-out (C\_out) for each combination.

**Applications :** N-Bit Full Adders are essential components in various digital arithmetic operations, including addition, subtraction, multiplication, and division. They are used in many digital systems, including microprocessors, digital signal processors (DSPs), FPGAs, and other integrated circuits that require arithmetic computations.

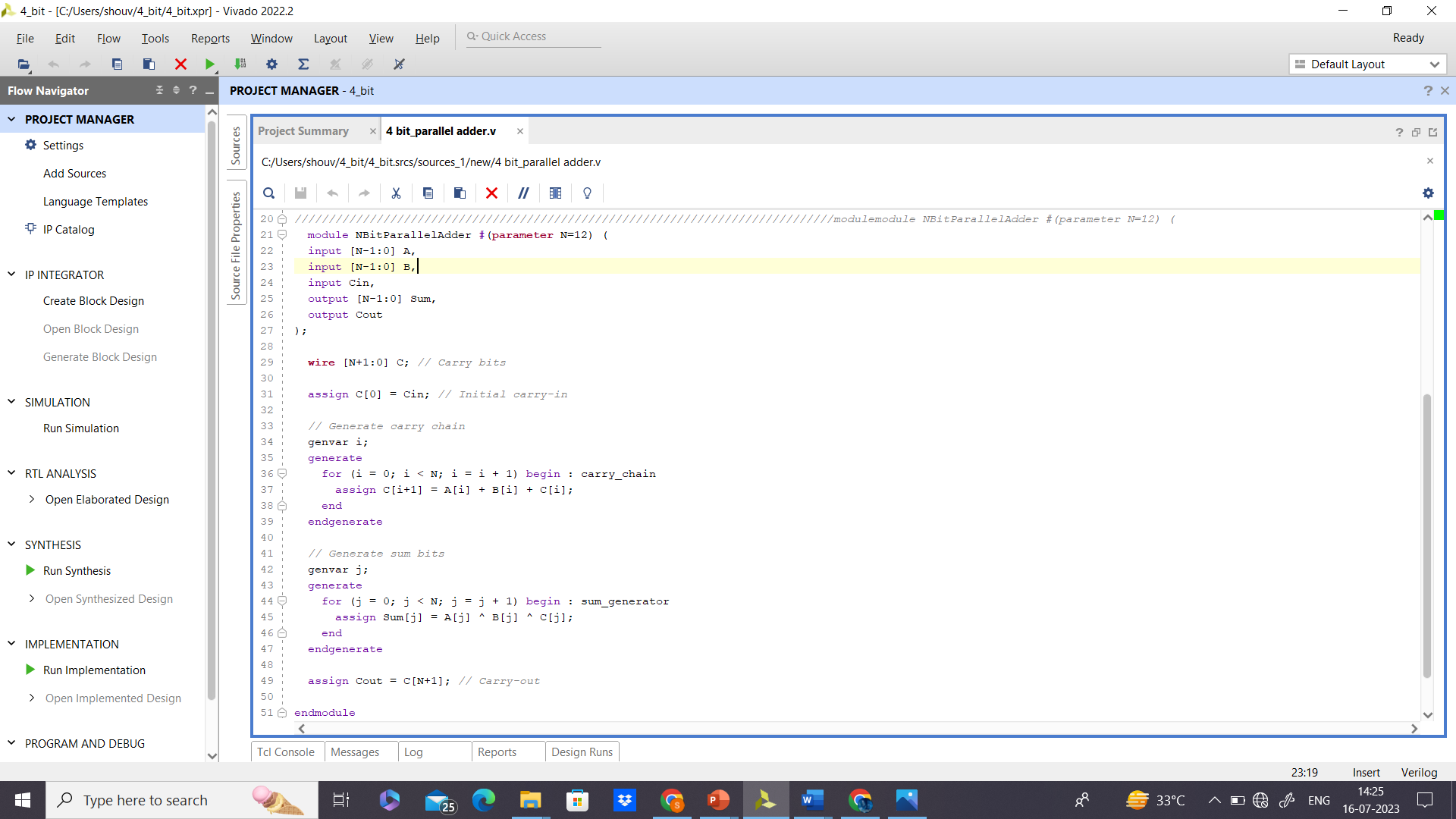
| **VERILOG CODE OF N-BIT\_FULL\_ADDER** |
| --- |

**CODE :**

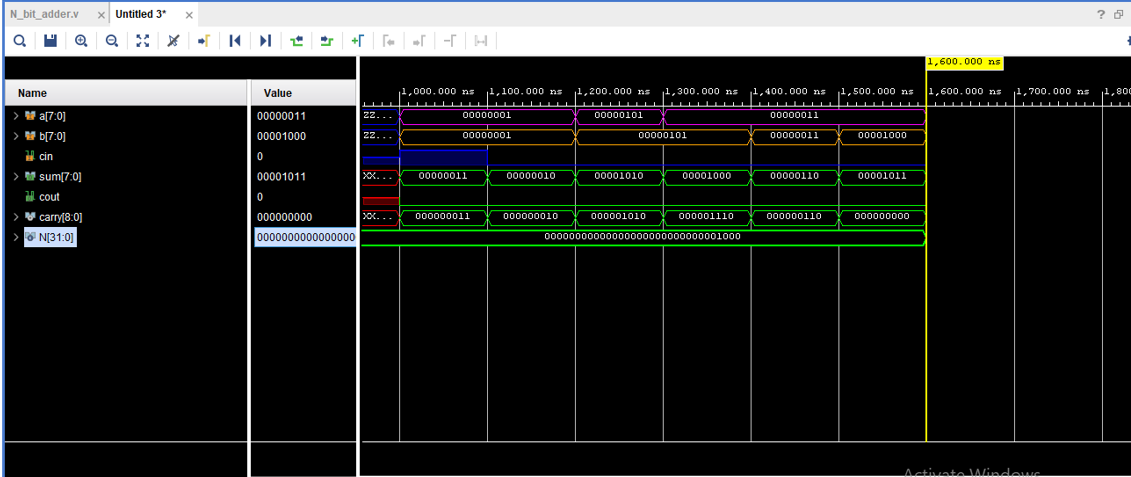
| **module NBitParallelAdder #(parameter N=12) (**  **input [N-1:0] A,**  **input [N-1:0] B,**  **input Cin,**  **output [N-1:0] Sum,**  **output Cout**  **);**    **wire [N+1:0] C; // Carry bits**    **assign C[0] = Cin; // Initial carry-in**    **// Generate carry chain**  **genvar i;**  **generate**  **for (i = 0; i < N; i = i + 1) begin : carry\_chain**  **assign C[i+1] = A[i] + B[i] + C[i];**  **end**  **endgenerate**    **// Generate sum bits**  **genvar j;**  **generate**  **for (j = 0; j < N; j = j + 1) begin : sum\_generator**  **assign Sum[j] = A[j] ^ B[j] ^ C[j];**  **end**  **endgenerate**    **assign Cout = C[N+1]; // Carry-out**    **endmodule** |
| --- |

| **SNAPSHOT OF N-BIT\_FULL\_ADDER** |
| --- |

**CODE :**



**OUTPUT :**



| **CONCLUSION** |
| --- |

**The N-Bit Full Adder is a crucial building block in digital arithmetic, allowing for the addition of multi-bit binary numbers. By cascading individual Full Adders, it can handle N-bit inputs efficiently, producing accurate N-bit sums and carry outputs. Its versatility and significance in digital circuit design make it a fundamental component in many complex digital systems.**

| **CODE DUMP OF BINARY TO GRAY CODE CONVERTER** |
| --- |

**A binary to Gray code converter is a combinational logic circuit that converts a binary number to its corresponding Gray code representation. The Gray code, also known as reflected binary code or unit distance code, is a binary numeral system where successive values differ by only one bit.**

**The conversion from binary to Gray code can be achieved using a simple algorithm that involves XORing adjacent bits of the binary number. Here's how the conversion works for a 4-bit binary number (B3 B2 B1 B0) to its Gray code representation (G3 G2 G1 G0):**

* **G3 = B3**
* **G2 = B3 XOR B2**
* **G1 = B2 XOR B1**
* **G0 = B1 XOR B0**

|  |
| --- |

**FIGURE\_7 BINARY TO GREY USING XOR .**

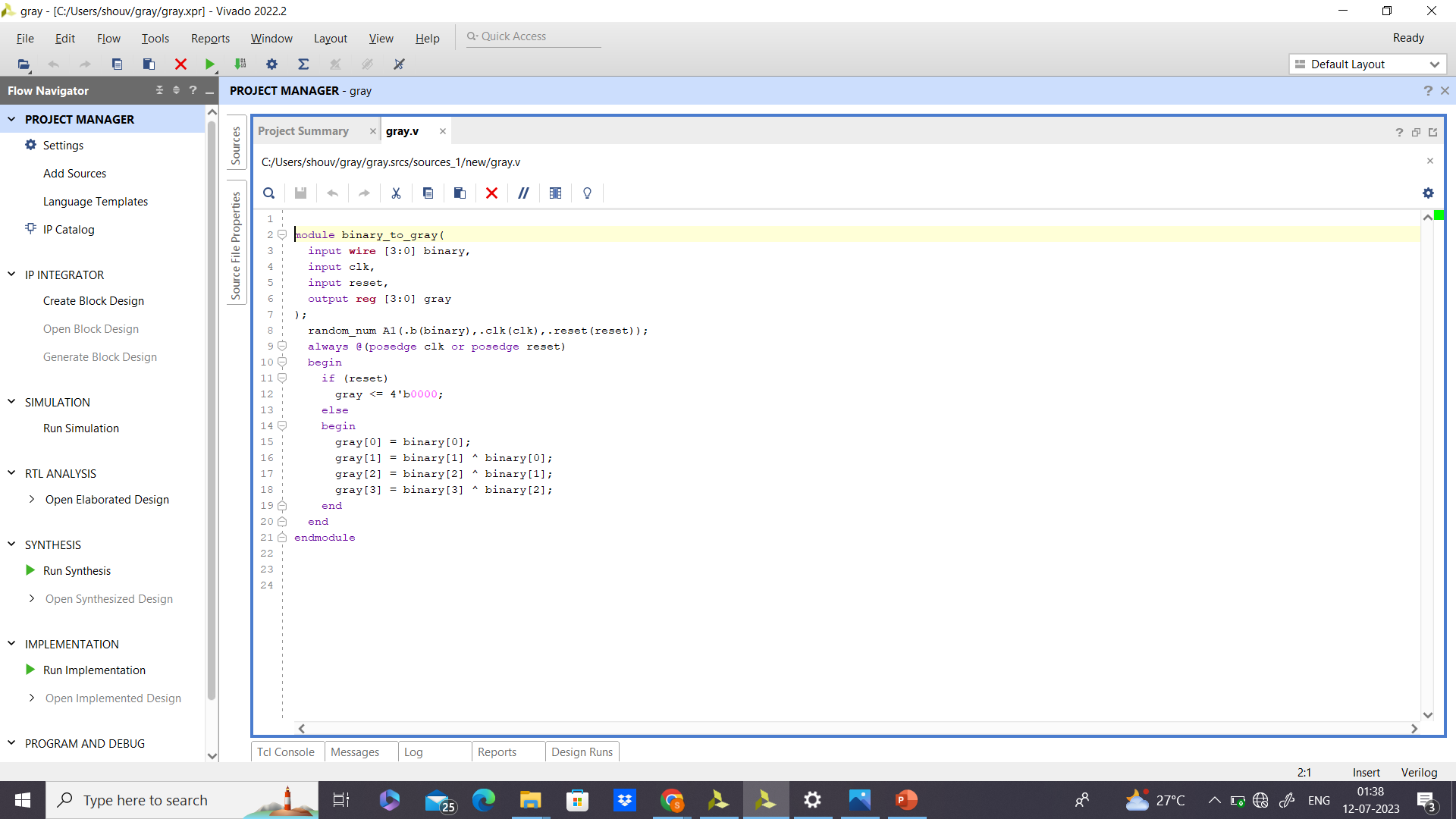
| **BINARY TO GRAY CODE CONVERTER** |
| --- |

**CODE :**

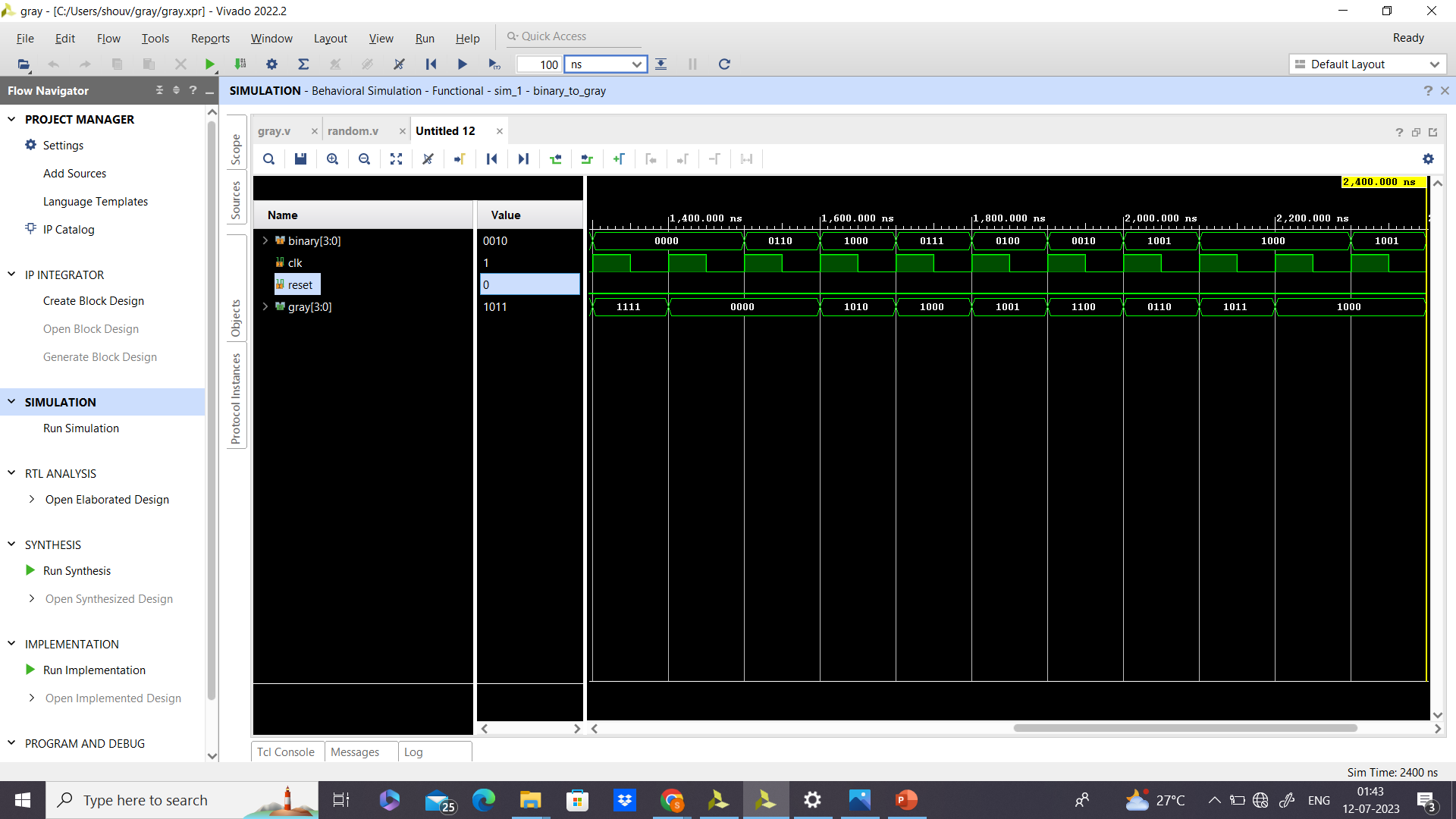
| **module binary to gray(**  **input wire [3:0] binary,**  **input ClK,**  **Input reset,**  **output reg [3:0] gray**  **);**  **random\_num A1(.b (binary), .clk(clk),.reset(reset));**  **always @(posedge clk or posedge reset)**  **begin**  **if (reset)**  **gray <= 4'60000;**  **else**  **begin**  **gray[0] = binary [0];**  **gray [1] = binary [1]^ binary [0];**  **gray [2] = binary [2]^ binary [1];**  **gray[3] = binary[3] ^ binary[2];**  **end**  **end**  **endmodule** |
| --- |

| **SNAPSHOT OF BINARY TO GRAY CONVERTER** |
| --- |

**CODE :**



**OUTPUT :**



| **CODE DUMP OF JK\_FLIP-FLOP** |
| --- |

The JK flip-flop is a sequential logic circuit element that is widely used in digital systems for various applications, such as counters, shift registers, and memory storage. It is an extension of the basic SR (Set-Reset) flip-flop, providing additional functionality and addressing some of the issues faced with the SR flip-flop.

**Functionality :** The JK flip-flop has two inputs, J (Jump) and K (Kill), along with the standard clock input (CLK) and an asynchronous preset (PRE) and clear (CLR) inputs. When the clock input is triggered, the JK flip-flop responds to the J and K inputs based on its current state and the clock edge.

**Operation :** The behavior of a JK flip-flop is defined by the following truth table:

CLK | J | K | Q(t) | Q(t+1)

↑ | 0 | 0 | 0 | 0 (No Change)

↑ | 0 | 0 | 1 | 1 (No Change)

↑ | 0 | 0 | 1 | 1 (No Change)

↑ | 0 | 1 | 0 | 0 (Reset)

↑ | 0 | 1 | 1 | 0 (Reset)

↑ | 1 | 0 | 0 | 1 (Set)

↑ | 1 | 0 | 1 | 1 (Set)

↑ | 1 | 1 | 0 | ~Q (Toggle)

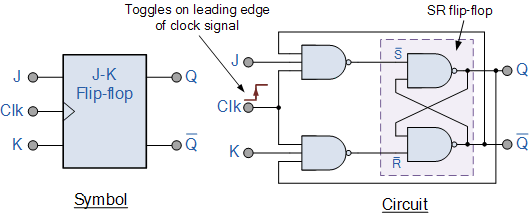
↑ | 1 | 1 | 1 | ~Q (Toggle)

**Characteristics :**

* The JK flip-flop can function as a T (Toggle) flip-flop when J and K are tied together.
* It overcomes the race condition issue of the SR flip-flop when both S and R are 1, ensuring reliable operation.
* JK flip-flops are edge-triggered, meaning they respond to the input changes at a specific clock edge (rising or falling).

**Applications :** JK flip-flops are used in various applications, including:

* Binary counters and frequency dividers.
* Shift registers and data storage elements.
* Digital clocks and frequency generators.
* Control circuits for sequential systems.

 **. . . FIGURE\_8 JK\_FLIP-FLOP.**

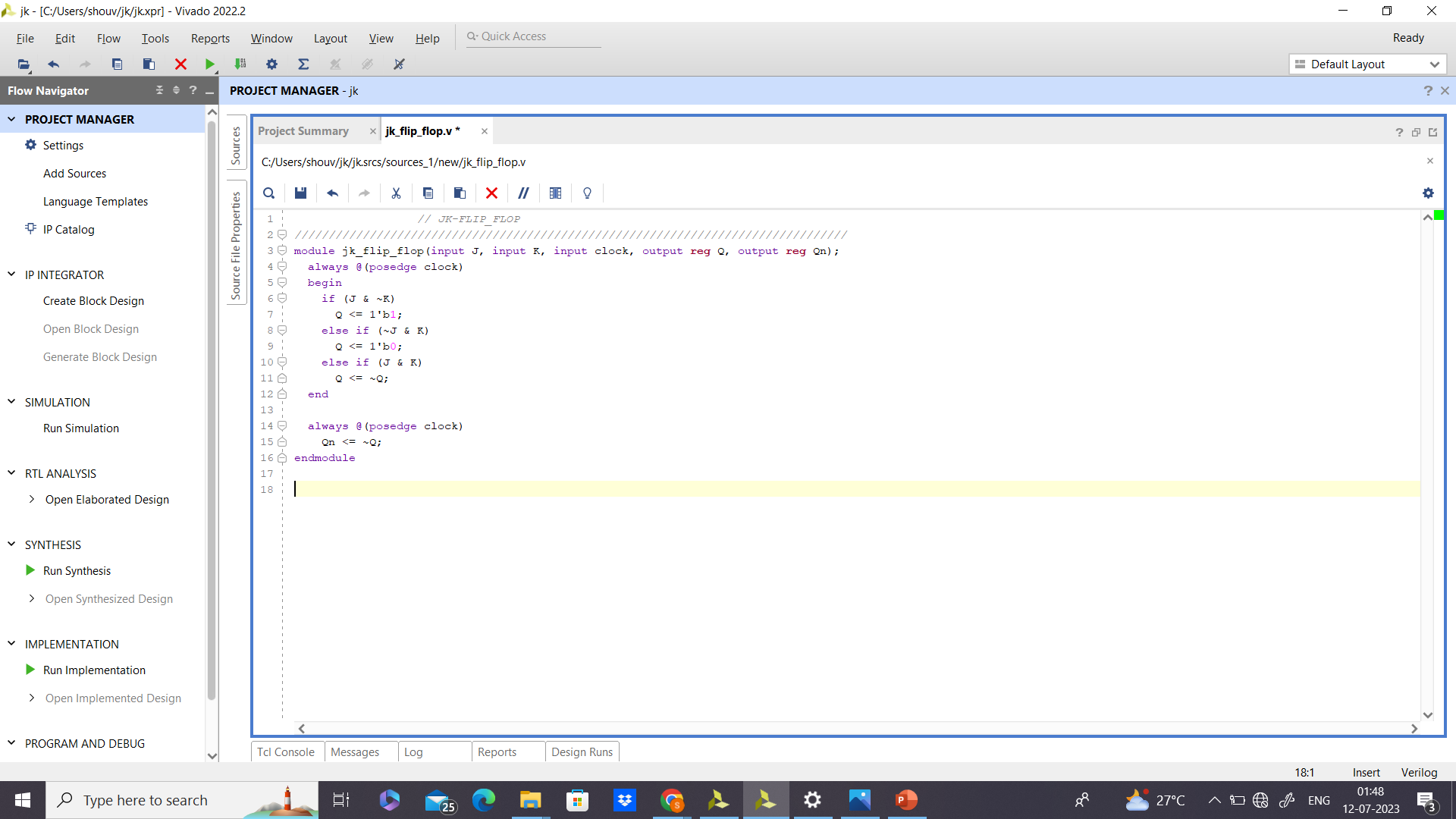
| **JK\_FLIP-FLOP** |
| --- |

**CODE :**

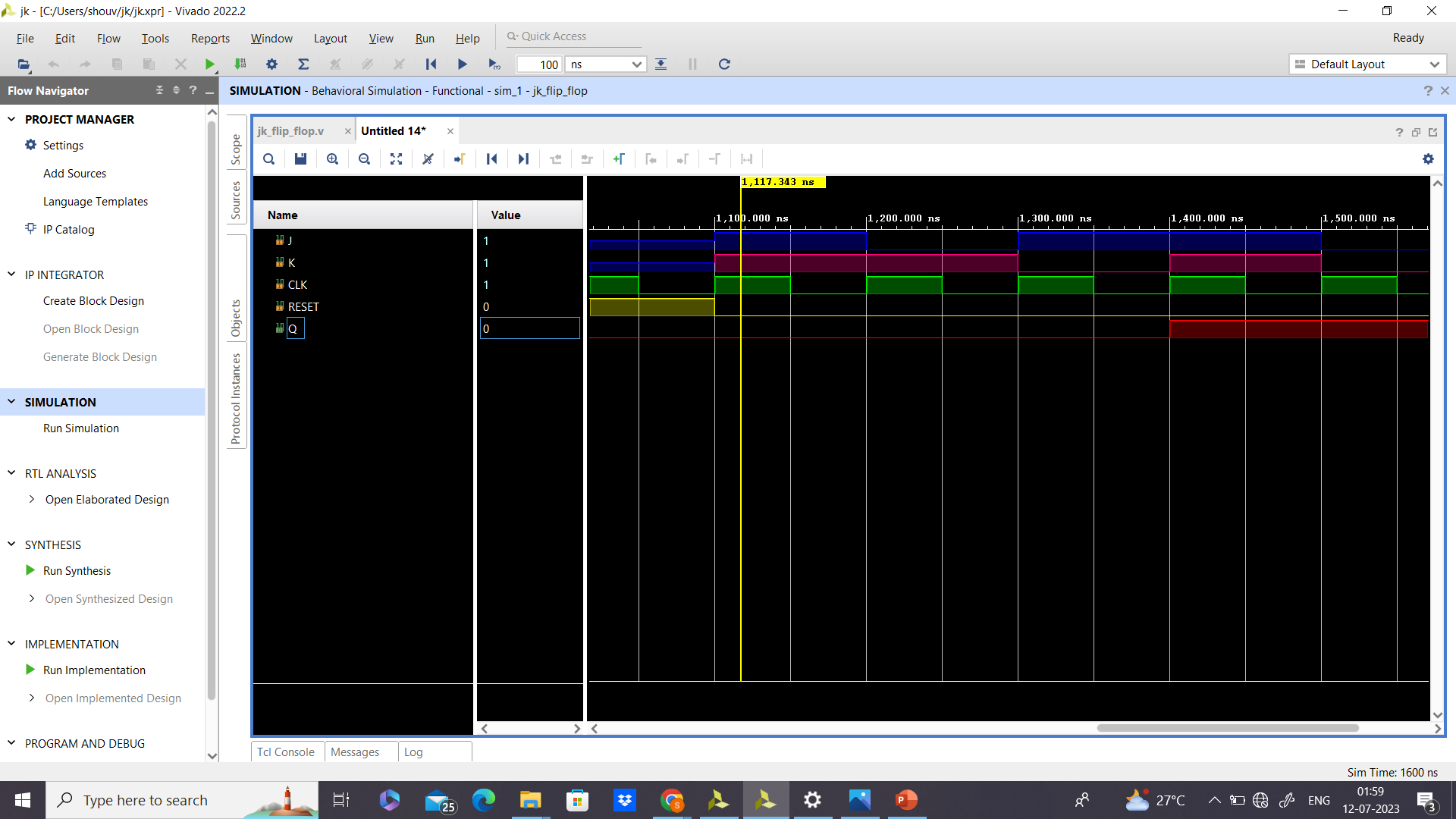
| **module jk\_flip\_flop (input J, input R, input clock, output reg Q , output reg Qn);**  **always @(posedge clock)**  **begin**  **if (J & ~K)**  **Q<=1’b1;**  **else if (~J & K)**  **Q<=1’b0;**  **Else if (J & K)**  **Q<=~Q;**  **end**  **always @(posedge clock)**  **Qn <= ~Q;**  **endmodule** |
| --- |

| **SNAPSHOT OF BINARY TO JK\_FLIP-FLOP** |
| --- |

**CODE :**



**OUTPUT :**



| **CONCLUSION** |
| --- |

**The JK flip-flop is a versatile and widely used sequential logic circuit element that addresses the limitations of the SR flip-flop. Its ability to toggle its output state and its various applications make it an essential component in digital circuit design and digital systems**.

‘RAMAN’ .

ALGORITHM .

| **ABSTRACT** |
| --- |

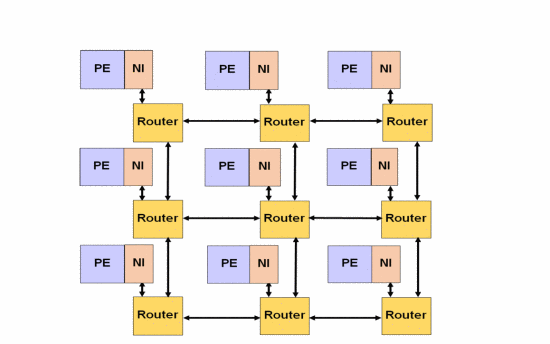
Application Mapping in Network-on-Chip (NoC) design is considered a vital challenge because of its NP-hard nature. Many efforts are made to address the application mapping problem, but none has satisfied all the requirements. For example, Integer Linear Programming (ILP) has achieved the best possible solution but lacks scalability. Advancements in Machine Learning (ML) have added new dimensions in solving the application mapping problem. This paper proposes RAMAN: Reinforcement Learning (RL) inspired algorithm for mapping applications onto mesh NoC. RAMAN is a modified Q-Learning technique inspired by RL, aiming to achieve the minimum communication cost for the application mapping problem. The results of RAMAN demonstrated that RL has enormous potential to solve application mapping problem without much complexity and computational cost. RAMAN has achieved the communication cost within the 6% of the optimal cost determined by ILP. Considering the computational overheads and complexity, the results of RAMAN are encouraging. Future work will improve RAMAN's performance and provide a new aspect to solve the application mapping problem.

| **INTRODUCTION** |
| --- |

**“RAMAN”** , a Reinforcement Learning (RL) inspired mapping algorithm for mesh NoC is developed to simplify the design space search and find optimal mapping at minimal computational cost.

**INDEX TERMS :** Network on Chip , Application Mapping , Q-learning ,

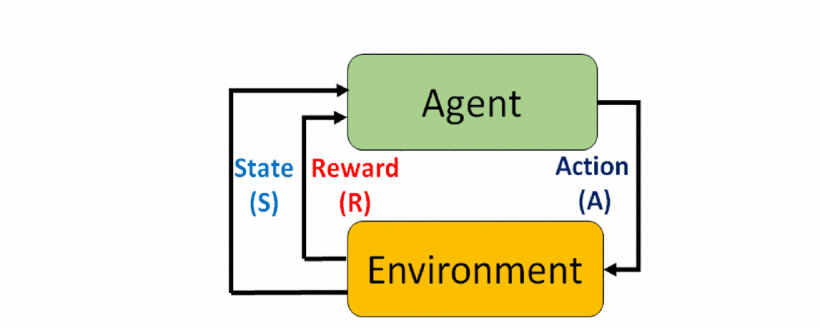
**>** Network-on-Chip (NoC) is an extensible and consistent interconnection paradigm in System-on-Chip (SoC) design. It is a viable alternative to the traditional arbitrated bus architecture to meet the challenges like performance, scalability, and flexibility.



**FIGURE\_9 NOC ARCHITECTURE.**

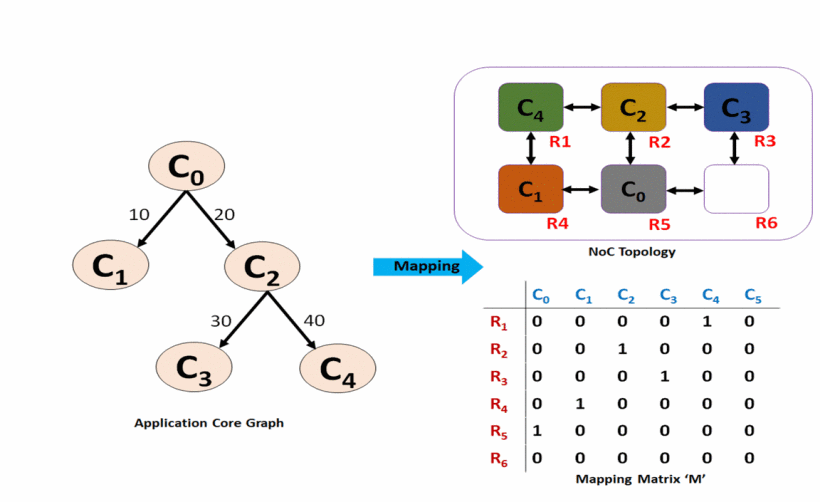
As illustrated in Figure\_9, a typical NoC architecture consists of processing elements (PEs), network interfaces (NIs), links, and routers. ANI module converts data packets from PE into fixed-length flow-control digits (flits). This array of flits is routed toward the intended destination hop-by-hop from one router to another router . As the number of PEs and the data traffic between them continues to grow on SoC, NoC design faces optimization challenges in application mapping, routing, scheduling, power management, reliability, etc.

**>** Application mapping is considered vital among all design processes; it directly impacts communication costs in the network. Communication cost is an efficient evaluation metric in NoC design as it helps in prioritizing communication and reducing congestion and latency of inter-processor communication in multi-core processors . In application mapping, if there are n number of cores, there are n! possible solutions. For higher values of n, there are many combinatorial solutions available, and finding the best solution becomes computationally costly. Hence, application mapping is considered an NP-hard combinatorial optimization (CO) problem and efficiently finding the optimized solution is always an uphill task.



**FIGURE\_10 RL-APPROACH.**

**>**  This work by us explores the possibility of using RL for application mapping with communication cost as the performance metric. Communication cost is the sum of the product of number of bits transferred per second between routers, and number of hops required to reach destination router from source router. Mitigating communication costs can indirectly lead to significant savings in power consumption and network latency . The XY routing algorithm is used to decide the communication path of the data packet and calculate the distance (no. of hops) between source and destination routers. The problem formulation for RAMAN is explained with the following definitions.



**FIGURE\_11 APPLICATION MAPPING.**

### **Definition\_1 :**

ACG is defined as directed graph G(C,A), where each vertex ci∈C denotes an application core, and the edge eij∈E denotes the link between the application cores ci to cj. The weight of the edge eij, represents the communication bandwidth.

### **Definition\_2 :**

TG is defined as mesh topology T(R,L), where each vertex ri∈R denotes a router in the mesh topology and the edge lij∈L denotes the physical link between the routers ri to ri. Mesh topology is a grid like structure where all routers are interconnected with their neighboring routers A mesh topology of 2×3 is shown in Figure\_11.

### **Definition\_3 :**

The bandwidth matrix is denoted as B. The element Wij of the bandwidth matrix represents the communication bandwidth between the nodes ci to cj. If there is no interconnection between ci to ci, the value of Wij is 0.

### **Definition\_4 :**

The distance matrix is denoted as D. The element dij of the distance matrix denotes the shortest routing distance between the routers ri(xi, yi) to rj(xj, yj) of TG. The distance is obtained by using the XY routing algorithm using [(1)](https://ieeexplore.ieee.org/document/#deqn1) [3].

dij=|xi−xj|+|yi−yj|

### **Definition 5 :**

The mapping matrix is denoted as M of size nr×nc, where nr denotes the number of routers in TG and nc denotes the number of cores in ACG. If there is a mapping between router ri and core cj then the element mij=1 otherwise it is 0.

### **Definition 6 :**

The communication cost is calculated as:

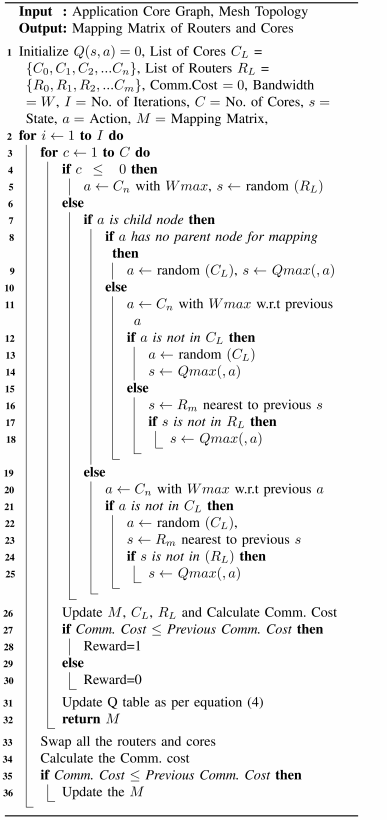
Comm.Cost=∑i=1n(No.ofHops∗Bandwidth)=(∑i=1n∑i=1ndij∗lij)∗(∑k=1n∑m=1nekm∗bkm)(2)

**>** Q-learning is a value-based RL technique for performing a task optimally in an environment using Markov Decision Process (MDP). It is an Off-policy Temporal Difference (TD) Controlled Method. As shown in Figure\_10, the agent acts according to the learning policy by observing the current state S of the environment and take action to move into new state. Depending on the quality of its action, the RL model produces a reward R to the agent and controls the agent's learning.  
  **TargetValue =R+γmaxQt-1(St+1,A)**

**TDError=[R+γmaxQt-1(St+1,A)−Qt−1(S,A)]**

**Qt(S,A)=Qt−1(S,A)+α[R+γmaxQt−1(St+1,A)−Qt−1(S,A)]**

| **RAMAN ALGORIHTM** |
| --- |



So , Basically we divide the entire algorithm in parts or modules and then we combine them to get the required output .

| **MODULES OF RAMAN ALGORITHM** |
| --- |

**MODULE : 1** **To find a random number .**

| module random\_num (b,clk,reset);  input clk,reset;  output b;  reg [7:0] b;  parameter a=5;  always @(posedge clk or posedge reset) begin  if(reset)  b <=8'd0;  else  b <=$urandom\_range(a,0);  $display ("b: %d",b);  end  endmodule |
| --- |

**MODULE : 2 To find the Index of maximum number of a Matrix**

| module Index\_of\_matrix(row\_index,column\_index,clk,reset);  input clk , reset;   parameter a=8;  parameter b=4;   reg [a-1:0] matrix [0:b-1][0:b-1];  output reg [a-1:0] row\_index;   output reg [a-1:0] column\_index;   reg [a-1:0] current\_max;  reg [a-1:0] i, j;    initial begin  $readmemh ("mba.txt",matrix);   end  always @(posedge clk or posedge reset)  if(reset) begin  current\_max=8'b00000000;  row\_index =8'b00000000;  column\_index =8'b00000000;  end  else begin  current\_max = matrix[0][0];  row\_index = 0;   column\_index= 0;    for (i = 0; i < a; i = i + 1)  begin  for (j = 0; j < a; j = j + 1)  begin  if (matrix[i][j] > current\_max)   begin  current\_max = matrix[i][j];   row\_index = i;   column\_index = j;   end  end  end  end  endmodule |
| --- |

**MODULE\_3 : To find the index of maximun number of row .**

| module FindIndex (index,clk,reset); input clk,reset; output index;  reg [3:0]index;  parameter Number =9;  parameter WIDTH = 4;  parameter SIZE = 13;  reg flag;  reg [WIDTH-1:0] array [SIZE-1:0];  initial begin  end  integer i;  initial begin   array[0] = 4'b0001;   array[1] = 4'b0010;  array[2] = 4'b0011;  array[3] = 4'b0100;  array[4] = 4'b0101;  array[5] = 4'b0110;  array[6] = 4'b0111;  array[7] = 4'b1000;  array[8] = 4'b0000;  array[9] = 4'b1010;  array[10]= 4'b1011;  array[11]= 4'b1001;   end always @(posedge clk or posedge reset) begin   if(reset)   index =4'b0000;  else   $display("Array elements:");  for ( i = 0; i < SIZE-1; i = i + 1)  $display("array[%0d] = %b", i, array[i]);  end    task findIndexTask;  begin  index = 0;  flag = 0;     for (i = 0; i < SIZE-1; i = i + 1) begin  if (array[i] == Number) begin  index = i;  flag = 1;   $display("index[%0d]=%b",i,index);  end  end    if (flag == 0)   index = 0;  $display("index not found");  end  endtask      always @(posedge clk or posedge reset) begin  if(reset)  index=4'b0000;  else   findIndexTask();  end   endmodule |
| --- |

**MODULE\_4 : To find the maximum number in a row.**

| module MatrixMaxValue\_row (  input wire clk,   output reg [7:0] next\_value  );  parameter bits=8;    reg [7:0] bw\_matrix [0:5][0:5];         reg [bits-1:0] next\_action;  reg [bits-1:0] next\_state;  reg [bits-1:0] rt;  reg [bits-1:0] cr;    reg flag\_r;    wire [bits-1:0] row;  wire [bits-1:0] random;    random\_num A(.b(row),.clk(clk));  random\_num B(.b(random),.clk(clk));    initial begin  $readmemh("bwa.txt", bw\_matrix);  end    reg [7:0] max\_val;  reg [7:0] temp\_val;    integer i,R,l1;   always @(posedge clk) begin  if(l1==32'd0)begin  flag\_r=0;  next\_action =row;  next\_state = random;  cr = next\_action;  rt<=next\_state;  end  else   if (l1==32'd1)begin   for (R=1;R<6;R=R+1) begin   max\_val = bw\_matrix[row][00];  for (i = 1; i < 6; i = i + 1) begin  temp\_val = bw\_matrix[row][i];  if (temp\_val > max\_val)  max\_val = temp\_val;  end  end  if(R==32'd6) begin  flag\_r=32'd1;  end   end  assign next\_value = max\_val;   end  endmodule |
| --- |

**MODULE\_5 : To find the maximum Number in a column .**

| module MatrixMaxValue\_col (  input wire clk,     output reg [7:0] next\_value1  );  parameter bits=8;    reg [bits-1:0] matrix [0:5][0:5];      reg [bits-1:0] next\_action;  reg [bits-1:0] next\_state;  reg [bits-1:0] rt;  reg [bits-1:0] cr;    reg flag\_c;    wire [bits-1:0] column;  wire [bits-1:0] random;    random\_num A(.b(column),.clk(clk));  random\_num B(.b(random),.clk(clk));    initial begin  $readmemh("ab.txt", matrix);  end    reg [7:0] max\_val;  reg [7:0] temp\_val;  integer f,R,l2;   always @(posedge clk) begin  if(l2==32'd0)begin  flag\_c=0;  next\_action=column;  next\_state=random;  cr=next\_action;  rt=next\_state;  end  else if(l2==32'd1) begin   for (R=1;R<6;R=R+1) begin   max\_val = matrix[0][cr];  for (f = 1; f < 6; f = f + 1) begin  temp\_val = matrix[f][cr];  if (temp\_val > max\_val)  max\_val = temp\_val;  end  end  if(R==32'd6) begin  flag\_c=32'd1;  end end  assign next\_value1 = max\_val;  end  endmodule |
| --- |

**MODULE\_6 To find the cost of the distance and bandwidth matrix .**

| **module pro ();**  **parameter cycles = 5;**  **parameter rows = 3, cols = 2;**  **parameter word = 8;**  **integer cv;**  **reg [word-1:0] hop;**  **reg [word-1:0] bw;**  **reg [word-1:0] bw\_matrix [0:(rows\*cols-1)][0:(rows\*cols-1)];**  **reg [word-1:0] dist\_matrix[0:(rows\*cols-1)][0:(rows\*cols-1)];**  **reg [word-1:0] rx[0:(rows\*cols-1)];**  **reg [word-1:0] cx[0:(rows\*cols-1)];**  **reg [word-1:0] r1, r2, c1, c2;**  **reg [word-1:0] result1[0:(rows\*cols-1)][0:(rows\*cols-1)];**  **integer cost;**  **integer cost\_1;**  **integer cost\_2;**  **integer cost\_4;**  **integer r, c;**  **reg clk; // Clock signal**  **initial begin**  **$readmemh("ab.txt", rx);**  **$readmemh("bc.txt", cx);**  **$readmemh("cd.txt", dist\_matrix);**  **$readmemh("de.txt", bw\_matrix);**  **clk = 0;**  **forever begin**  **#5 clk = ~clk;**  **end**  **end**  **always @(posedge clk) begin**  **cv = 0;**  **r = 0;**  **c = 0;**  **cost\_2 = 0;**  **cost\_4 = 0;**  **for (r = 0; r < rows\*cols; r = r + 1) begin**  **for (c = 0; c < rows\*cols; c = c + 1) begin**  **r1 = rx[r];**  **c1 = cx[r];**  **r2 = rx[c];**  **c2 = cx[c];**  **hop = dist\_matrix[r1][r2];**  **bw = bw\_matrix[c1][c2];**  **cost\_1 = hop \* bw;**  **#5 cost\_2 = cost\_2 + cost\_1;**  **#5 cost = cost\_2;**  **result1[r][c] = cost\_1;**  **#5 cost\_4 = cost\_4 + result1[r][c];**  **$display("Loop %0d-%0d: cost\_2 = %d, cost = %d, cost\_4 = %d", r, c, cost\_2, cost, cost\_4);**  **end**  **end**  **end**  **endmodule** |
| --- |

| **ENTIRE CODE IN ONE FRAME AFTER COMPILING** |
| --- |

**RAMAN ALGORITHM :**

| module ip\_test(RX1,CX1,cost,clk,reset);  output RX1,CX1,cost;  input clk,reset;  parameter rows=3,cols=2,max\_value =16,max\_bw=128,iterations=1000;  parameter bits=8;  parameter random=5;  reg [0:rows\*cols-1] Q1 [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [0:rows\*cols-1] map [0:(rows\*cols-1)][0:(rows\*cols-1)];;  reg [0:rows\*cols-1]RX1;  reg [0:rows\*cols-1]CX1;  reg [0:(rows\*cols-1)] COST\_O;  reg flag\_cost;  reg [bits-1:0] Q [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0]RX [0:(rows\*cols-1)] ;  reg [bits-1:0] CX [0:(rows\*cols-1)];  reg [bits-1:0]cost\_oldld\_1;  wire [bits-1:0] random;  reg [bits-1:0] map\_prefinal [0:(rows\*cols-1)][0:(rows\*cols-1)]; //[0:(rows\*cols-1)];  reg [bits-1:0] Q\_temp [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0] map\_temp [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0] dist\_matrix [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0] bw\_matrix [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0] bw\_matrix\_temp [0:(rows\*cols-1)][0:(rows\*cols-1)];  reg [bits-1:0] router\_temp [0:(rows\*cols-1)];  reg [bits-1:0] core\_temp [0:(rows\*cols-1)];  reg [bits-1:0] routers [0:(rows\*cols-1)];  reg [bits-1:0] cores [0:(rows\*cols-1)];  reg [bits-1:0] routers\_1 [0:(rows\*cols-1)];  reg [bits-1:0] cores\_1 [0:(rows\*cols-1)];  reg [bits-1:0] dist\_matrix\_rt [0:(rows\*cols-1)];  reg [bits-1:0] dist\_matrix\_rt1 [0:(rows\*cols-1)];  reg [bits-1:0] next\_action;  reg [bits-1:0] next\_state;  reg [bits-1:0] rt;  reg [bits-1:0] cr;  reg load\_1;  reg load\_2;  reg value\_1;  reg flag\_state;  reg flag\_action;  reg [bits-1:0] rx [0:(rows\*cols-1)];  reg [bits-1:0] cx [0:(rows\*cols-1)];  reg [bits-1:0] next\_value;  reg flag;  reg dist\_rt1\_index;  integer rx\_end;  integer cx\_end;  integer x,R,f,i,j,j1,k,k1,l,r,c,y,m1,j2,f2,routers\_1\_index,cores\_1\_index;  integer hop,bw,cost;  integer cost\_old,cost\_new;  integer m2;    integer cost\_3;  reg cost\_flag;  reg[bits-1:0]value\_action;  reg [bits-1:0] next\_value1;  reg [bits-1:0] value;  reg router\_flag;  reg [bits-1:0] cv;  reg rew;  reg routers\_empty;  reg cores\_empty;  reg rx\_empty;  reg cx\_empty;  reg value\_cr;  reg flag\_cr;  reg old\_cr;  reg [bits-1:0] c1;  reg [bits-1:0] c2;  reg [bits-1:0] r1;  reg [bits-1:0] r2;  reg [bits-1:0] bwx [0:(rows\*cols-1)];  initial begin  $readmemh("dist.txt", dist\_matrix);  $readmemh("bw.txt", bw\_matrix);  $readmemh("zeros.txt", Q);  $readmemh("zeros.txt", map\_prefinal);  $readmemh("RC.txt", router\_temp);  $readmemh("RC.txt", core\_temp);  $readmemh("rx.txt", RX);  $readmemh("rx.txt",CX);  $readmemh("bwx1.txt",bwx);  end  always @(posedge clk) begin  if(reset==1) begin  load\_1 <=0;  load\_2<=0;  cr <=0;  next\_value <=0;  j<=0;  j1<=0;  k<=0;  l<=0;  R<=0;  y<=0;  f2<=0;  m1<=0;  next\_value1<=0;  next\_value1<=0;  routers\_empty<=0;  value\_action<=0;  value<=0;  m2<=0; |
| --- |

| r1<=0;  c1<=0;  r2<=0;  c2<=0;  routers\_1\_index=1;  cores\_1\_index =1;  rt =0;  cost\_old<=0;  cost\_new<=0;  cv<=0;  end  else begin  for(j=32'd0;j<=1000;j=j+32'd1) begin :Iterations  load\_1<=1;  load\_2<=1;  value\_1<=0;  flag\_state<=0;  flag\_action <=0;  routers\_1\_index<=1;  cores\_1\_index<=1;  m2<=0;  for(k=32'd0;k<=(rows\*cols-1);k=k+32'd1) begin : intialize  for(l=32'd0;l<=(rows\*cols-1);l=l+32'd1) begin  map\_temp[k][l]<=0;  Q\_temp[k][l]<=Q[k][l];  bw\_matrix\_temp[k][l]<=bw\_matrix[k][l];  end  end  for(k1=32'd0;k1<=(rows\*cols-1);k1=k1+32'd1)  rx[k1]<=0;  cx[0]<=bwx[0];  cx[1]<=bwx[1];  cx [2] <=bwx[2];  cx[3]<=bwx[3];  cx[4]<=bwx[4];  cx[5]<=bwx[5];  dist\_matrix\_rt1[k1] <=0;  dist\_matrix\_rt[k1] <=0;  end  rx\_end<=1;  cx\_end<=1;  cr<=0;  flag<=0;  dist\_rt1\_index<=1;  value\_cr<=0;  old\_cr<=0;  flag\_cr=0;  if(j==32'd0) begin  for(k=32'd0;k<=(rows\*cols-1);k=k+32'd1) begin  routers[k]<=router\_temp[k];  cores[k]<=core\_temp[k];  routers\_1[0]<=bwx[0]; |
| --- |

| routers\_1[1]<=bwx[1];  routers\_1[2]<=bwx[2];  routers\_1[3]<=bwx[3];  routers\_1[4]<=bwx[4];  routers\_1[5]<=bwx[5];  cores\_1[k]<=0;  end  end  else begin  for(k=32'd0;k<=(rows\*cols-1);k=k+32'd1) begin  routers[k]<=routers\_1[k];  cores[k]<=cores\_1[k];  routers\_1[0]<=bwx[0];  routers\_1[1]<=bwx[1];  routers\_1[2]<=bwx[2];  routers\_1[3]<=bwx[3];  routers\_1[4]<=bwx[4];  routers\_1[5]<=bwx[5];  cores\_1[k]<=0;  end  end  for (R=32'd0;R<(rows\*cols-1);R=R+32'd1)begin :RLOOP  if(R==32'd0)begin  next\_action[bits-1:0]<=32'd2;  next\_state <=routers[5];  cr[bits-1:0]=next\_action[bits-1:0];  assign rt =next\_state;  end  else begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(bw\_matrix[cr][f]>bw\_matrix[cr][f+1]) begin  next\_value[bits:0]<=bw\_matrix[cr][f];  end  if(bw\_matrix[f][cr]>bw\_matrix[f+1][cr]) begin  next\_value1[bits:0]<=bw\_matrix[f][cr];  end  end  $display("nextvalue=%h",next\_value);    if (next\_value[bits-1:0]==0) begin  if (next\_value1[bits-1:0]==0) begin  next\_action[bits-1:0]<=cores[4];  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(Q\_temp[f][next\_action]>Q\_temp[f+1][next\_action]) begin  value = f;  end  end  next\_state<=value;  cr [bits-1:0]<= next\_action [bits-1:0];  assign rt = next\_state;  end |
| --- |

| else begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(bw\_matrix[cr][f]>bw\_matrix[cr][f+1]) begin  value\_action [bits-1:0]<=f;  end  end  next\_action [bits-1:0]<=value\_action[bits-1:0];  cr [bits-1:0]<= next\_action [bits-1:0];  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(cores\_1[f]==cr[bits-1:0])begin  flag\_cr<=1;  end  else begin  flag\_cr<=0;  end  end  if(flag\_cr==1) begin  cr[bits-1:0]=random[bits-1:0];  next\_action[bits-1:0]<=cr[bits-1:0];  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(Q\_temp[f][next\_action]>Q\_temp[f+1][next\_action]) begin  value[bits-1:0]<=f;  end  end  next\_state[bits-1:0]=value[bits-1:0];  end  else begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  dist\_matrix\_rt[f]<=dist\_matrix[rt][f];  dist\_matrix\_rt1[f]<=0;  end  dist\_rt1\_index<=1;  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  for (y=32'd0;y<=(rows\*cols-1);y=y+32'd1) begin  if(dist\_matrix\_rt[y]==1) begin  dist\_matrix\_rt1[dist\_rt1\_index]<=y;  dist\_rt1\_index<=dist\_rt1\_index+1;  end  end  end  flag\_state<=0;  for (i=32'd0;i<=(rows\*cols-1);i=i+32'd1) begin  if (flag\_state==0) begin  next\_state[bits-1:0]=dist\_matrix\_rt[l][i];  for (f=32'd0;f<(rows\*cols-1);f=f+32'd1) begin  if (routers\_1[f]==next\_state[bits-1:0]) begin  value\_1<=1;  end  end  if(value\_1==1) begin  flag\_state=0;  end  else if (value\_1==0) begin  flag\_state=1; |
| --- |

| end  else begin  flag\_state=1;  end  end  end  for (f=32'd0;f<(rows\*cols-1);f=f+32'd1) begin  if(routers\_1[f]==next\_state[bits-1:0]) begin  router\_flag<=1;  end  else begin  router\_flag<=0;  end  end  if(router\_flag==1) begin  for (k=32'd0;k<=(rows\*cols-1);k=k+32'd1) begin  dist\_matrix\_rt1[k]<=0;  end  dist\_rt1\_index<=1;  for (l=32'd0;l<=(rows\*cols-1);l=l+32'd1) begin  if(dist\_matrix\_rt1[l]<=2)begin  dist\_matrix\_rt1[dist\_rt1\_index]<=1;  dist\_rt1\_index<=dist\_rt1\_index+1;  end  end  flag\_state<=0;    for (j2=0;j2<6;j2=j2+1) begin  if(flag\_state==0) begin  next\_state[bits-1:0]<=dist\_matrix\_rt1[l][j2];  for (f2=0;f2<6;f2=f2+1) begin  if(routers\_1[f2]==next\_state[bits-1:0]) begin  value\_1<=1;  end  end  if(value\_1==1) begin  flag\_state=0;  end  else if (value\_1==0) begin  flag\_state=1;  end  else begin  flag\_state=1;  end  end  end  end  for (j2=0;j2<6;j2=j2+1) begin  if(routers\_1[j2]==next\_state[bits-1:0]) begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(Q\_temp[f][next\_action]>Q\_temp[f+1][next\_action]) begin  assign value = f;  end  end |
| --- |

| next\_state[bits-1:0]=value[bits-1:0];  end  end  end  end  end  else begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(bw\_matrix[f][cr]>bw\_matrix[f+1][cr]) begin  assign value\_action = f;  end  end  next\_action[bits-1:0]<=value\_action[bits-1:0];  cr[bits-1:0]<=next\_action[bits-1:0];    for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(cores\_1[f]==cr[bits-1:0]) begin  flag\_cr<=1;  end  else begin  flag\_cr<=0;  end  end  if(flag\_cr==1) begin  cr[bits-1:0]=random[bits-1:0];  next\_action[bits-1:0]<=cr[bits-1:0];  end    for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  dist\_matrix\_rt[f]<=dist\_matrix[rt][f];  dist\_matrix\_rt1[f]<=0;  end  dist\_rt1\_index<=1;  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  for (y=32'd0;y<=(rows\*cols-1);y=y+32'd1) begin  if(dist\_matrix\_rt[y]==1) begin  dist\_matrix\_rt1[dist\_rt1\_index]<=y;  dist\_rt1\_index<=dist\_rt1\_index+1;  end  end  end  flag\_state<=0;  for (i=32'd0;i<=(rows\*cols-1);i=i+32'd1) begin  if(flag\_state==0) begin  next\_state[bits-1:0]=dist\_matrix\_rt[1][i];  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(routers\_1[f]==next\_state[bits-1:0]) begin  value\_1<=1;  end  end  if(value==1) begin  flag\_state=0;  end  else if (value\_1==0) begin |
| --- |

| flag\_state=1;  end  else begin  flag\_state=1;  end  end  end  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(routers\_1[f]==next\_state[bits-1:0]) begin  router\_flag<=1;  end  else begin  router\_flag<=0;    end  end  if(router\_flag==1) begin  for (k=32'd0;k<=(rows\*cols-1);k=k+32'd1) begin  dist\_matrix\_rt1[k]<=0;  end  dist\_rt1\_index<=1;  for (l=0;l<=(rows\*cols-1);l=l+1) begin  if(dist\_matrix\_rt[l]==2) begin  dist\_matrix\_rt1[dist\_rt1\_index]<=l;  dist\_rt1\_index<=dist\_rt1\_index+1;  end  end  flag\_state<=0;  for (j2=0;j2<6;j2=j2+1) begin  if(flag\_state==0) begin  next\_state[bits-1:0]=dist\_matrix\_rt1[1][j2];  for (f2=0;f2<6;f2=f2+1) begin  if(routers\_1[f2]==next\_state[bits-1:0]) begin  value\_1<=1;  end  end  if(value\_1==1) begin  flag\_state=0;  end  else if(value\_1==0) begin  flag\_state=1;  end  else begin  flag\_state=1;  end  end  end  end  for (j2=0;j2<=dist\_rt1\_index;j2=j2+1) begin  if(routers\_1[j2]==next\_state[bits-1:0]) begin  for (f=32'd0;f<=(rows\*cols-1);f=f+32'd1) begin  if(Q\_temp[f][next\_action]>Q\_temp[f+1][next\_action]) begin  value[bits-1:0]<=f;  end |
| --- |

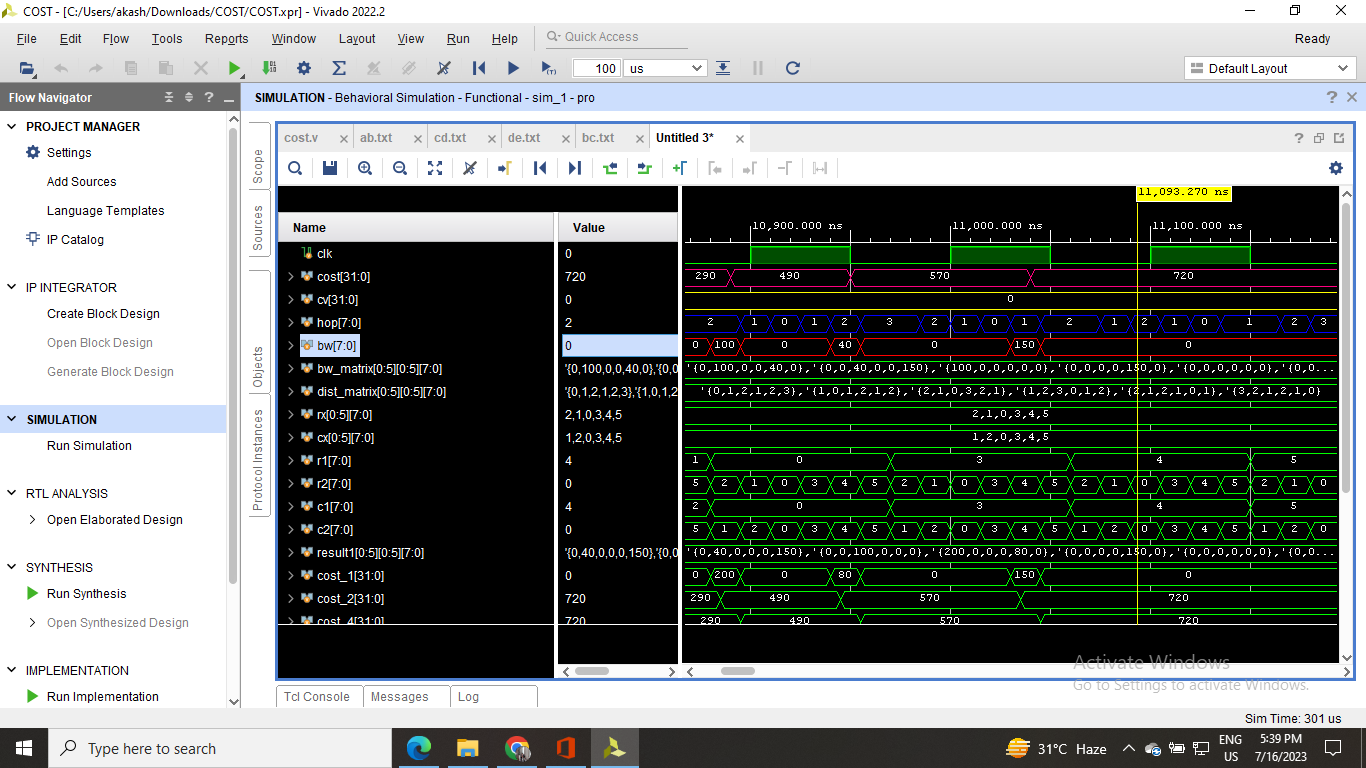
| end  next\_state[bits-1:0]=value[bits-1:0];  end  end  end  end  rt[bits-1:0]<=next\_state[bits-1:0];  for (f=0;f<=(rows\*cols-1);f=f+1) begin  Q\_temp[next\_state][f]<=-8'sd128;  if(routers\_1[f]==0) begin  routers\_empty<=0;  end  else if(next\_state==routers\_1[f]) begin  routers\_empty<=0;  end  else if (next\_state==routers\_1[f]) begin  routers\_empty<=0;  end  else begin  routers\_empty<=1;  end  if(cores\_1[f]==0) begin  cores\_empty<=0;  end  else if (next\_action==cores\_1[f]) begin  cores\_empty<=0;  end  else begin  cores\_empty<=1;  end  if(rx[f]==0) begin  rx\_empty<=0;  end  else begin  rx\_empty<=1;  end  if(cx[f]==0) begin  cx\_empty<=0;  end  else begin  cx\_empty<=1;  end  end  map\_temp[next\_state][next\_state]<=1;  bw\_matrix\_temp[next\_state][next\_state]<=-8'sd128;    if(!routers\_empty) begin  routers\_1[routers\_1\_index]<=next\_state[bits-1:0];  load\_1<=1;  end  else if (routers\_empty==0) begin  load\_1<=0;  end  else begin |
| --- |

| routers\_1\_index<=routers\_1\_index+1;  routers\_1[routers\_1\_index]<=next\_state[bits-1:0];  load\_1<=1;  end    if(!cores\_empty) begin  cores\_1[cores\_1\_index]<=next\_action[bits-1:0];  load\_2<=1;  end  else if (cores\_empty==0) begin  load\_2<=0;  end  else begin  cores\_1\_index<=cores\_1\_index+1;  cores\_1[cores\_1\_index]<=next\_action[bits-1:0];  load\_2<=1;  end  if(load\_1==1) begin  if(rx\_empty==0) begin  rx[1][rx\_end]<=next\_state[bits-1:0];  end  else begin  rx\_end=rx\_end+1;  rx[1][rx\_end]=next\_state[bits-1:0];  end  end  if(load\_2==1) begin  if(rx\_empty==0) begin  cx[1][rx\_end]<=next\_action[bits-1:0];  end  else begin  cx\_end=cx\_end+1;  cx[1][cx\_end]=next\_action;  cost\_flag<=1;  end  end  end    if (cost\_flag==0) begin  cost<=0;  end  else begin  cost\_2 =0;  cost=1;  for (r=0;r<6;r=r+1) begin  r1[bits-1:0]<=rx[r];  c1[bits-1:0]<=cx[r];  for (c=0;c<6;c=c+1) begin  r2[bits-1:0]<=rx[c];  c2[bits-1:0]=cx[c];  hop<=dist\_matrix[r1][r2];  bw<=bw\_matrix[c1][c2];  cost\_2<=hop\*bw; |
| --- |

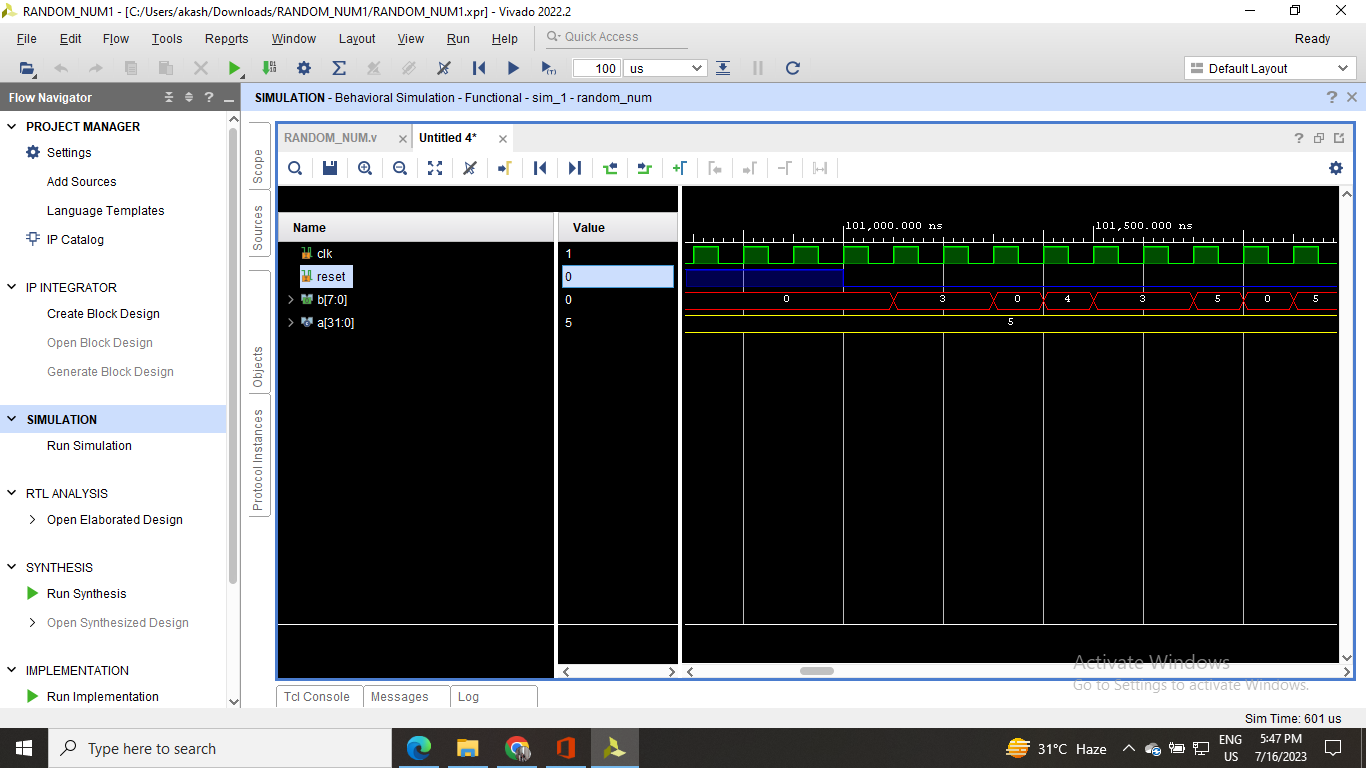
| cost<=cost+cost\_2-8'h1;  end  end  if(cv==0) begin  cost\_new <=cost;  cost\_old <=cost;  cv<=cv+1;  end  else begin  cost\_new<=cost;  end  end  if(cost\_new<=cost\_old) begin  cost\_old<=cost\_new;    for (m1=0;m1<6;m1=m1+1) begin  map\_prefinal[l][m1]<=map\_temp[l][m1];  end  for (m2=0;m2<5;m2=m2+1) begin  RX[m2]<=rx[m2];  end  for (m2=0;m2<=(rows\*cols-1);m2=m2+1) begin  CX[m2]=cx[m2];  end  rew<=1;  end  else begin  rew<=0;  end  for (x=32'd0;x<=(rows\*cols-1);x=x+32'd1) begin  Q[rx[x]][cx[x]]=Q[rx[x]][cx[x]]+rew;  end  end  end  initial begin  assign COST\_O = cost\_old;  end  integer gen\_1;  integer gen\_2;  initial begin  for (gen\_1 = 0; gen\_1 <= (rows \* cols - 1); gen\_1 = gen\_1 + 1) begin  for (gen\_2=0;gen\_2<=(rows\*cols-1);gen\_2=gen\_2+1)  Q1[gen\_1][gen\_2] = Q[gen\_1][gen\_2];  map[gen\_1][gen\_2]=map\_prefinal[gen\_1][gen\_2];  RX1[gen\_1] = RX[gen\_1];  CX1[gen\_1] = CX[gen\_1];  end  end  endmodule |
| --- |

| **SNAPSHOTS OF RAMAN ALGORITHM** |
| --- |

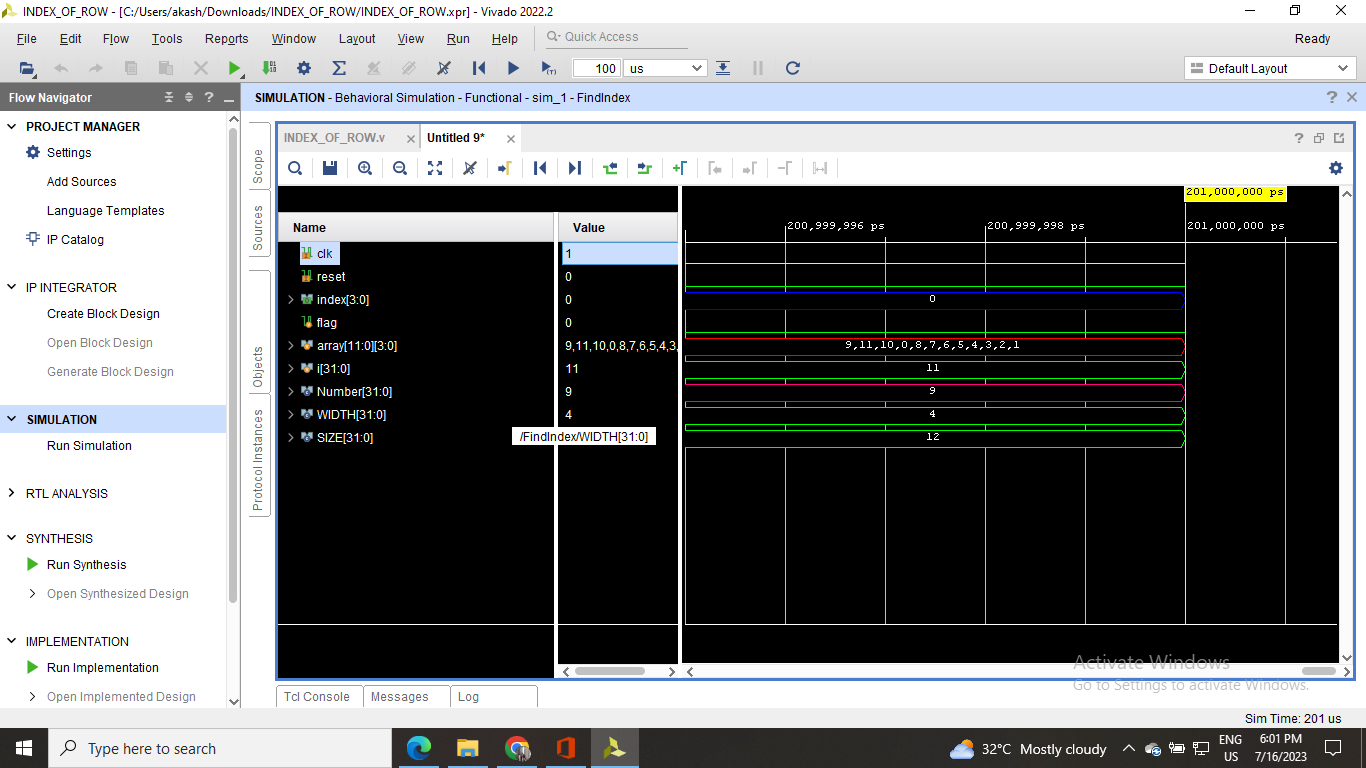
**SNAPSHOT OF COST MODULE :**



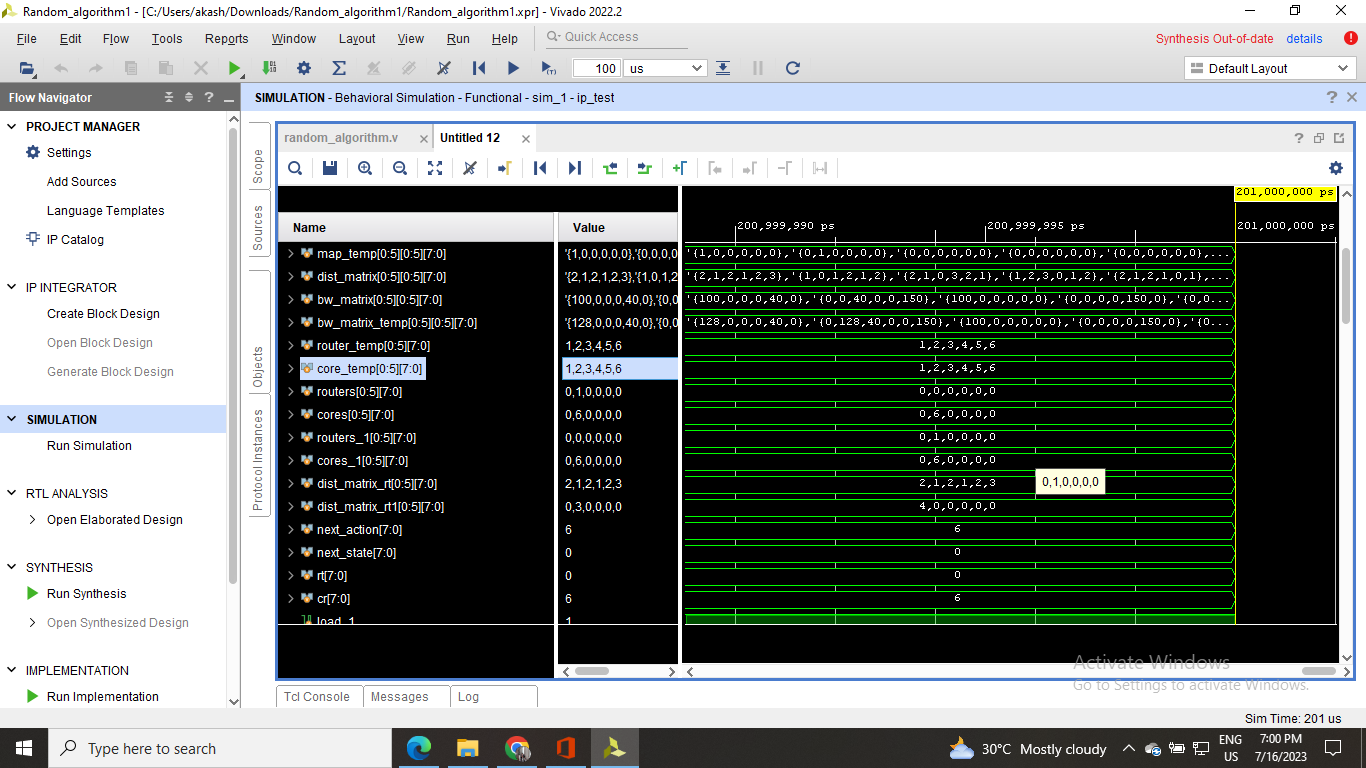
**SNAPSHOT OF RANDOM NUMBER :**



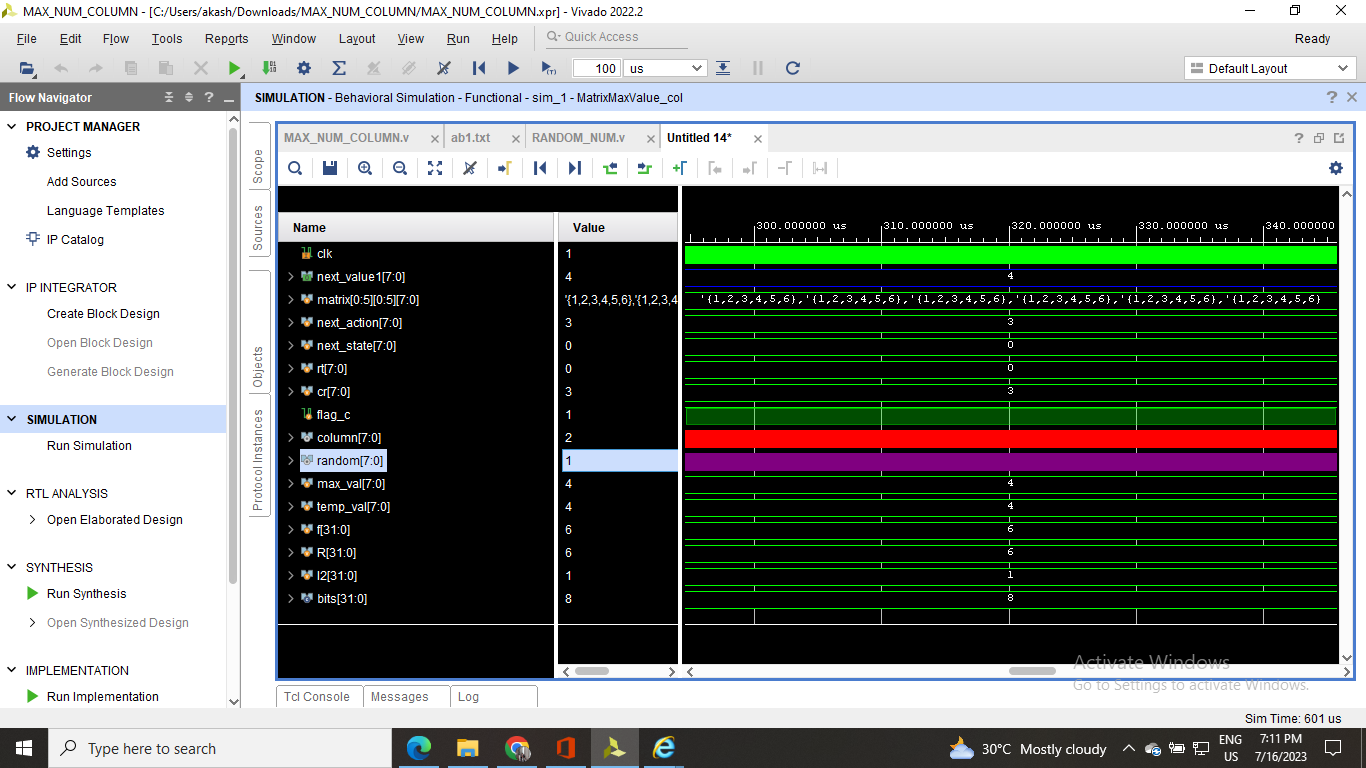
**SNAPSHOT OF INDEX OF A NUMBER IN A ROW :**



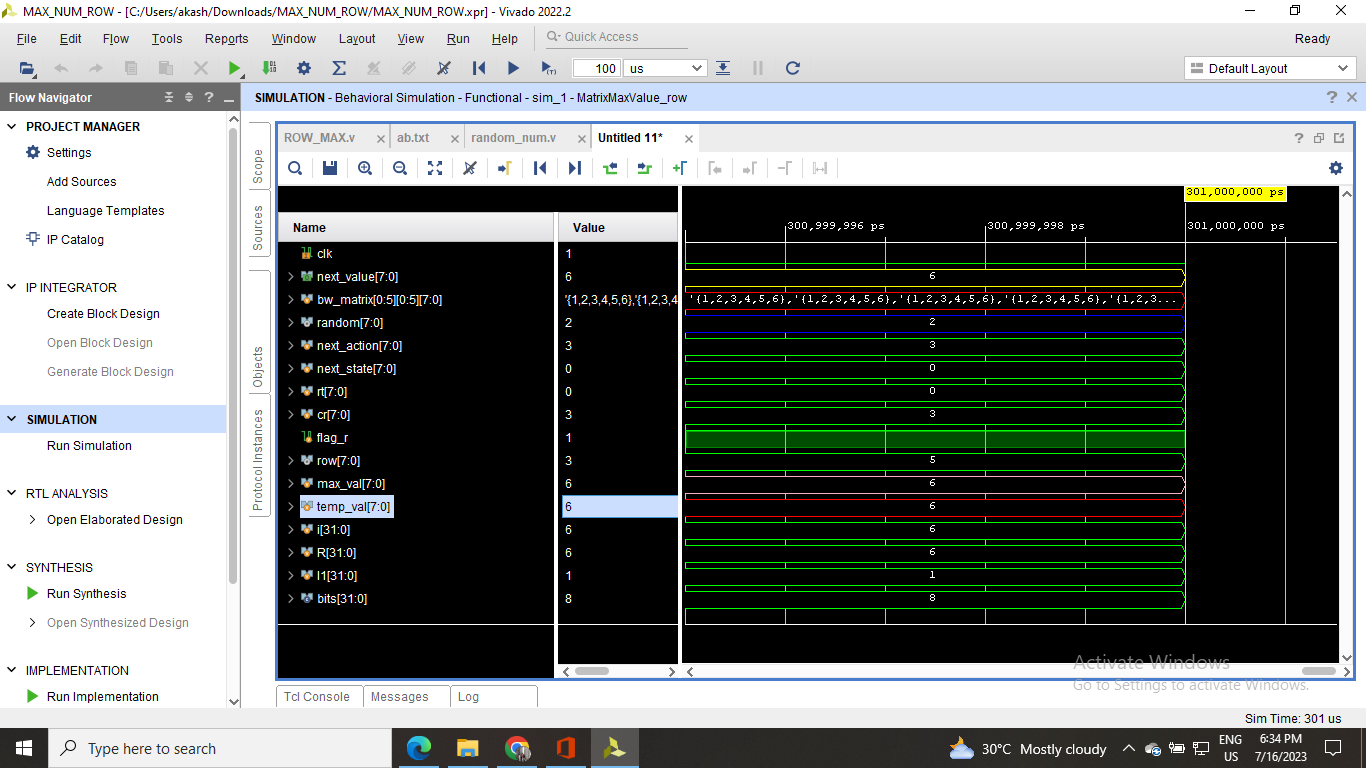
**SNAPSHOT OF RAMAN ALGORIHTM ENTIRE CODE :**



**SNAPSHOT OF MAXIMUM NUMBER IN A COLUMN :**



**SNAPSHOT OF MAXIMUM NUMBER IN A ROW :**



| **CONCLUSION** |
| --- |

In the “Code dumping of digital circuits” on FPGA , our primary objectives were to design and implement various digital circuits, including a full adder, N-bit parallel adder, Gray to binary converter, and JK flip-flop, on the PYNQ-Z2 board. We successfully designed, simulated, and implemented the following digital circuits on the PYNQ-Z2 board: a full adder, an N-bit parallel adder, a Gray to binary converter, and a JK flip-flop. Each circuit was thoroughly tested to ensure their correct functionality.

**Results and Performance** : The implemented circuits exhibited the expected behavior, and all of them performed accurately according to their specifications. The N-bit parallel adder showed efficient and reliable performance for various input sizes, while the Gray to binary converter provided seamless conversion between the two numeral systems. Moreover, the JK flip-flop demonstrated stable operation and precise control over its outputs.

**Lessons Learned** : This project provided us with valuable hands-on experience in digital circuit design and FPGA implementation. We learned how to leverage the capabilities of the PYNQ-Z2 board effectively and gained insights into FPGA programming techniques. Additionally, we became proficient in using Vivado and PYNQ environments for hardware description and high-level **synthesis.**

**Future Directions** : In the future, we envision further optimizing the circuits' resource utilization and exploring power-saving techniques for more energy-efficient designs. Additionally, integrating these circuits into larger systems or combining them with other FPGA-based applications could be an interesting avenue for exploration.

**Overall Project Impact** : This project allowed us to gain a deeper understanding of digital circuit design principles and FPGA implementation. The successful implementation and testing of various circuits on the PYNQ-Z2 board showcased the practical applicability of FPGA-based systems. The knowledge and skills acquired during this project will undoubtedly prove valuable in our future endeavors in the field of digital electronics and hardware design.

| **CONCLUSION** |
| --- |

In second project we have performed RAMAN algorithm in Verilog, a Q-learning based RL-approach,is proposed for application mapping in NoC design and communication cost is used as a performance metric for evaluating the solution.It is veriﬁed from the examples that the RL agent acquired the policy to reduce communication costs in signiﬁcantly less time. RAMAN is scalable; random mapping graphs demonstrate the scalability with nodes up to 121.

Notably,RAMAN can ﬁnd the optimized solution for unseen graphs without any prior training.

In this work, we explored the applicability of RAMAN to mapping problems, and studied its

convergence for different application sizes.It implies that the model possesses the generalized performance for a complex optimization problem. However, RAMAN is less efﬁcient for the applications whose size is greater than 36 nodes and there is a need to improve RAMAN further.

We take this as a motivation to reﬁne RAMAN and make it more efﬁcient for large size networks in our future work. Although 2D mesh NoCs topology with XY routing is used in RAMAN, this approach can be further adapted for other regular network topologies with deterministic routing schemes.

The idea of minimizing the communication cost is not limited to application mapping; it can be effective in any similar optimization problems. RAMAN can improve the design exploration by

simplifying and reducing search space and can be used as input to other optimization methods.

| **REFERENCES** |
| --- |

[1] .<https://www.electronics-tutorials.ws/sequential/seq_2.html>

[2].https://www.physics.purdue.edu/quantum/files/Advanced%20Optical%20Materials%20-%202023%20- %20Qi%20%20Recent%20Progresses%20in%20Machine%20Learning%20Assisted%20Raman%20Spectroscopy.pdf

[3]. <https://ieeexplore.ieee.org/document/9667806>

[4]. https://www.youtube.com/watch?v=w3jNkZ-5s-U&list=PLwdnzlV3ogoVGq4TIpX4NH6QEFYiAnyvA

[5]. https://chat.openai.com/