AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Computer and Systems Engineering
International Credit Hours Engineering Programs (i.CHEP)

Research Project Report

Design, Implementation and Test of a Networks-on-Chip (NoC) Router using VHDL

Course Code **CSE215**	*Course Name* **Electronic Design Automation**	
	Semester Spring 2020	**Date of Submission** 3/6/2020

#	Student ID	Grade (PASS/FAIL)
1	17P5026	
2	17P6069	
3	17P8182	
4	17P3061	
5	17P8042	

In case of group research; list all students IDs.
DO NOT WRITE STUDENTS NAMES

"I certify that this report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons."

STUDENTS MUST SIGN THIS PAGE. ELECTRONIC SIGNATURE IS ACCEPTED.

#	Student ID	Student Signature
1	17P5026	Ahmed Khaled Aly
2	17P6069	Andrew Yasser Shaker
3	17P8182	George Emad Younan
4	17P3061	Karim Mikhael Farid
5	17P8042	Youssef Emad

In case of group research; list all students IDs.
DO NOT WRITE STUDENTS NAMES

Table of Contents:

1.0 Introduction:

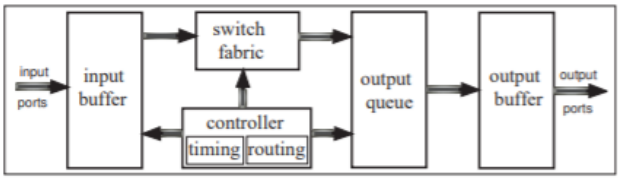
This project is aimed to design and implement a tested simple router using a synthesizable VHDL code. routers are used in computer networks applications, by forwarding data in packets format across a network toward the desired destinations using a process called “routing” that use headers and forwarding tables to guarantee the best path for the packets to be forwarded according to design requirements.

The paradigm of Network on chip provides an integrated solution for achieving shorter delays in communication and efficient interconnection between process elements, but have limited silicon area, this makes it sort of a difficult task to design NoC-based systems.
The categorization of NOC Routers is done based on two different characteristics: The location of queues and buffers within the router, and the switch fabric type. There are four main types of routers:

1. Input-queue routers
2. Output-queue routers
3. Shared-queue routers
4. Input/output-queue routers

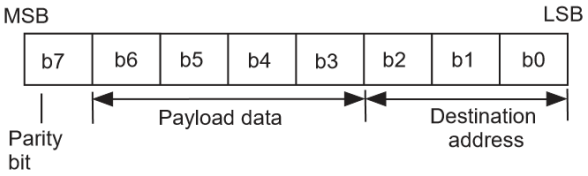
The position of queues inside the routers is crucial; as it directly affects the delay of the router, quality of service, and packet loss.
The router has 4 main building blocks: the input buffer – the switch fabric – the output queue – the output buffer. The data packets are stored at the input buffer once it arrives asynchronously with packet ready signal indicator (PR).

Then controller reads the packet header and configures the SF to direct the packet to the suitable queue (output buffer), which is a bunch of FIFOs buffers synchronized with both clock edges. The controller does the routing tables and timing synchronization. the output buffer delivers the packet from the FIFOs to output ports depending on target destination availability using a round robin scheduler algorithm.



Fig#1.1: Block diagram for an output-queue router

This RR algorithm serves the waiting queues one after the other in a fixed order at the output, using a signal called receive ready which indicates the availability of the target destination. Packets are sent to the next hop with Packet Sent (PS) signal for synchronization purposes. The packet is an eight bits data represented in the form of: first three bits represents the destination address; the following four bits has the sent data and the most significant bit is for parity check for errors.



Fig#1.2: Format of a packet

This project consists of several modules that combine to make the router: 8 bits register – 8 bit Demux – Block Ram – a Gray counter – Gray to binary converter – FIFO module – FIFO controller – Round Robin Schedular – A Module to combine these modules. The Gray code is used here to facilitate error detection.

During this project, It was assumed that:

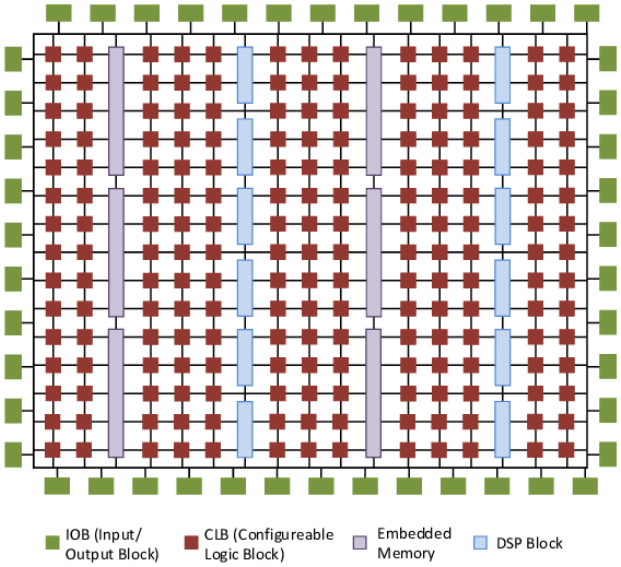
- A round robin schedular was used
- The packet input is already in the desired format
- Write ready turns to 0 after clock cycle
- In the packet header, 2 bits only are use as there are 4 output buffers only

The project was done on Xilinx ISE CAD tool starting from the design specification and was coded in VHDL language and passed through behavioral simulation then synthesis process then the post synthesis simulation down to placing and routing and time analysis simulation to generate the bit file.

2.0 Design Flow:

There are two types of technology when we think about chip design and production: FPGA & AISEC.

The AISEC: Application specific integrated circuit technology, which mostly being used in critical systems where security is crucial, is an expensive technology. However, it can be used where mass producing chips is requested.



Fig#2.1: Architecture of FPGA

The FPGA: field programmable chips that has logic blocks, I/O bloc, memory elements and high resources unlike CPLDs that have lower resources. FPGA can be reprogrammed to provide different functionalities as requested. It also can be easily tested. It can't be used in mass productions.

There are 4 main phases in the FPGA design flow: Design – simulation – synthesis, analysis and verification – layout, manufacturing preparation.

Design:

The design flow is started with the design (high-level design) phase that use specification sheet to define: the input ports size, datatypes, restrictions on waveforms entering the input, required clock frequency, time constraints in general that defines the relations of these signals with each other , the output required size and data types and synchronization with clock details; the block functionality of the project in between which helps the designer to detect a design approach to help executing the required function.

These languages (VHDL or Verilog) are used to isolate the designers from the details of the hardware implementation compared to other techniques. Some designs can be described using system level languages to describe the transactions that happens in hardware.

Some designs can be described by its schematic connections between components using schematic entry or table entry to give much more visibility into the hardware. IP integrators can be used to connect components in design to construct the system, making it easier to modify components, it is considered as semi-automated design as the IPs are already designed and is just integrated.

Behavioral Simulation:

After the high-level design, comes the behavioral simulation (RTL Simulation) to check if the design meets the exploitations from the functional view (check for the functionality of the design). If it passes then we take the design to synthesis, which finds out equivalent hardware gates or components that represent the hardware description written in the code.

Synthesis :

the synthesis Process takes inputs the VHDL or Verilog code and outputs a netlist that defines every basic component (gates, flip flops, etc…) chosen from desired technology and connections. This can be made using VHDL or Verilog or spice, or many options and there are many different tools for synthesis are based on the chosen technology, and the area of silicon is detected to know if it can fit in wafer or FPGA according to target implementation, it also gives timing, and power consumption analysis.

Verification techniques:

One of the verification techniques is gate comparison is comparing the output of the synthesis tools against the RTL gates. This helps proving that there is no change happened when synthesis happened. This is made before fabrication.

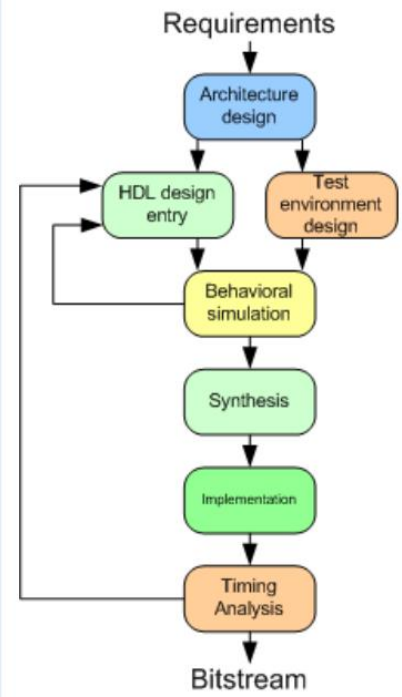
Other verification techniques are code coverage analysis, assertion analysis, functional coverage analysis, constraints random simulations. Those help to prove that testcases asserted in simulation covered all parts in the code and all the implementations or gates that are produced in synthesis, also makes sure that there is no part in the code is not relevant to the design as a security factor and quality assurance.

Post synthesis simulation:

After synthesis the target technology is detected and the timing information of this technology is added e.g. setup time, hold time, and propagation delay of combinational logic. Post synthesis simulation is done to know if the design meets requirements after those parameters are added or not. Then equivalent gate count of design is precisely detected. This is used to compare implementation of different technologies.

Layout phases:

After this comes placement and routing (physical synthesis) to put the components in their physical places in chip and the files (GPS for ASIC or bit stream for FPGA. After placement and routing a third simulation **(Static Timing Analysis**) is done considering additional factor of placement and routing on chip, as the transition of data within the micrometer of chip (signal path) can cause certain delays that may affect the critical path in circuit. So, simulation of signals is done three times in total. A bitstream file containing the programming information of FPGA is generated. Then Bit code is placed on the FPGA when I connect the cable to the computer. Then comes planning of Flip flop and the auto flow of the placement and route.



Fig#2.2: FPGA Design flow phases

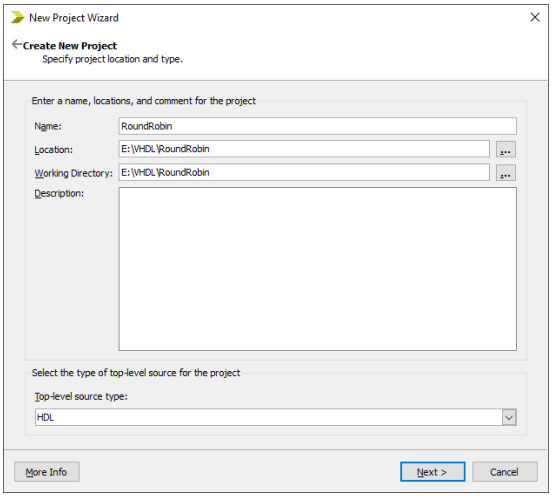
Additional procedures that can be done:

Another model is the emulation which is making one of the components in design as logic analyzer (internal logic analyzer) that its pins are connected to test points internally in chip. and the output is serial to serial port to be read on software tool. This allows us to enter in depth and point some parameters that can't be inspected easily. this happens at runtime, sampling and waveforms occur at simulator to check for the signals.

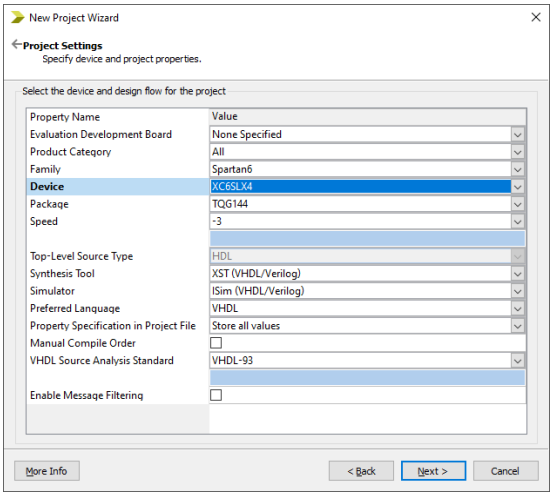
Thermal simulations for chips while working by color coding the amount of dissipated power can be detected along with the hotspot in the chip.it is very important to balance that dissipation (heat sinks or fans needed).

Points or tracks in chip which has high clock frequency may have different fabrication techniques to prevent interference by shielding or changing the layout to meet design specs using transmission line models to model those tracks. This helps seeing the effect on near tracks. BIT file is generated to be used to configure the target FPGA device using a cable.

CAD tools like the Xilinx ISE help to go develop designs through various design flow phases:

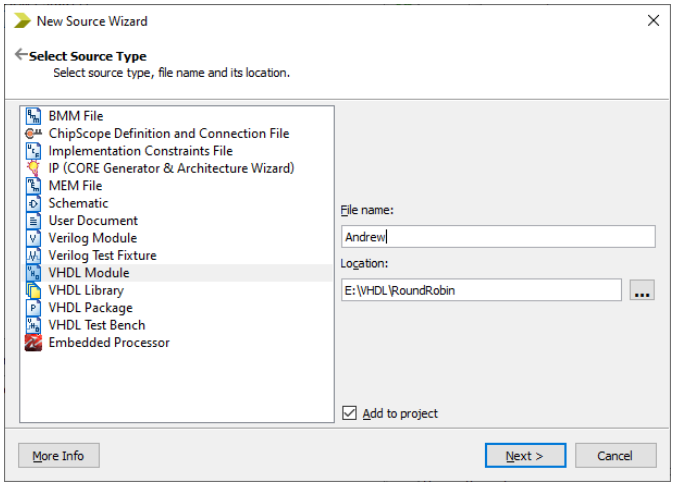


Fig#2.3: Creating new project on ISE

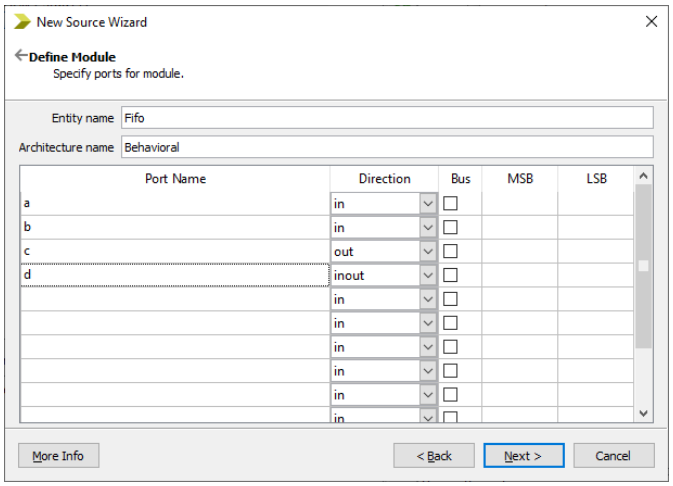


Fig#2.4: determining family from Spartan, Vertex, Kintex & Artix

The tool allows renaming files and choosing the input and output ports (names & types) and choosing the type of files from various types like: VHDL or Verilog module, Testbench module, User constraint files or even embedded processors. In addition to many facilities like syntax checking in addition to all three types of simulations.



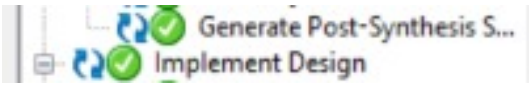
Fig#2.5: Creating new file and choosing type



Fig#2.6: Creating new file and choosing type

Simulations with Xilinx ISE:

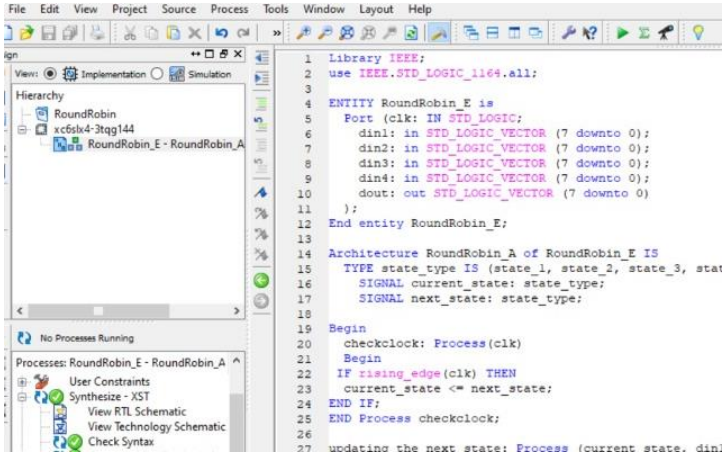
behavioral Simulation or RTL Simulation is performed before synthesis process to check for the functionality of the design. It can be performed on VHDL or Verilog codes. In this process, signals and variables are observed, procedures and functions are traced carefully. It is a quick simulation that allows t to change the code. This happens if the design does not perform the required functionality. As this happens before the process of synthesis, timing and resource usage properties are not known.



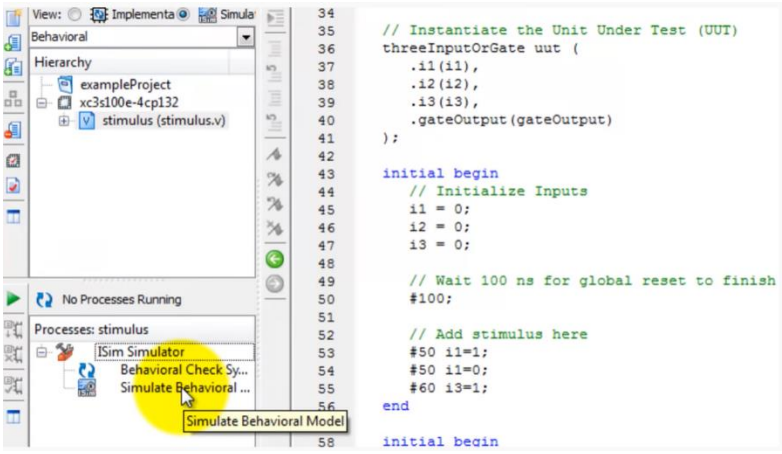
Fig#2.7: Post-synthesis simulation on ISE tools

Post synthesis Simulation is the second step that gives information about the logic operation of the circuit. to know if the design meets requirements after the target technology is detected and the timing information of this technology is added after the translate process. If the functionality is not as expected, then the designer has to modify the code and follow the design flow steps again.

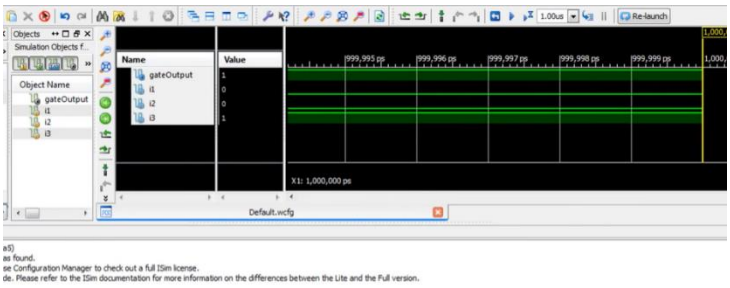
Static Timing Analysis, which is the third step can be done after MAP or Place and route processes. After mapping process, a timing report lists delays of signal paths taken from the design logic. Post Place and Route timing report shows information about timing delays to provide a useful summary of timing of the design.



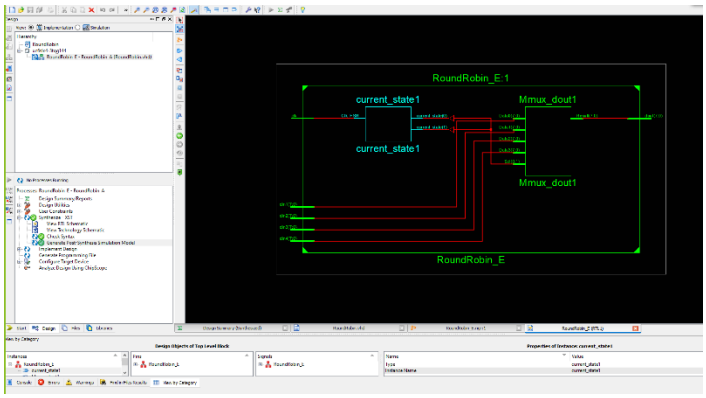
Fig#2.8: syntax Check process on RR module by ISE tools



Fig#2.9: Behavioral simulation on ISE tools

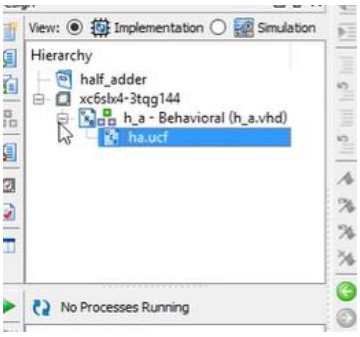


Fig#2.10: example of behavioral simulation results on ISE



Fig#2.11: RTL schematic of Round Robin on ISE

The ISE Xilinx tools allow us to define constraints. This is assigning the ports in the design to the physical elements (ex. pins, switches, buttons) of the device & specifying time requirements of the design and the info is stored in UCF file (user constraints file).



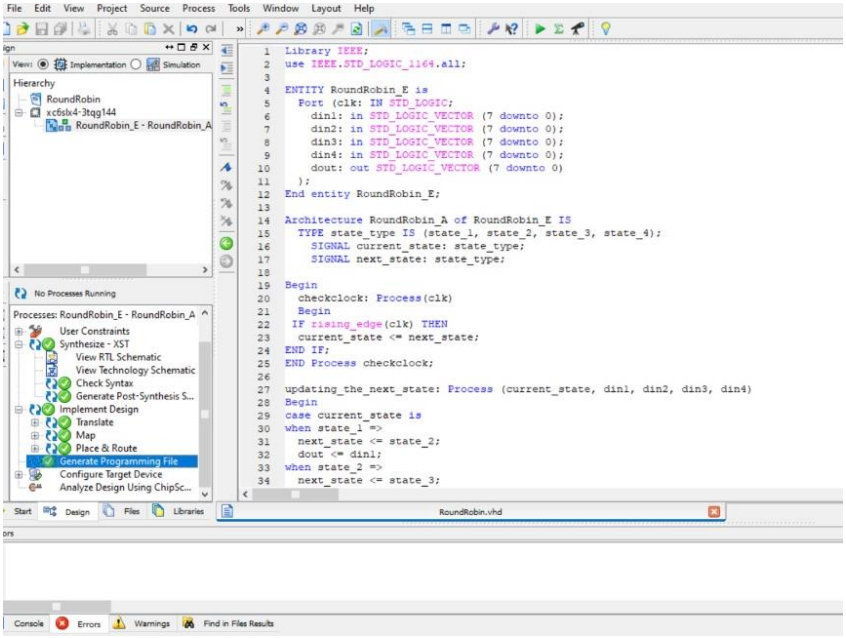
Fig#2.12: UCF file in ISE tool

The process of Synthesis Implementation is done on three steps on the ISE:

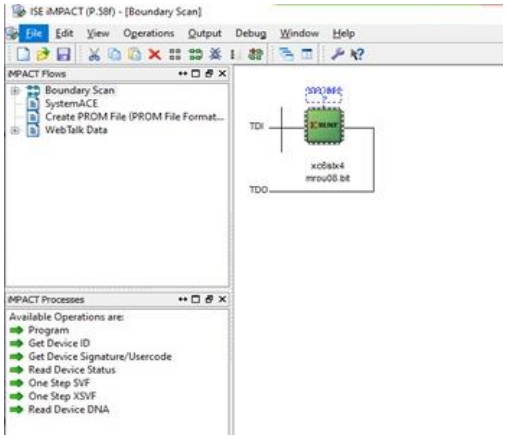
1. Translate: The input constraints and netlists are added together in a logic design file

2. Map: to fit the defined logic, which is Combinational Logic Blocks (CLB), Input Output Blocks (IOB). It generates a file which physically represents the design mapped to the components of FPGA.

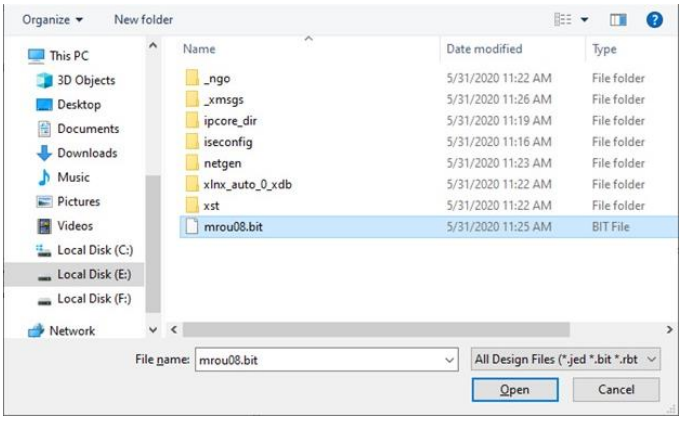
3. Place and Route: places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks.



Fig#2.13: Implementation steps of Round Robin module on ISE tool



Fig#2.14: of 8-bits register module on ISE tool



Fig#2.15: saving bit file of 8-bits register module on ISE tool

3.0 Literature Review:

We will sort the References including papers and books that contains useful data about “NOC Routers implementation” in chronological order:

1- L. Benini and G. D. Micheli, “Networks on chips: A new SoC paradigm,” IEEE Computer Magazine, vol. 35, no. 1, pp. 70–78, Jan. 2002.

In this article the problems of SOC (System-on-chip) is discussed, and the article introduces the NOC (Networks-on-chip) as a borrowed concept from the network engineers to get over the problems of the SOC.

2- LogicCore. (2004, Nov. 11,) Xilinx Technical Document, Asynchronous FIFO v6.1. Product Specification DS232

In this article, an Asynchronous FIFO is implemented and illustrated, diagrams and tables are also implemented.

3- Haytham El Miligi, “Networks-on-Chips: Modeling, Analysis, and Design Methodologies,” Victoria, B.C., Canada: 2011

4- Minakshi M. Wanjari, Pankaj Agrawal, R. V. Kshirsagar “Design of NoC Router Architecture using VHDL,” India: International Journal of Computer Applications (0975 – 8887) Volume 115 – No. 4, April 2015.

In this paper, NoC router architecture is presented, which has low latency and requires less area. Also, a design is implemented in VHDL and simulated.

5- Benini and G. D. Micheli returned again in 2017 with a new article: “Networks on Chips: 15 Years Later” to discuss their article from 2002 (mentioned above), and how virtually all large-scale chips are now designed with this paradigm (NOC).

4.0 Design Implementation:

The Register Module:

This module has 4 inputs, and one output, the input datat_in is entered in the registered and outputted if and only if Clock_En = 1 AND Reset = 0. (and its outputted when the clock edge is positive).If the reset clock edge is 1 the output is set to zeros. And if the Clock_En is zero the current output value remains as it is.

The register module was implemented by using a simple If condition, if the reset is equal to 1 then the data_out is zeros, else if the clock is on the rising edge and the clock enable is equal to 1 then the data is outputted (data_out = data_in)

The 8-bit DeMux:

A vector of 8 bits is inputted in the demux and outputted if the En = 1, but if the En is 0 the port keeps its value. But its outputted to the port depending on the selector of the demux. There are 4 output ports and therefore the selector is 2 bits. (00 for the 1st output, 01 for the 2nd..etc). The demux is implemented using an if condition and a switch case, if the enable is 1 then we navigate to the switch case, when the selection is 00 the first data is selected, when the selection is 01 the 2nd is selected..etc. the sensitivity list of the process is the enable, the data in and the selection.

Block Ram:

This module receives data (a vector of 7 bits) and it has another 3 parameters for read and 3 for write. So the first parameter is the enable, where if its equal to 1 its going to allow reading/writing. Then the address port (read/write) can be set to whatever value desired to be read from or written to. The reason why the address port is 3 bits is because there are 8 slots in the ram. And 3 bits can express 8 values.

So whenever the clock of read/write hits 1, and the read/write enabled, the read/write happens to the address selected from the 8 addresses. The code of the block ram was implemented using 2 processes, their sensitivity list is the CLKS (A,B). If the clk's triggered, then we check if the Write/read is enabled, then by a simple switch cases, we select which slot of the ram to read or write according to the pointer value.

Gray Counter:

This module is a simple counter. It increments on the rising edge of a clock whenever the Enable signal is set to 1. The reset sets it to 0. And it also counts in a gray code manner: 000, 001,011,010,110,111,101,100. The counter is implemented by using a process of a sensitivity list clock and reset, if the reset is equal to 1 then the output is restarted, else if both the clock is triggered and the enable is equal to 1, a switch case is executed, where for each case, the output is that case + 1, so if its 000 the output is 001..etc.

Gray Converter:

This module converts the output of the counter to binary, because the ram, FIFO controller works in binary and not gray. It simply changes the truth table
000, 001,011,010,110,111,101,100 To 000,001,010,011,100,101,110,111. The gray converter was implemented through a very simple mapping 3 line code, the most significant bit on the output is equal to the most significant bit of the input, the 2nd most significant bit on the output Is equal to the most significant bit in the input XORed with the 2nd most significant bit on the input. And the last bit on the output is the 3 bits of the input XORed together.

FIFO Controller:

This module contains 2 previous components discussed, converter and counter.
Whenever the reset signal is 1 the module restarts, meaning write and read valids are equal to 0, pointers are zeroed, empty is set to 1 and full is set to 0.
Only when the read/write clock is 1, the module is capable to perform read/write operations. So the module receives read/write requests. And it can't perform reading if the FIFO is empty, simultaneously it cannot write to a full FIFO. And this is what sets the read / write valid signals. The code implementation here was done by creating 2 components (instantiations) of a counter, and 2 components of a converter, one for the read and one for the write, each with their counters connected to the gray to binary converter, and then connected to the pointers, the main process of the fifo controller has the signals: read request, write request and reset in the sensitivity list. There are 2 if conditions. The first one is written for the write request and the 2nd is for the read request, so concerning the first one, if the reset is equal 1 then full is 0 and empty is 1, else if the request is 1 then we check if the read pointer is equal to the write pointer + 1 (which means that the FIFO is full). So the request is neglected and the valid is set to 0, and full is set to 1 else the valid is set to 1. Concerning the 2nd, if the reset is equal to 1 then full is 0 and empty to 1, else if the request is 1 then we check if the 2 pointers are equal, and if they are then the FIFO is empty and there is nothing to read from it. And the read valid is set to 0/ empty set to 1 in that case, else the read valid is set to 1.

FIFO:

This module contains the FIFO Controller and the Ram. The datain is written into the FIFO (of course following the restrictions based on the FIFO controller) and gets read using the dataout signal (also accordingly with the FIFO controller). The rclk and wclk are mapped with the rdclk and wrclk of the FIFO controller, respectively. And the rreq wreq signals are mapped with the r_req and w_req signals, respectively. Read/write pointers of the fifo controller are connected to the ram's read/write addresses (through the converter and the counter).

As for the implementation of this module, it didn't contain any logic. It was just mapping the 2 components FIFO controller and Ram. So basically we create 2 components. We map all of the FIFO's signals to the controller except the data_in, which gets mapped to the ram. The read and write pointers also gets into the ram from the fifo, and the write valid read valid which are outputted from the fifo controller gets into the ram as the enables. The clocks get into the ram as well. And the ram's main output is the data_out which is mapped to the FIFO's main output.

Round Robin Schedular:

This module is a round robin schedular that takes 4 inputs. Each rising edge of the clock the output differs from the 4 FIFOs. Meaning at the first clock cycle the output is taken from d_in(1), and at the second clock cycle the output is d_in(2). The schedular was implemented by using a two process moore implementation. One for checking the clock , and one for updating the next state and checks for the current state.

Router:

The router contains 4 registers, 4 demuxes, 4 round robin schedular and 16 FIFOs.

The data gets into the registers, in the form of a packet, the register is just a buffer.

So if the register receives clock en, it outputs the data, if it gets a reset, it zeroes the data.

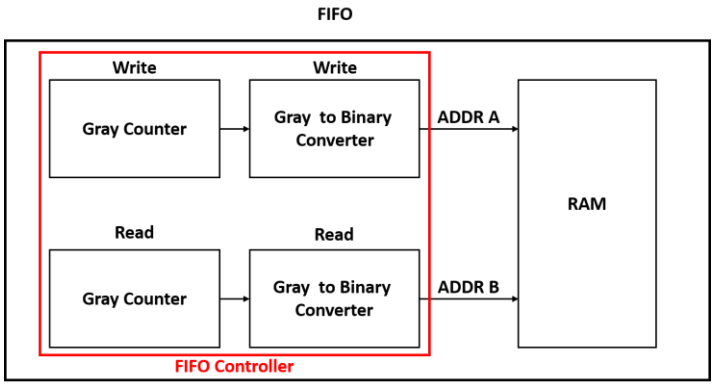
The demux receives a selection line (which is discussed earlier) from the first 2 bits in the data, which is going to select the where is the destination of the output. Every demux has 4 FIFOs connected. A FIFO for each output. The data is transferred from the demux to the FIFO. The Rclock (R-request) is designed so that every clock cycle a type of FIFO works, There are 4 types of FIFOs, for every port there is a type FIFOs 1,2,3,4 connected

to output ports 1,2,3,4. For example, every clock cycle a FIFO with no. 11,21,31,41 works, (with one at the end), because they are connected to different round robins. Round robin takes the data from 4 FIFOs, and the data is outputted through one output.

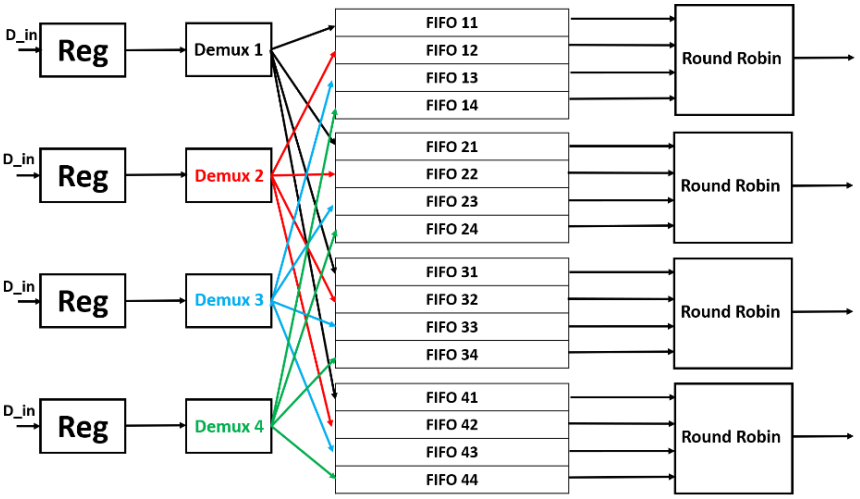
The implementation was by creating 4 components of registers, 4 components of demux, 4 components of Round Robins, and 16 Components of FIFOs, then they were mapped as shown in the figure.

The router module's logic has 4 processes:

In the first process, the sensitivity list is the selection variables of all the demuxes, so that the demux can choose which FIFO the packet should go to. The second process is responsible for syncing the demuxes, by using the wclock as a sensitivity signal. So whenever its on its rising edge, the write request happens throughout the FIFOs, only when 3 things happens, the write signal is 1, the reset is not 0, and the enable is available. The third process checks for the readclock, and reset, if the reset is 1 the state is the same, if the clock is triggered, the state is transitioned. The fourth process is responsible for the read syncing (round robin). Its implemented by switch case, where the variable is the current state, so at a given state, the current state increments to the next state, and the read sync is updated.



Fig#4.1: created block Diagram of FIFO

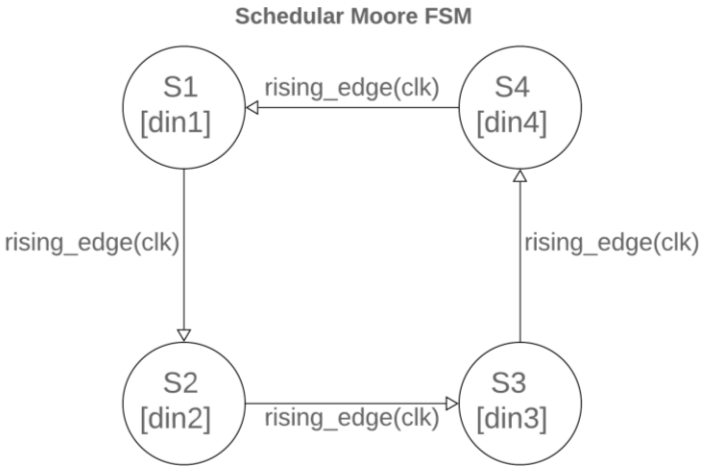


Fig#4.2: created block Diagram of router

5.0 Schedular Design and FSM Implementation:

Scheduler Design:

The scheduler is implemented using Moore way, as the output is generated depending on the current state only unlike Mealy which needs the current state and the input value. In the scheduler, the output is equal to the right din (input) according to the state the FSM is in right now. The scheduler is shown in the figure.



Fig#5.1: Schedular FSM

Different Implementation Styles:

Finite state machines can be implemented in VHDL by doing certain 4 tasks which are:

1) Asynchronous Reset (RS), 2) Update current state (CS),
3) Determine next state (NS), 4) Generate output (OP).

These tasks can be implemented in three different styles which are:

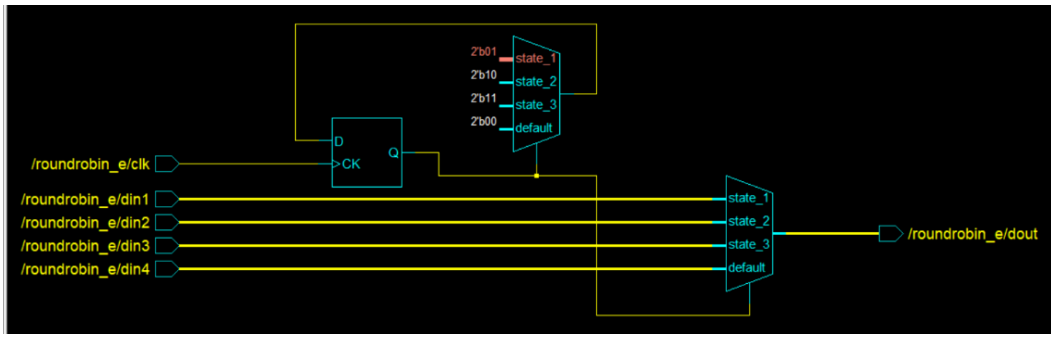
- One process: All 4 tasks are made in one process which has sensitivity list of (clock, reset, current state, inputs) for Mealy or (clock, reset, current state) for Moore. This style is not preferred as it is less readable from the others, but it is easier to debug.
- Two processes: The tasks are divided on two processes. The first do the RS and CS while the other do NS and OP. The sensitivity list of P1 is (clock, reset) while P2 is (current state, inputs) in a Mealy or Moore FSM.
- Three processes: P1 do RS and CS, P2 do NS and P3 do OP. Sensitivity list of P1 and P2 is the same for Mealy or Moore which is (clock, reset) for P1 and (current state, inputs) for P2 while P3 is (current state) for Moore and (current state, inputs) for Mealy.

Synthesis analysis:

The scheduler code is synthesized into:

1) Flip flop: As in the updating current state process there is an if condition that checks the rising edge of the clock and the sensitivity list is on the clock, so a flipflop is infered.
2) Two multiplexers: As there is a switch case on the present state which make two things, which are determine next state and generate the output. Therefore, they are two MUXs that have current state as selectors.

   The flip flop gets the next state as input and with the rising edge of clock it produces the current state which acts as a selector for the 2 MUXs, so that the first one produces the selected next state that goes into the flip flop while the second MUX produces the right output from the selected input.

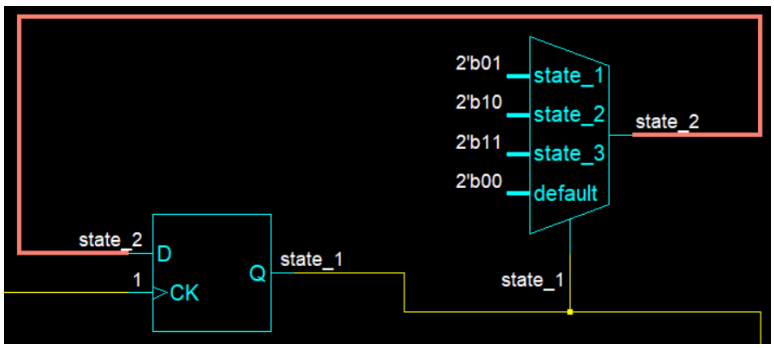


Fig#5.2: Schedular Schematic

Timing Analysis:

$T_C \geq t_{pcq} + t_{pd} + t_{setup}$

The clock time should be bigger than or equal to the propagation delay of the combinational circuit **plus** the clock to output propagation delay plus the setup time of the flip flop.



Fig#5.3: Schedular's critical path

In the scheduler the critical pass is the one between the flip flop and next state MUX.

Therefore, the minimum cock cycle can be is $\mathbf{t}_{pcq}$ of flip flop + t_{pd} of the MUX + t_{setup} of flip flop.

While t_{hold} of flip-flop must be less than or equal t_{cd} of the MUX plus t_{ccq} of flip flop. Lastly, slack = Tc − tsetup − tpcq + tpd.

Timing Report

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

Clock Signal	Clock buffer(FF name)	Load
clk	BUFGP	2

Asynchronous Control Signals Information:

No asynchronous control signals found in this design

Timing Summary:

Speed Grade: -3
Minimum period: 2.190ns (Maximum Frequency: 456.663MHz)
Minimum input arrival time before clock: No path found
Maximum output required time after clock: 5.021ns
Maximum combinational path delay: 5.402ns

Timing Details:

All values displayed in nanoseconds (ns)

==========

Timing constraint: Default period analysis for Clock 'clk'
Clock period: 2.190ns (frequency: 456.663MHz)
Total number of paths / destination ports: 3 / 2

Delay: 2.190ns (Levels of Logic = 1)
Source: current_state_FSM_FFd2 (FF)
Destination: current_state_FSM_FFd2 (FF)
Source Clock: clk rising
Destination Clock: clk rising

Data Path: current_state_FSM_FFd2 to current_state_FSM_FFd2

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
FD:C->Q	10	0.447	0.856	current_state_FSM_FFd2 (current_state_FSM_FFd2)
INV:I->O	1	0.206	0.579	current_state_FSM_FFd2-In1_INV_0 (current_state_FSM_FFd2-In)
FD:D		0.102		current_state_FSM_FFd2
Total		2.190ns (0.755ns logic, 1.435ns route) (34.5% logic, 65.5% route)		

==========

Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
Total number of paths / destination ports: 16 / 8

Offset: 5.021ns (Levels of Logic = 2)
Source: current_state_FSM_FFd2 (FF)
Destination: dout<7> (PAD)
Source Clock: clk rising

Data Path: current_state_FSM_FFd2 to dout<7>

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
FD:C->Q	10	0.447	1.221	current_state_FSM_FFd2 (current_state_FSM_FFd2)
LUT6:I0->O	1	0.203	0.579	Mmux_dout11 (dout_0_OBUF)
OBUF:I->O		2.571		dout_0_OBUF (dout<0>)
Total		5.021ns (3.221ns logic, 1.800ns route) (64.2% logic, 35.8% route)		

==========

Timing constraint: Default path analysis
Total number of paths / destination ports: 32 / 8

Delay: 5.402ns (Levels of Logic = 3)
Source: din2<7> (PAD)
Destination: dout<7> (PAD)

Data Path: din2<7> to dout<7>

Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
IBUF:I->O	1	1.222	0.827	din2_7_IBUF (din2_7_IBUF)
LUT6:I2->O	1	0.203	0.579	Mmux_dout81 (dout_7_OBUF)
OBUF:I->O		2.571		dout_7_OBUF (dout<7>)

```
----------------------------------------
Total                5.402ns (3.996ns logic, 1.406ns route)
                            (74.0% logic, 26.0% route)
==================================================
Cross Clock Domains Report:
--------------------------
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   2.190|         |         |         |
---------------+---------+---------+---------+---------+
==================================================
Total REAL time to Xst completion: 15.00 secs
Total CPU time to Xst completion: 14.98 secs
-->
Total memory usage is 4492644 kilobytes
Number of errors   :    0 (   0 filtered)
Number of warnings :    0 (   0 filtered)
Number of infos    :    0 (   0 filtered
```

6.0 Test and Simulation results:

Tested feature	***Inputs***								***Outputs***				***Delay***
	Data 1	***Data 2***	***Data 3***	***Data 4***	***wr 1***	***wr 2***	***wr 3***	***w r4***	***Dataout1***	***Dataout2***	***Dataout3***	***Dataout4***	
Output buffer4 (Testcase 1)	1000 0111	1000 1011	1001 0011	1010 0011	1	1	1	1				10000111	40 ns
												10001011	
												10010011	
												10100011	
Output buffer3 (Testcase 1)	1000 0110	1000 0110	1000 0110	1000 0110	1	1	1	1			10000110		40 ns
											10000110		
											10000110		
											10000110		
Output buffer2 (Testcase 3)	1000 0101	1000 0101	1000 0101	1000 0101	1	1	1	1		10000101			40 ns
										10000101			
										10000101			
										10000101			
Output buffer1 (Testcase 4)	1000 0100	1000 0100	1000 0100	1000 0100	1	1	1	1	10000100				40 ns
									10000100				
									10000100				
									10000100				

Fig#6.1: Test Plan

We used the exhaustive testing strategy. May be the exhaustive is not the best answer at all time, but in our case, we used it because by the 16 case we tested we have tested every input and output, so it is a not a waste of time, and 16 case is not that large number

of test cases relatively. At the end we have only increased the test case code by 3 lines for each case. so overall, we increased it by 12 lines of codes only.

7.0 Conclusion:

In conclusion, mapping the signals in router module was a challenge. The implementation was done by instancing of four register and demuxes and 16 FiFOs and 4 RR schedulers. The best implementation style for the FSM schedular is a two process Moore. Because the output doesn't depend on anything except on the current state. Also, two process because choosing 1 wouldn't offer as much organization as a one process style. Concerning the timing results: The time report: Minimum period: 2.190ns (Maximum Frequency: 456.663 MHz). For future work. We may consider working with other FSM styles. We may use mealy as its generally faster. (Although more is more stable and safer). An exhaustive test strategy was used.

8. Task Distribution List:

#	Student ID	Tasks
1	17P5026	the project workload is distributed **equally** among all team members
2	17P6069	
3	17P8182	
4	17P3061	
5	17P8042	

Appendix A – <<VHDL Model Source Code>>

8-bit register:

```
library IEEE;
USE IEEE.STD_Logic_1164.all;

ENTITY IB_E is
Port ( Data_in: in std_logic_vector( 7 downto 0);
      Clock_En: in std_logic;
      Clock: in std_logic;
      Reset: in std_logic;
      Data_out: out std_logic_Vector ( 7 downto 0 )
     );
END IB_E;


Architecture IB_A of IB_E is begin
PROCESS (Clock, Reset)
begin

if Reset = '1' then
  Data_out <= "00000000";

elsif rising_edge(Clock) and Clock_En = '1' then
Data_out <= Data_in;

            else
              null;
end if;
end PROCESS ;
end IB_A;
```

Demux:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

ENTITY demux_e IS
            PORT(d_in: IN std_logic_vector (7 downto 0); -- input data vector
                sel: IN std_logic_vector (1 downto 0); -- 2 selesction bits
                en: IN std_logic; -- enable flag
                -- output data vectors
                d_out1: OUT std_logic_vector (7 downto 0);
                d_out2: OUT std_logic_vector (7 downto 0);
                d_out3: OUT std_logic_vector (7 downto 0);
                d_out4: OUT std_logic_vector (7 downto 0));
END ENTITY demux_e;
```

```
ARCHITECTURE demux_a of demux_e is
BEGIN
 P1:PROCESS(en,d_in,sel) IS BEGIN -- sensitivity list contains all inputs
            if en = '1' THEN
             case sel is -- all cases of selection bits are dealt with
              when "00" => d_out1 <= d_in;
   when "01" => d_out2 <= d_in;
   when "10" => d_out3 <= d_in;
   when others => d_out4 <= d_in;
  end case;
 end if;
 END PROCESS P1;
END ARCHITECTURE demux_a;
```

Block Ram:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity BlockRam_E is
  Port (d_in: IN STD_LOGIC_VECTOR (7 downto 0);
       ADDRA: IN STD_LOGIC_VECTOR (2 downto 0);
       ADDRB: IN STD_LOGIC_VECTOR (2 downto 0);
       WEA, REA, CLKA, CLKB: IN STD_LOGIC;
       d_out: OUT STD_LOGIC_VECTOR (7 downto 0)
      );
   END entity BlockRam_E;

   Architecture block_ram_A of BlockRam_E is
    Signal Reg0: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg1: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg2: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg3: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg4: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg5: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg6: STD_LOGIC_VECTOR (7 downto 0);
    Signal Reg7: STD_LOGIC_VECTOR (7 downto 0);
    -- creating 8 registers

       Begin
     write: Process(CLKA)  IS BEGIN-- responsible for write
     IF rising_edge (CLKA) THEN -- check that clock A is rising
       IF WEA = '1' THEN -- check that write is enabled
         case ADDRA is
         when "000" =>
           Reg0 <= d_in;
         when "001" =>
           Reg1 <= d_in;
         when "010" =>
```

```
        Reg2 <= d_in;
       when "011" =>
        Reg3 <= d_in;
       when "100" =>
        Reg4 <= d_in;
       when "101" =>
        Reg5 <= d_in;
       when "110" =>
        Reg6 <= d_in;
       when others =>
        Reg7 <= d_in;
       end case;
    end if;
   end if;
    END Process write;

    read: Process (CLKB)  IS BEGIN -- responsible for read
    IF rising_edge(CLKB) THEN -- check that clock A is rising
     IF REA = '1' THEN -- check that read is enabled
       case ADDRB is
        when "000" =>
        d_out <= Reg0;
       when "001" =>
        d_out <= Reg1;
       when "010" =>
        d_out <= Reg2;
       when "011" =>
        d_out <= Reg3;
       when "100" =>
        d_out <= Reg4;
       when "101" =>
        d_out <= Reg5;
       when "110" =>
        d_out <= Reg6;
       when others =>
        d_out <= Reg7;
       end case;
    end if;
   end if;
    END Process read;
   END block_ram_A;
```

Gray counter:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY gray_counter IS
  PORT( En,clock,Reset : IN std_logic;
       Count_out: OUT std_logic_vector (2 downto 0));
END ENTITY gray_counter;
```

```
ARCHITECTURE behav OF gray_counter IS

 signal s_out :std_logic_vector (2 downto 0):="000";
 BEGIN
   Count_out <= s_out;
   cs: PROCESS (clock, Reset) is begin

      IF Reset = '1' THEN
         s_out <= "000";
      ELSIF (rising_edge(clock)and En = '1') THEN

        case s_out is
          when "000" =>    s_out <= "001";
          when "001" =>    s_out <= "011";
          when "011" =>    s_out <= "010";
          when "010" =>    s_out <= "110";
          when "110" =>    s_out <= "111";
          when "111" =>    s_out <= "101";
          when "101" =>    s_out <= "100";
          when "100" =>    s_out <= "000";
         when others =>    s_out <= "000";
        end case;

      END IF;

   END PROCESS cs;
END ARCHITECTURE behav;
```

Gray to binary converter:

```
-- Code for Gray to binary converter
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity gray_to_binary is
Port ( gray_in : in STD_LOGIC_VECTOR (2 downto 0); -- Gray code input
     bin_out : out STD_LOGIC_VECTOR (2 downto 0));     -- Binary code output
end gray_to_binary;

architecture Behavioral of gray_to_binary is begin

bin_out(2)<= gray_in(2);                                    -- most significant bit remains the same
bin_out(1)<= gray_in(2) xor gray_in(1);                     -- b(i) = b(i+1) xor g(i)
bin_out(0)<= gray_in(2) xor gray_in(1) xor gray_in(0);                  -- repeat until least significant bit

end Behavioral;
```

Fifo controller:

```
            LIBRARY IEEE;
USE IEEE.Std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY FIFO_C IS
  PORT (
    reset : IN STD_LOGIC;
    rdclk : IN STD_LOGIC;
    wrclk : IN STD_LOGIC;
    r_req : IN STD_LOGIC;
    w_req : IN STD_LOGIC;
    write_valid : OUT STD_LOGIC;
    read_valid : OUT STD_LOGIC;
    wr_ptr : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    rd_ptr : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    empty : OUT STD_LOGIC;
    full : OUT STD_LOGIC);
END FIFO_C;

Architecture FIFO_C_ARC OF FIFO_C IS

  COMPONENT gray_to_binary
  PORT (gray_in : in STD_LOGIC_VECTOR (2 downto 0); -- Gray code input
      bin_out : out STD_LOGIC_VECTOR (2 downto 0));      -- Binary code output
  END COMPONENT;

  COMPONENT gray_counter
    PORT( En,clock,Reset : IN std_logic;
       Count_out: OUT std_logic_vector (2 downto 0));
  END COMPONENT;

  signal R_valid : STD_LOGIC;
  signal W_valid : STD_LOGIC;
  signal R_PTR: STD_LOGIC_VECTOR (2 DOWNTO 0);
  signal W_PTR : STD_LOGIC_VECTOR (2 DOWNTO 0);
  signal gray_in_write : STD_LOGIC_VECTOR (2 DOWNTO 0);
  signal gray_in_read : STD_LOGIC_VECTOR (2 DOWNTO 0);

  BEGIN
    -- gray counter
    gc_read : gray_counter PORT MAP(R_valid,rdclk,reset,gray_in_read);
    gc_write : gray_counter PORT MAP(W_valid,wrclk,reset,gray_in_write);
    -- gray to binary convertor
    gtb_read : gray_to_binary PORT MAP(gray_in_read,R_PTR);
    gtb_write : gray_to_binary PORT MAP(gray_in_write,W_PTR);
    -- connections
    wr_ptr <= W_PTR ;
    rd_ptr <= R_PTR ;
    write_valid <= W_valid;
```

```
   read_valid <= R_valid;
   main : process(r_req,w_req,reset) IS
   begin
    IF(reset = '1') THEN
      full <= '0';
      empty <= '1';
    ELSIF(w_req = '1') THEN
      IF( (std_logic_vector( unsigned(W_PTR) + 1)  = R_PTR) or (W_PTR = "111" and R_PTR="000")) THEN--
cheacking if the memory is full
        full <= '1';
        empty <= '0';
        W_valid <='0';
       ELSE
        full <= '0';
        empty <= '0';
        W_valid <='1';
      END IF;
    ELSIF(r_req = '1')THEN
      IF(R_PTR = W_PTR)THEN-- checking if the memory is empty
        empty <= '1';
        full <= '0' ;
        R_valid <='0';
      Else
        full <= '0';
        empty <= '0';
        R_valid <='1';
      END IF;
    ELSE
      W_valid <='0';
      R_valid <='0';
    END IF;
   END PROCESS main;
END ARCHITECTURE FIFO_C_ARC;
```

FiFO:

```
LIBRARY IEEE;
USE IEEE.Std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY FIFO_E IS
  PORT (
   reset : IN STD_LOGIC;
   rclk : IN STD_LOGIC;
   wclk : IN STD_LOGIC;
   rreq : IN STD_LOGIC;
   wreq : IN STD_LOGIC;
   datain : IN STD_LOGIC_VECTOR (7 downto 0);
   dataout : OUT STD_LOGIC_VECTOR (7 downto 0);
   empty : OUT STD_LOGIC;
   full : OUT STD_LOGIC);
END FIFO_E;
```

```
Architecture FIFO_A OF FIFO_E IS
  COMPONENT  FIFO_C IS
  PORT (
    reset : IN STD_LOGIC;
    rdclk : IN STD_LOGIC;
    wrclk : IN STD_LOGIC;
    r_req : IN STD_LOGIC;
    w_req : IN STD_LOGIC;
    write_valid : OUT STD_LOGIC;
    read_valid : OUT STD_LOGIC;
    wr_ptr : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    rd_ptr : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    empty : OUT STD_LOGIC;
    full : OUT STD_LOGIC);
  END COMPONENT FIFO_C;
  COMPONENT BlockRam_E is
  Port (d_in: IN STD_LOGIC_VECTOR (7 downto 0);
        ADDRA: IN STD_LOGIC_VECTOR (2 downto 0);
        ADDRB: IN STD_LOGIC_VECTOR (2 downto 0);
        WEA, REA, CLKA, CLKB: IN STD_LOGIC;
        d_out: OUT STD_LOGIC_VECTOR (7 downto 0)
      );
  END COMPONENT BlockRam_E;
  -- SIGNALS  SIGNAL
  SIGNAL  s_reset : STD_LOGIC;
  SIGNAL  s_rclk : STD_LOGIC;
  SIGNAL  s_wclk : STD_LOGIC;
  SIGNAL  s_rreq : STD_LOGIC;
  SIGNAL  s_wreq : STD_LOGIC;
  SIGNAL  s_datain : STD_LOGIC_VECTOR (7 downto 0);
  SIGNAL  s_dataout : STD_LOGIC_VECTOR (7 downto 0);
  SIGNAL  s_empty : STD_LOGIC;
  SIGNAL  s_full : STD_LOGIC;
  SIGNAL  s_write_valid : STD_LOGIC;
  SIGNAL  s_read_valid: STD_LOGIC;
  SIGNAL  s_wr_ptr : STD_LOGIC_VECTOR (2 DOWNTO 0);
  SIGNAL s_rd_ptr: STD_LOGIC_VECTOR (2 DOWNTO 0);
  BEGIN
  -- connections
  --inputs
    s_reset <= reset;
    s_rclk <= rclk;
    s_wclk <= wclk;
    s_rreq <= rreq;
    s_wreq <= wreq;
    s_datain <= datain;
    --outputs
    dataout <= s_dataout;
    empty <= s_empty;
    full <= s_full;
  ----------
```

```
 fifocont:  FIFO_C PORT MAP(s_reset, s_rclk, s_wclk, s_rreq, s_wreq,s_write_valid,
                        s_read_valid, s_wr_ptr, s_rd_ptr, s_empty, s_full);

 memoryram:  BlockRam_E PORT MAP(s_datain, s_wr_ptr, s_rd_ptr, s_write_valid, s_read_valid, s_wclk, s_rclk,
s_dataout);
END ARCHITECTURE FIFO_A;
```

Round Robin:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
ENTITY RoundRobin_E is
  Port (clk: IN STD_LOGIC;
    din1: in STD_LOGIC_VECTOR (7 downto 0);
    din2: in STD_LOGIC_VECTOR (7 downto 0);
    din3: in STD_LOGIC_VECTOR (7 downto 0);
    din4: in STD_LOGIC_VECTOR (7 downto 0);
    dout: out STD_LOGIC_VECTOR (7 downto 0)
  );
End entity RoundRobin_E;
Architecture RoundRobin_A of RoundRobin_E IS
  TYPE state_type IS (state_1, state_2, state_3, state_4);
    SIGNAL current_state: state_type;
    SIGNAL next_state: state_type;
Begin
  checkclock: Process(clk)
  Begin
 IF rising_edge(clk) THEN
  current_state <= next_state;
END IF;
END Process checkclock;
updating_the_next_state: Process (current_state, din1, din2, din3, din4)
Begin
case current_state is
when state_1 =>
  next_state <= state_2;
  dout <= din1;
when state_2 =>
  next_state <= state_3;
  dout <= din2;
when state_3 =>
  next_state <= state_4;
  dout <= din3;
when others =>
  next_state <= state_1;
  dout <= din4;
  END case;
  END Process updating_the_next_state;
END RoundRobin_A;
```

Router:

```
LIBRARY IEEE;
USE IEEE.Std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY Router_E IS
  PORT (
    rst : IN STD_LOGIC;
    rclock : IN STD_LOGIC;
    wclock : IN STD_LOGIC;
    datai1 : IN STD_LOGIC_VECTOR (7 downto 0);
    wr1 : IN STD_LOGIC;
    datai2 : IN STD_LOGIC_VECTOR (7 downto 0);
    wr2 : IN STD_LOGIC;
    datai3 : IN STD_LOGIC_VECTOR (7 downto 0);
    wr3 : IN STD_LOGIC;
    datai4 : IN STD_LOGIC_VECTOR (7 downto 0);
    wr4 : IN STD_LOGIC;
    datao1 : OUT STD_LOGIC_VECTOR (7 downto 0);
    datao2 : OUT STD_LOGIC_VECTOR (7 downto 0);
    datao3 : OUT STD_LOGIC_VECTOR (7 downto 0);
    datao4 : OUT STD_LOGIC_VECTOR (7 downto 0));
END Router_E;
Architecture Router_A OF Router_E IS
  COMPONENT FIFO_E IS
  PORT (
    reset : IN STD_LOGIC;
    rclk : IN STD_LOGIC;
    wclk : IN STD_LOGIC;
    rreq : IN STD_LOGIC;
    wreq : IN STD_LOGIC;
    datain : IN STD_LOGIC_VECTOR (7 downto 0);
    dataout : OUT STD_LOGIC_VECTOR (7 downto 0);
    empty : OUT STD_LOGIC;
    full : OUT STD_LOGIC);
  END COMPONENT FIFO_E;
  COMPONENT RoundRobin_E is
  Port (clk: IN STD_LOGIC;
    din1: in STD_LOGIC_VECTOR (7 downto 0);
    din2: in STD_LOGIC_VECTOR (7 downto 0);
    din3: in STD_LOGIC_VECTOR (7 downto 0);
    din4: in STD_LOGIC_VECTOR (7 downto 0);
    dout: out STD_LOGIC_VECTOR (7 downto 0)
  );
End COMPONENT RoundRobin_E;
COMPONENT demux_e IS
            PORT(d_in: IN std_logic_vector (7 downto 0); -- input data vector
                sel: IN std_logic_vector (1 downto 0); -- 2 selesction bits
                en: IN std_logic; -- enable flag
                -- output data vectors
                d_out1: OUT std_logic_vector (7 downto 0);
```

```
            d_out2: OUT std_logic_vector (7 downto 0);
            d_out3: OUT std_logic_vector (7 downto 0);
            d_out4: OUT std_logic_vector (7 downto 0));
END COMPONENT demux_e;
COMPONENT IB_E is
Port ( Data_in: in std_logic_vector( 7 downto 0);
      Clock_En: in std_logic;
      Clock: in std_logic;
      Reset: in std_logic;
      Data_out: out std_logic_Vector ( 7 downto 0 )
     );
END COMPONENT IB_E;
    SIGNAL  s_rst : STD_LOGIC;
    SIGNAL  s_rclock : STD_LOGIC;
    SIGNAL  s_wclock : STD_LOGIC;
    SIGNAL  s_datai1 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_wr1 : STD_LOGIC;
    SIGNAL  s_datai2 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_wr2 :  STD_LOGIC;
    SIGNAL  s_datai3 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_wr3 : STD_LOGIC;
    SIGNAL  s_datai4 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_wr4 : STD_LOGIC;
    SIGNAL  s_datao1 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_datao2 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_datao3 : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_datao4 : STD_LOGIC_VECTOR (7 downto 0);
    ---------/FIFO EMPTY&FULL/---------
    SIGNAL  s_FIFO11_empty : STD_LOGIC;
    SIGNAL  s_FIFO11_full : STD_LOGIC;
    SIGNAL  s_FIFO12_empty : STD_LOGIC;
    SIGNAL  s_FIFO12_full : STD_LOGIC;
    SIGNAL  s_FIFO13_empty : STD_LOGIC;
    SIGNAL  s_FIFO13_full : STD_LOGIC;
    SIGNAL  s_FIFO14_empty : STD_LOGIC;
    SIGNAL  s_FIFO14_full : STD_LOGIC;
    ----------------------------------
    SIGNAL  s_FIFO21_empty : STD_LOGIC;
    SIGNAL  s_FIFO21_full : STD_LOGIC;
    SIGNAL  s_FIFO22_empty : STD_LOGIC;
    SIGNAL  s_FIFO22_full : STD_LOGIC;
    SIGNAL  s_FIFO23_empty : STD_LOGIC;
    SIGNAL  s_FIFO23_full : STD_LOGIC;
    SIGNAL  s_FIFO24_empty : STD_LOGIC;
    SIGNAL  s_FIFO24_full : STD_LOGIC;
    ----------------------------------
    SIGNAL  s_FIFO31_empty : STD_LOGIC;
    SIGNAL  s_FIFO31_full : STD_LOGIC;
    SIGNAL  s_FIFO32_empty : STD_LOGIC;
    SIGNAL  s_FIFO32_full : STD_LOGIC;
    SIGNAL  s_FIFO33_empty : STD_LOGIC;
```

```
SIGNAL  s_FIFO33_full : STD_LOGIC;
SIGNAL  s_FIFO34_empty : STD_LOGIC;
SIGNAL  s_FIFO34_full : STD_LOGIC;
-----------------------------------
SIGNAL  s_FIFO41_empty : STD_LOGIC;
SIGNAL  s_FIFO41_full : STD_LOGIC;
SIGNAL  s_FIFO42_empty : STD_LOGIC;
SIGNAL  s_FIFO42_full : STD_LOGIC;
SIGNAL  s_FIFO43_empty : STD_LOGIC;
SIGNAL  s_FIFO43_full : STD_LOGIC;
SIGNAL  s_FIFO44_empty : STD_LOGIC;
SIGNAL  s_FIFO44_full : STD_LOGIC;
----------/FIFO request signals/-----------
SIGNAL  s_FIFO11_read_request : STD_LOGIC;
SIGNAL  s_FIFO11_write_request : STD_LOGIC;
SIGNAL  s_FIFO12_read_request : STD_LOGIC;
SIGNAL  s_FIFO12_write_request : STD_LOGIC;
SIGNAL  s_FIFO13_read_request : STD_LOGIC;
SIGNAL  s_FIFO13_write_request : STD_LOGIC;
SIGNAL  s_FIFO14_read_request : STD_LOGIC;
SIGNAL  s_FIFO14_write_request : STD_LOGIC;
-----------------------------------------
SIGNAL  s_FIFO21_read_request : STD_LOGIC;
SIGNAL  s_FIFO21_write_request : STD_LOGIC;
SIGNAL  s_FIFO22_read_request : STD_LOGIC;
SIGNAL  s_FIFO22_write_request : STD_LOGIC;
SIGNAL  s_FIFO23_read_request : STD_LOGIC;
SIGNAL  s_FIFO23_write_request : STD_LOGIC;
SIGNAL  s_FIFO24_read_request : STD_LOGIC;
SIGNAL  s_FIFO24_write_request : STD_LOGIC;
-----------------------------------------
SIGNAL  s_FIFO31_read_request : STD_LOGIC;
SIGNAL  s_FIFO31_write_request : STD_LOGIC;
SIGNAL  s_FIFO32_read_request : STD_LOGIC;
SIGNAL  s_FIFO32_write_request : STD_LOGIC;
SIGNAL  s_FIFO33_read_request : STD_LOGIC;
SIGNAL  s_FIFO33_write_request : STD_LOGIC;
SIGNAL  s_FIFO34_read_request : STD_LOGIC;
SIGNAL  s_FIFO34_write_request : STD_LOGIC;
-----------------------------------------
SIGNAL  s_FIFO41_read_request : STD_LOGIC;
SIGNAL  s_FIFO41_write_request : STD_LOGIC;
SIGNAL  s_FIFO42_read_request : STD_LOGIC;
SIGNAL  s_FIFO42_write_request : STD_LOGIC;
SIGNAL  s_FIFO43_read_request : STD_LOGIC;
SIGNAL  s_FIFO43_write_request : STD_LOGIC;
SIGNAL  s_FIFO44_read_request : STD_LOGIC;
SIGNAL  s_FIFO44_write_request : STD_LOGIC;
----------------/syncronize signals/-----------------
TYPE state_type IS (state_1, state_2, state_3, state_4);
SIGNAL s_current_state: state_type;
```

```
SIGNAL s_next_state: state_type;
SIGNAL s_current_read_sync : STD_LOGIC_VECTOR (3 downto 0);
--------------------/IB Outputs/--------------------
SIGNAL  s_reg1_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_reg2_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_reg3_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_reg4_datao : STD_LOGIC_VECTOR (7 downto 0);
--------------------/DeMux ENABLE/---------------------
SIGNAL  s_demux1_en : STD_LOGIC;
SIGNAL  s_demux2_en : STD_LOGIC;
SIGNAL  s_demux3_en : STD_LOGIC;
SIGNAL  s_demux4_en : STD_LOGIC;
--------------------/DeMux Inputs/---------------------
SIGNAL  s_demux1_datai : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux2_datai : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux3_datai : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux4_datai : STD_LOGIC_VECTOR (7 downto 0);
--------------------/DeMux_1 Outputs/---------------------
SIGNAL  s_demux1_datao1 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux1_datao2 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux1_datao3 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux1_datao4 : STD_LOGIC_VECTOR (7 downto 0);
--------------------/DeMux_2 Outputs/---------------------
SIGNAL  s_demux2_datao1 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux2_datao2 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux2_datao3 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux2_datao4 : STD_LOGIC_VECTOR (7 downto 0);
--------------------/DeMux_3 Outputs/---------------------
SIGNAL  s_demux3_datao1 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux3_datao2 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux3_datao3 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux3_datao4 : STD_LOGIC_VECTOR (7 downto 0);
--------------------/DeMux_1 Outputs/---------------------
SIGNAL  s_demux4_datao1 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux4_datao2 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux4_datao3 : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_demux4_datao4 : STD_LOGIC_VECTOR (7 downto 0);
--------------------/FIFO outputs/----------------------
SIGNAL  s_FIFO11_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO12_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO13_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO14_datao : STD_LOGIC_VECTOR (7 downto 0);
-----------------------------------------------------
SIGNAL  s_FIFO21_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO22_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO23_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO24_datao : STD_LOGIC_VECTOR (7 downto 0);
-----------------------------------------------------
SIGNAL  s_FIFO31_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO32_datao : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL  s_FIFO33_datao : STD_LOGIC_VECTOR (7 downto 0);
```

```
    SIGNAL  s_FIFO34_datao : STD_LOGIC_VECTOR (7 downto 0);
    ------------------------------------------------------
    SIGNAL  s_FIFO41_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_FIFO42_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_FIFO43_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_FIFO44_datao : STD_LOGIC_VECTOR (7 downto 0);
    ----------------/DeMux selection lines/-----------------
    SIGNAL  s_demux1_sel : STD_LOGIC_VECTOR (1 downto 0);
    SIGNAL  s_demux2_sel : STD_LOGIC_VECTOR (1 downto 0);
    SIGNAL  s_demux3_sel : STD_LOGIC_VECTOR (1 downto 0);
    SIGNAL  s_demux4_sel : STD_LOGIC_VECTOR (1 downto 0);
    ----------------------/scheduler output signals/----------------------
    SIGNAL  s_scheduler1_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_scheduler2_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_scheduler3_datao : STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL  s_scheduler4_datao : STD_LOGIC_VECTOR (7 downto 0);
    --------------------/FIFO write selctor/---------------------
    SIGNAL  s_FIFO1_en : STD_LOGIC_VECTOR (3 downto 0);
    SIGNAL  s_FIFO2_en : STD_LOGIC_VECTOR (3 downto 0);
    SIGNAL  s_FIFO3_en : STD_LOGIC_VECTOR (3 downto 0);
    SIGNAL  s_FIFO4_en : STD_LOGIC_VECTOR (3 downto 0);

BEGIN
    ------------/connections/--------------
    s_rst <= rst;
    s_rclock <= rclock;
    s_wclock <= wclock;
    ---------------------------------------
    s_datai1 <= datai1 ;
    s_datai2 <= datai2 ;
    s_datai3 <= datai3 ;
    s_datai4 <= datai4 ;
    ---------------------------------------
    datao1 <= s_datao1;
    datao2 <= s_datao2;
    datao3 <= s_datao3;
    datao4 <= s_datao4;
    ---------------------------------------
    s_wr1 <= wr1;
    s_wr2 <= wr2;
    s_wr3 <= wr3;
    s_wr4 <= wr4;
    ----------------------------------------
    s_demux1_datai <= s_reg1_datao;
    s_demux2_datai <= s_reg2_datao;
    s_demux3_datai <= s_reg3_datao;
    s_demux4_datai <= s_reg4_datao;
    ----------------------------------------
    s_demux1_en <= (s_wr1 and s_wclock);
    s_demux2_en <= (s_wr2 and s_wclock);
    s_demux3_en <= (s_wr3 and s_wclock);
```

```
s_demux4_en <= (s_wr4 and s_wclock);
----------------------------------------
s_datao1 <= s_scheduler1_datao;
s_datao2 <= s_scheduler2_datao;
s_datao3 <= s_scheduler3_datao;
s_datao4 <= s_scheduler4_datao;
----------------------------------------
s_FIFO11_read_request <= (s_current_read_sync(0) and (not s_FIFO11_empty));
s_FIFO21_read_request <= (s_current_read_sync(0) and (not s_FIFO21_empty));
s_FIFO31_read_request <= (s_current_read_sync(0) and (not s_FIFO31_empty));
s_FIFO41_read_request <= (s_current_read_sync(0) and (not s_FIFO41_empty));
----------------------------------------
s_FIFO12_read_request <= (s_current_read_sync(1) and (not s_FIFO12_empty));
s_FIFO22_read_request <= (s_current_read_sync(1) and (not s_FIFO22_empty));
s_FIFO32_read_request <= (s_current_read_sync(1) and (not s_FIFO32_empty));
s_FIFO42_read_request <= (s_current_read_sync(1) and (not s_FIFO42_empty));
----------------------------------------
s_FIFO13_read_request <= (s_current_read_sync(2) and (not s_FIFO13_empty));
s_FIFO23_read_request <= (s_current_read_sync(2) and (not s_FIFO23_empty));
s_FIFO33_read_request <= (s_current_read_sync(2) and (not s_FIFO33_empty));
s_FIFO43_read_request <= (s_current_read_sync(2) and (not s_FIFO43_empty));
----------------------------------------
s_FIFO14_read_request <= (s_current_read_sync(3) and (not s_FIFO14_empty));
s_FIFO24_read_request <= (s_current_read_sync(3) and (not s_FIFO24_empty));
s_FIFO34_read_request <= (s_current_read_sync(3) and (not s_FIFO34_empty));
s_FIFO44_read_request <= (s_current_read_sync(3) and (not s_FIFO44_empty));
----------------------------------------
--/DeMux selection lines Connections/--
s_demux1_sel(0) <= s_datai1(0);
s_demux1_sel(1) <= s_datai1(1);
----------------------------------------
s_demux2_sel(0) <= s_datai2(0);
s_demux2_sel(1) <= s_datai2(1);
----------------------------------------
s_demux3_sel(0) <= s_datai3(0);
s_demux3_sel(1) <= s_datai3(1);
----------------------------------------
s_demux4_sel(0) <= s_datai4(0);
s_demux4_sel(1) <= s_datai4(1);
------------------------/Input Buffers/-------------------------
IB_1 : IB_E PORT MAP(s_datai1,s_wr1,s_wclock,s_rst,s_reg1_datao);
IB_2 : IB_E PORT MAP(s_datai2,s_wr2,s_wclock,s_rst,s_reg2_datao);
IB_3 : IB_E PORT MAP(s_datai3,s_wr3,s_wclock,s_rst,s_reg3_datao);
IB_4 : IB_E PORT MAP(s_datai4,s_wr4,s_wclock,s_rst,s_reg4_datao);
-------------------------------------------------------------/Demux/-------------------------------------------------------------
DeMux_1 : demux_e PORT
MAP(s_demux1_datai,s_demux1_sel,s_demux1_en,s_demux1_datao1,s_demux1_datao2,s_demux1_datao3,s_de
mux1_datao4);
DeMux_2 : demux_e PORT
MAP(s_demux2_datai,s_demux2_sel,s_demux2_en,s_demux2_datao1,s_demux2_datao2,s_demux2_datao3,s_de
mux2_datao4);
```

```
   DeMux_3 : demux_e PORT
MAP(s_demux3_datai,s_demux3_sel,s_demux3_en,s_demux3_datao1,s_demux3_datao2,s_demux3_datao3,s_de
mux3_datao4);
   DeMux_4 : demux_e PORT
MAP(s_demux4_datai,s_demux4_sel,s_demux4_en,s_demux4_datao1,s_demux4_datao2,s_demux4_datao3,s_de
mux4_datao4);
   ----------------------------------------------------------------------------------------------------------------------------------------
   FIFO_11 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO11_read_request,s_FIFO11_write_request,s_demux1_datao1,s_FIFO11_datao,
s_FIFO11_empty,s_FIFO11_full);
   FIFO_12 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO12_read_request,s_FIFO12_write_request,s_demux2_datao1,s_FIFO12_datao,
s_FIFO12_empty,s_FIFO12_full);
   FIFO_13 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO13_read_request,s_FIFO13_write_request,s_demux3_datao1,s_FIFO13_datao,
s_FIFO13_empty,s_FIFO13_full);
   FIFO_14 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO14_read_request,s_FIFO14_write_request,s_demux4_datao1,s_FIFO14_datao,
s_FIFO14_empty,s_FIFO14_full);
 ----------------------------------------------------------------------------------------------------------------------------------------
   FIFO_21 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO21_read_request,s_FIFO21_write_request,s_demux1_datao2,s_FIFO21_datao,
s_FIFO21_empty,s_FIFO21_full);
   FIFO_22 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO22_read_request,s_FIFO22_write_request,s_demux2_datao2,s_FIFO22_datao,
s_FIFO22_empty,s_FIFO22_full);
   FIFO_23 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO23_read_request,s_FIFO23_write_request,s_demux3_datao2,s_FIFO23_datao,
s_FIFO23_empty,s_FIFO23_full);
   FIFO_24 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO24_read_request,s_FIFO24_write_request,s_demux4_datao2,s_FIFO24_datao,
s_FIFO24_empty,s_FIFO24_full);
   ----------------------------------------------------------------------------------------------------------------------------------------
   FIFO_31 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO31_read_request,s_FIFO31_write_request,s_demux1_datao3,s_FIFO31_datao,
s_FIFO31_empty,s_FIFO31_full);
   FIFO_32 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO32_read_request,s_FIFO32_write_request,s_demux2_datao3,s_FIFO32_datao,
s_FIFO32_empty,s_FIFO32_full);
   FIFO_33 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO33_read_request,s_FIFO33_write_request,s_demux3_datao3,s_FIFO33_datao,
s_FIFO33_empty,s_FIFO33_full);
   FIFO_34 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO34_read_request,s_FIFO34_write_request,s_demux4_datao3,s_FIFO34_datao,
s_FIFO34_empty,s_FIFO34_full);
   ----------------------------------------------------------------------------------------------------------------------------------------
   FIFO_41 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO41_read_request,s_FIFO41_write_request,s_demux1_datao4,s_FIFO41_datao,
s_FIFO41_empty,s_FIFO41_full);
```

```
   FIFO_42 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO42_read_request,s_FIFO42_write_request,s_demux2_datao4,s_FIFO42_datao,
s_FIFO42_empty,s_FIFO42_full);
   FIFO_43 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO43_read_request,s_FIFO43_write_request,s_demux3_datao4,s_FIFO43_datao,
s_FIFO43_empty,s_FIFO43_full);
   FIFO_44 : FIFO_E PORT
MAP(s_rst,s_rclock,s_wclock,s_FIFO44_read_request,s_FIFO44_write_request,s_demux4_datao4,s_FIFO44_datao,
s_FIFO44_empty,s_FIFO44_full);
   ----------------------------------------------------------------/scheduler/-----------------------------------------------------------
   scheduler_1 : RoundRobin_E PORT
MAP(s_rclock,s_FIFO11_datao,s_FIFO12_datao,s_FIFO13_datao,s_FIFO14_datao,s_scheduler1_datao);
   scheduler_2 : RoundRobin_E PORT
MAP(s_rclock,s_FIFO21_datao,s_FIFO22_datao,s_FIFO23_datao,s_FIFO24_datao,s_scheduler2_datao);
   scheduler_3 : RoundRobin_E PORT
MAP(s_rclock,s_FIFO31_datao,s_FIFO32_datao,s_FIFO33_datao,s_FIFO34_datao,s_scheduler3_datao);
   scheduler_4 : RoundRobin_E PORT
MAP(s_rclock,s_FIFO41_datao,s_FIFO42_datao,s_FIFO43_datao,s_FIFO44_datao,s_scheduler4_datao);
   ----------------------------------------------------------------/sync read request/-----------------------------------------------------
   FIFO_SEL : PROCESS(s_demux1_sel,s_demux2_sel,s_demux3_sel,s_demux4_sel)
   BEGIN
    case s_demux1_sel is
          when "00" =>    s_FIFO1_en <= "0001";
          when "01" =>    s_FIFO1_en <= "0010";
          when "10" =>    s_FIFO1_en <= "0100";
          when "11" =>    s_FIFO1_en <= "1000";
         when others =>    s_FIFO1_en <= "0000";
    end case;
    case s_demux2_sel is
          when "00" =>    s_FIFO2_en <= "0001";
          when "01" =>    s_FIFO2_en <= "0010";
          when "10" =>    s_FIFO2_en <= "0100";
          when "11" =>    s_FIFO2_en <= "1000";
         when others =>    s_FIFO2_en <= "0000";
    end case;
    case s_demux3_sel is
          when "00" =>    s_FIFO3_en <= "0001";
          when "01" =>    s_FIFO3_en <= "0010";
          when "10" =>    s_FIFO3_en <= "0100";
          when "11" =>    s_FIFO3_en <= "1000";
         when others =>    s_FIFO3_en <= "0000";
    end case;
    case s_demux4_sel is
          when "00" =>    s_FIFO4_en <= "0001";
          when "01" =>    s_FIFO4_en <= "0010";
          when "10" =>    s_FIFO4_en <= "0100";
          when "11" =>    s_FIFO4_en <= "1000";
         when others =>    s_FIFO4_en <= "0000";
    end case;
   END PROCESS FIFO_SEL;
   write_sync_demux1 : PROCESS(s_wclock)
```

```
Begin
 if rising_edge(s_wclock) THEN
   s_FIFO11_write_request <= (s_wr1 and (not s_FIFO11_full) and (not s_rst) and s_FIFO1_en(0));
   s_FIFO21_write_request <= (s_wr1 and (not s_FIFO21_full) and (not s_rst) and s_FIFO1_en(1));
   s_FIFO31_write_request <= (s_wr1 and (not s_FIFO31_full) and (not s_rst) and s_FIFO1_en(2));
   s_FIFO41_write_request <= (s_wr1 and (not s_FIFO41_full) and (not s_rst) and s_FIFO1_en(3));
   ----------------------------------------
   s_FIFO12_write_request <= (s_wr2 and (not s_FIFO12_full) and (not s_rst) and s_FIFO2_en(0));
   s_FIFO22_write_request <= (s_wr2 and (not s_FIFO22_full) and (not s_rst) and s_FIFO2_en(1));
   s_FIFO32_write_request <= (s_wr2 and (not s_FIFO32_full) and (not s_rst) and s_FIFO2_en(2));
   s_FIFO42_write_request <= (s_wr2 and (not s_FIFO42_full) and (not s_rst) and s_FIFO2_en(3));
   ----------------------------------------
   s_FIFO13_write_request <= (s_wr3 and (not s_FIFO13_full) and (not s_rst) and s_FIFO3_en(0));
   s_FIFO23_write_request <= (s_wr3 and (not s_FIFO23_full) and (not s_rst) and s_FIFO3_en(1));
   s_FIFO33_write_request <= (s_wr3 and (not s_FIFO33_full) and (not s_rst) and s_FIFO3_en(2));
   s_FIFO43_write_request <= (s_wr3 and (not s_FIFO43_full) and (not s_rst) and s_FIFO3_en(3));
   ----------------------------------------
   s_FIFO14_write_request <= (s_wr4 and (not s_FIFO14_full) and (not s_rst) and s_FIFO4_en(0));
   s_FIFO24_write_request <= (s_wr4 and (not s_FIFO24_full) and (not s_rst) and s_FIFO4_en(1));
   s_FIFO34_write_request <= (s_wr4 and (not s_FIFO34_full) and (not s_rst) and s_FIFO4_en(2));
   s_FIFO44_write_request <= (s_wr4 and (not s_FIFO44_full) and (not s_rst) and s_FIFO4_en(3));
 END if;
END PROCESS write_sync_demux1;
-------------------------------------------------------------------------------------------------------------------------------------------------------
checkclock: Process(s_rclock,s_rst)
Begin
 IF s_rst = '1' THEN
   s_current_state <= state_1 ;
 ELSIF rising_edge(s_rclock) THEN
   s_current_state <= s_next_state;
 END IF;
END Process checkclock;
read_sync: Process (s_current_state,s_current_read_sync)
Begin
case s_current_state is
when state_1 =>
 s_next_state <= state_2;
 s_current_read_sync <= "0001";
when state_2 =>
 s_next_state <= state_3;
 s_current_read_sync <= "0010";
when state_3 =>
 s_next_state <= state_4;
 s_current_read_sync <= "0100";
when others =>
 s_next_state <= state_1;
 s_current_read_sync <= "1000";
END case;
END Process read_sync;
-------------------------------------------------------------------------------------------------------------------------------------------------------
END ARCHITECTURE Router_A;
```

Appendix B – <<VHDL Test Bench Source Code>>

```
LIBRARY IEEE;
USE IEEE.Std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Router_T_E IS
END Router_T_E;

Architecture Router_T_A OF Router_T_E IS
  COMPONENT Router_E IS
    PORT (
      rst : IN STD_LOGIC;
      rclock : IN STD_LOGIC;
      wclock : IN STD_LOGIC;
      datai1 : IN STD_LOGIC_VECTOR (7 downto 0);
      wr1 : IN STD_LOGIC;
      datai2 : IN STD_LOGIC_VECTOR (7 downto 0);
      wr2 : IN STD_LOGIC;
      datai3 : IN STD_LOGIC_VECTOR (7 downto 0);
      wr3 : IN STD_LOGIC;
      datai4 : IN STD_LOGIC_VECTOR (7 downto 0);
      wr4 : IN STD_LOGIC;
      datao1 : OUT STD_LOGIC_VECTOR (7 downto 0);
      datao2 : OUT STD_LOGIC_VECTOR (7 downto 0);
      datao3 : OUT STD_LOGIC_VECTOR (7 downto 0);
      datao4 : OUT STD_LOGIC_VECTOR (7 downto 0));
  END COMPONENT Router_E;
      SIGNAL s_rst :  STD_LOGIC;
      SIGNAL s_rclock :  STD_LOGIC;
      SIGNAL s_wclock :  STD_LOGIC;
      SIGNAL s_datai1 :  STD_LOGIC_VECTOR (7 downto 0);
      SIGNAL s_wr1 :  STD_LOGIC;
      SIGNAL s_datai2 :  STD_LOGIC_VECTOR (7 downto 0);
      SIGNAL s_wr2 :  STD_LOGIC;
```

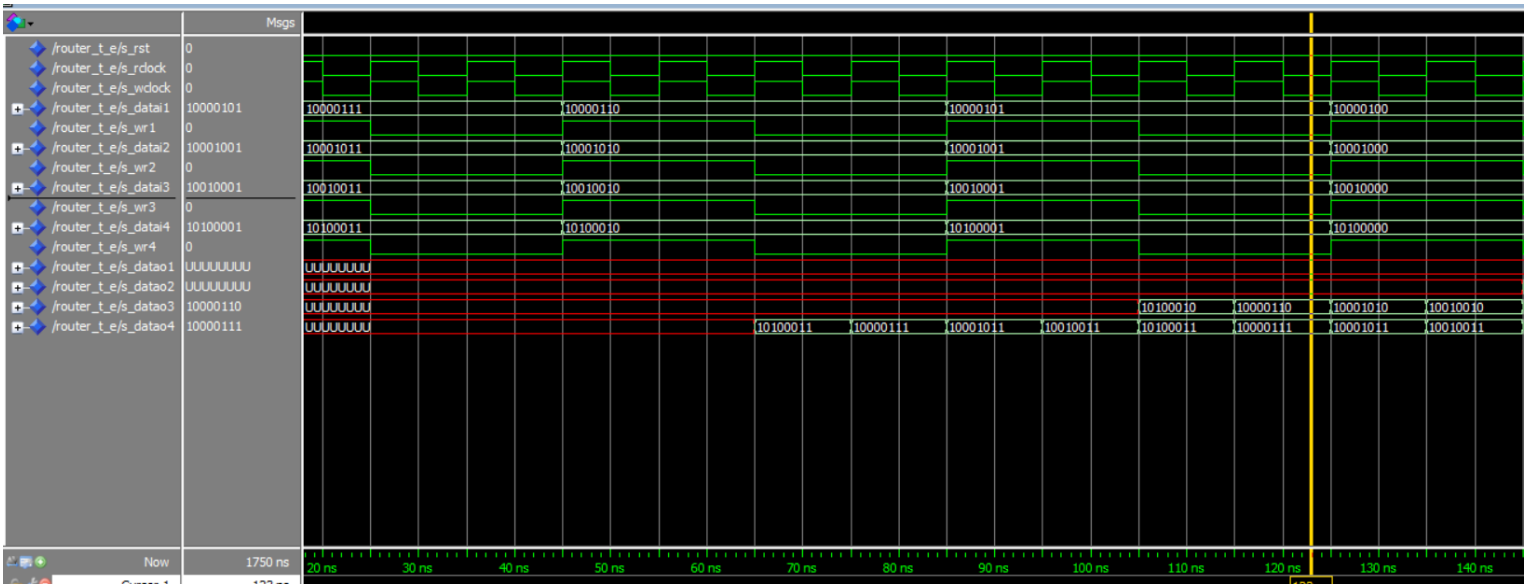
```
    SIGNAL s_datai3 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL s_wr3 :  STD_LOGIC;
    SIGNAL s_datai4 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL s_wr4 :  STD_LOGIC;
    SIGNAL s_datao1 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL s_datao2 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL s_datao3 :  STD_LOGIC_VECTOR (7 downto 0);
    SIGNAL s_datao4 :  STD_LOGIC_VECTOR (7 downto 0);
 BEGIN
    ROUTER : Router_E PORT
MAP(s_rst,s_rclock,s_wclock,s_datai1,s_wr1,s_datai2,s_wr2,s_datai3,s_wr3,s_datai4,s_wr4,s_datao1,s_datao2,s_
datao3,s_datao4);
    read_clock: PROCESS IS
    BEGIN
       s_rclock <= '0' , '1' AFTER 5 ns;
       WAIT FOR 10 ns;
    END PROCESS read_clock;
   write_clock: PROCESS IS
    BEGIN
       s_wclock <= '0' , '1' AFTER 5 ns;
       WAIT FOR 10 ns;
  END PROCESS write_clock;
  --test case 1 : reset && input data from all input buffers to output buffer 4
  p1: process is begin
  s_rst <= '1';
  WAIT FOR 5 ns;
  s_rst <= '0';
  s_datai1 <= "10000111"; --FIFO41
  s_datai2 <= "10001011"; --FIFO42
  s_datai3 <= "10010011"; --FIFO43
  s_datai4 <= "10100011"; --FIFO44
  s_wr1 <= '1';
  s_wr2 <= '1';
  s_wr3 <= '1';
```

```
s_wr4 <= '1';
WAIT FOR 20 ns;
s_wr1 <= '0';
s_wr2 <= '0';
s_wr3 <= '0';
s_wr4 <= '0';
wait FOR 20 ns;
-- test case 2: input data from all input buffers to output buffer 3
s_datai1 <= "10000110"; --FIFO31
s_datai2 <= "10001010"; --FIFO32
s_datai3 <= "10010010"; --FIFO33
s_datai4 <= "10100010"; --FIFO34
s_wr1 <= '1';
s_wr2 <= '1';
s_wr3 <= '1';
s_wr4 <= '1';
WAIT FOR 20 ns;
assert s_datao4 <= "10100011" -- value in datai4
report "Test case 1 error"
severity warning;
s_wr1 <= '0';
s_wr2 <= '0';
s_wr3 <= '0';
s_wr4 <= '0';
wait FOR 20 ns;
-- test case 3: input data from all input buffers to output buffer 2
s_datai1 <= "10000101"; --FIFO21
s_datai2 <= "10001001"; --FIFO22
s_datai3 <= "10010001"; --FIFO23
s_datai4 <= "10100001"; --FIFO24
s_wr1 <= '1';
s_wr2 <= '1';
s_wr3 <= '1';
s_wr4 <= '1';
```

```
   WAIT FOR 20 ns;
   assert s_datao3 <= "10100010"  -- value in datai4
   report "Test case 2 error"
   severity warning;
   s_wr1 <= '0';
   s_wr2 <= '0';
   s_wr3 <= '0';
   s_wr4 <= '0';
   wait FOR 20 ns;
   -- test case 4: input data from all input buffers to output buffer 1
   s_datai1 <= "10000100"; --FIFO11
   s_datai2 <= "10001000"; --FIFO12
   s_datai3 <= "10010000"; --FIFO13
   s_datai4 <= "10100000"; --FIFO14
   s_wr1 <= '1';
   s_wr2 <= '1';
   s_wr3 <= '1';
   s_wr4 <= '1';
   WAIT FOR 20 ns;
   assert s_datao2 <= "10100001" -- value in datails
   report "Test case 3 error"
   severity warning;
   s_wr1 <= '0';
   s_wr2 <= '0';
   s_wr3 <= '0';
   s_wr4 <= '0';
   wait For 40 ns;
   assert s_datao1 <= "10100000" -- value in datai4
   report "Test case 4 error"
   severity warning;
   wait FOR 40 ns;
   END PROCESS p1 ;
END Architecture Router_T_A;
```

Appendix C – <<Simulation Waveform Output>>

Results of testcase 1 and 2:



Results of testcase 3 and 4: