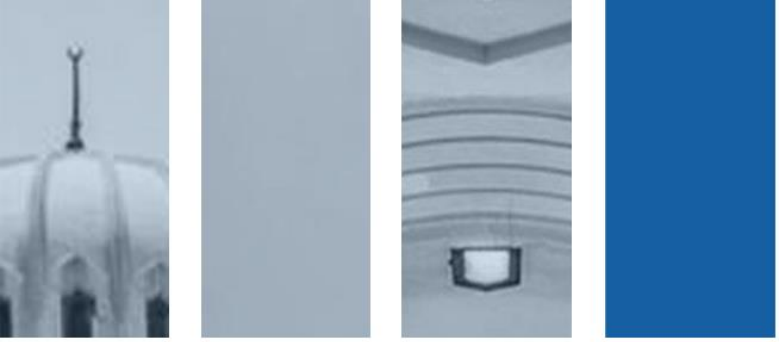


Networks-On-Chip Router

Program: CESS

Course Code: CSE312

Course Name:

Electronic Design Automation

Submitted to:

Dr Haytham Azmi El-Miligi
Eng Michael Hany Mofeed
Eng Omar Moataz Saad

Submitted by:

1. ***Farah Loay*** ***18P7388***
2. ***Hana Yasser*** ***18P5007***
3. ***Ilaria Refaat*** ***18P3050***
4. ***Laila Mohamed*** ***18P9654***
5. ***Malak Mohamed*** ***18P9114***
6. ***Mariam Mahmoud*** ***18P9102***

Students' Full Names:

Farah Loay Hamed Alanany	18P7388
Hana Yasser Amged Mohamed El Sokkary	18P5007
Ilaria Refaat Ghobrial Messiha	18P3050
Laila Mohamed Mohamed Abdelfattah Aborizka	18P9654
Malak Mohamed Gadelrab Abdou Madkour	18P9114
Mariam Mahmoud Fawzi Mostafa Galal	18P1092

Networks-on-Chip (NoC) Router

Abstract

This report is a documentation on the NoC Router from what it is, to its design flow, design implementation, scheduler, the modules that makes it up, their synthesizable codes and how they behave; in addition to some literature reviews about the Noc Router, types of routers generally and the method of building a simple chip. We have used both ModelSim and Xilinx ISE in this project for simulation and synthesis purposes.

Moreover, the report also includes the VHDL code of the router in Appendix A, the test bench code in Appendix B, the Simulation waveform output for the implemented code using ModelSim in Appendix C, and the gate count, timing summary, and device utilization summary can be found in Appendix D, these all can be found at the end of the report, a testing strategy for the router can be found as well within the report.

Table of Contents

List of Figures

Extra Figures can be found in Appendix C and D to represent the wave form and synthesis for each module implemented.

1. INTRODUCTION

In this project using the methodologies we studied in Electronic Design Automation (CSE 312) course we implement and test a simple router using vhdl hardware description language. This project includes the implementation of the router with its testbench. We assumed that the router we are implementing has 4 data inputs and outputs instead of n. In the beginning we should explain why we need NOC routers. A Network-on-Chip is a new paradigm of communication network into System-on-Chip (SoC), It overcomes the problems of traditional bus-based SoC and meets the communication requirement of the next SoC. A router is one of the most important communication backbone in NoC. Using buses as communication strategy does not give any flexibility for the needs of communication. Secondly using shared buses does not scale very well as the number of resources increases in number. These drawbacks have been overcome in Network-on-chip by implementing a communication network of routers, unlike computer networks, NoC has shorter communication delay.

There are many types of routers:

1. *Wired and wireless routers*

These routers are mostly used in homes and small offices. Wired routers share data over cables and create wired local area networks, while wireless routers use antennas to share data and create wireless local area networks

2. *Edge routers*

These routers are either wired or wireless; they are used mainly to distribute data packets between one or more networks but not within a network, so they keep your network communicating smoothly with other networks.

3. *Core routers*

These routers are also either wired or wireless and are used to distribute data packets but within the same network, they are designed to do the heavy lifting of data transfer.

4. *Virtual routers*

Unlike physical routers, virtual routers are pieces of software that allow computers and servers to operate like routers. They'll share data packets just as physical routers do. They can also help get remote offices up and running on your network more quickly.

Building Blocks

The router is built of two main blocks, the datapath that contains several modules implemented by combinational logic. The second main block is the controller and it is implemented by a finite state machine (FSM), it controls data flow inside the router and applies a round-robin scheduler at the output ports. The datapath contains:

1. *Input Buffer*

Our implementation of an input buffer is an 8-bit register. The main functionality of this module is to store the packet once it arrives.

2. Switch Fabric

The switch fabric responsibility is to provide full connectivity between the inputs and outputs of the switch with reasonable hardware and delay. Implemented by an 8-bit demux, the controller module reads the first 3 bits in the incoming packet, then sets the connection to the output queue according to destination address.

3. Output Queue

Output queues are implemented as First-in/First-out (FIFO) buffers. The main building blocks of a buffer are: memory element, control register, and controller FIFO buffers can be implemented in many ways but in our project we implemented a block RAM as a memory element. The control register is implemented by a gray counter and a gray to binary converter to write/read to/from any memory address.

And finally the controller is implemented in module 6 as a FIFO controller.

4. Output Buffer

An output buffer is implemented the same way as an input buffer.

How to Build a Simple Chip?

1. Specification:

The client determines what the chip is supposed to do, how quickly it can work, etc. It is not a design and is considered as a set of requirements.

2. Behavior:

The behavioral explanation seems to be far more accurate than the previous phase, since it is written as an executable program not in simple English like the specification.

3. Register-transfer:

The time behavior of the system is completely defined, the permitted input and output values are known on any clock cycle, however the logic isn't really defined as gates. The framework is defined as Boolean operations which are placed in the abstract memory elements. From the Boolean logic functions, only the briefest delay estimates can be done.

4. Logic:

In this phase we can view the system design as Boolean logic gates, flip-flops, and latches. Even though know we know more about the system's structure, we are still unable to make precise delay measurements.

5. Circuit:

In the phase the implementation of the system is done in the form of transistors.

6. Layout:

This is the last design that is to be fabricated.

CAD Tools

Automating the design process, using computer-aided design (CAD) software that automate parts of the design process, is the only practical approach to design chips considering performance and design time constraints. When executed correctly, using software to automate design certainly helps us solve three issues: working with several layers of abstraction is

simpler if you are not immersed in the specifics of a specific design step; computer programs will do a better job of evaluating cost trade-offs as they're more analytical; and finally computers work faster than us when a task that is well-defined is given to it.

2. DESIGN FLOW

FPGA design flow:

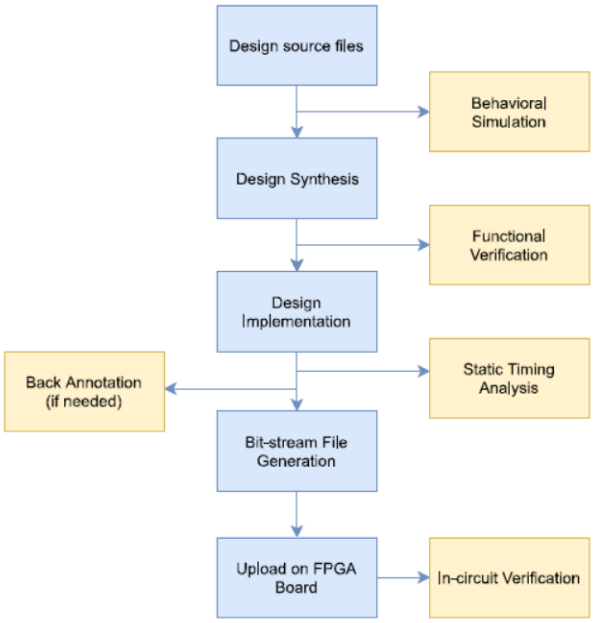


Figure 1 Design Flow

In this section, we'll provide the FPGA design flow steps

1. *Design Specification*

First, we determine the functionality and the constraints like the performance, power, area, latency and throughput, then we determine our architectural design base on this specification which will be divided into minor modules to be connected together to make the system level module, this can facilitate to us in tracing the errors or some modules can be reused, also in this step we determine and choose the FPGA board that our design will be loaded on it.

2. *Design Entry*

It can be made by different ways, as it can be by schematic or by hardware description language or by combination of both, the decision is based on the designer and the design as if the design is small, we can choose the schematic and we choose the hardware description language if the design is more complex, also the schematic based gets us closer to the hardware implementation details and in HDL, we think more in an algorithmic way , here we choose to represent our design by VHDL, we make the submodules by the VHDL code in the Modelsim, we also can write the constraints especially the clock in a XDC file.

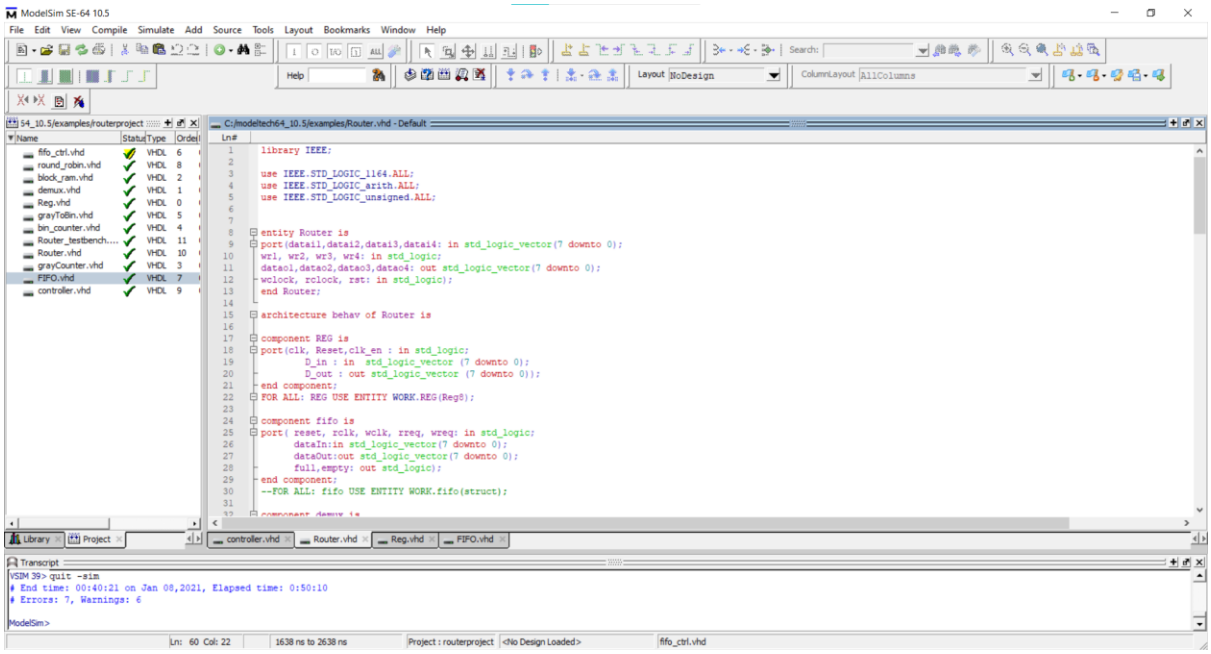


Figure 2 Example of VHDL code in Modelsim

3. Behavioral Simulation

We make the behavioral simulation by viewing the wave form for the inputs and the outputs either by forcing the values of the inputs manually or by making the testbench and the testbench is preferable for larger designs, this step make us sure that the code is free of bugs and to ensure the behavior of the code and its functionality behave as expected.as we made for each submodule.

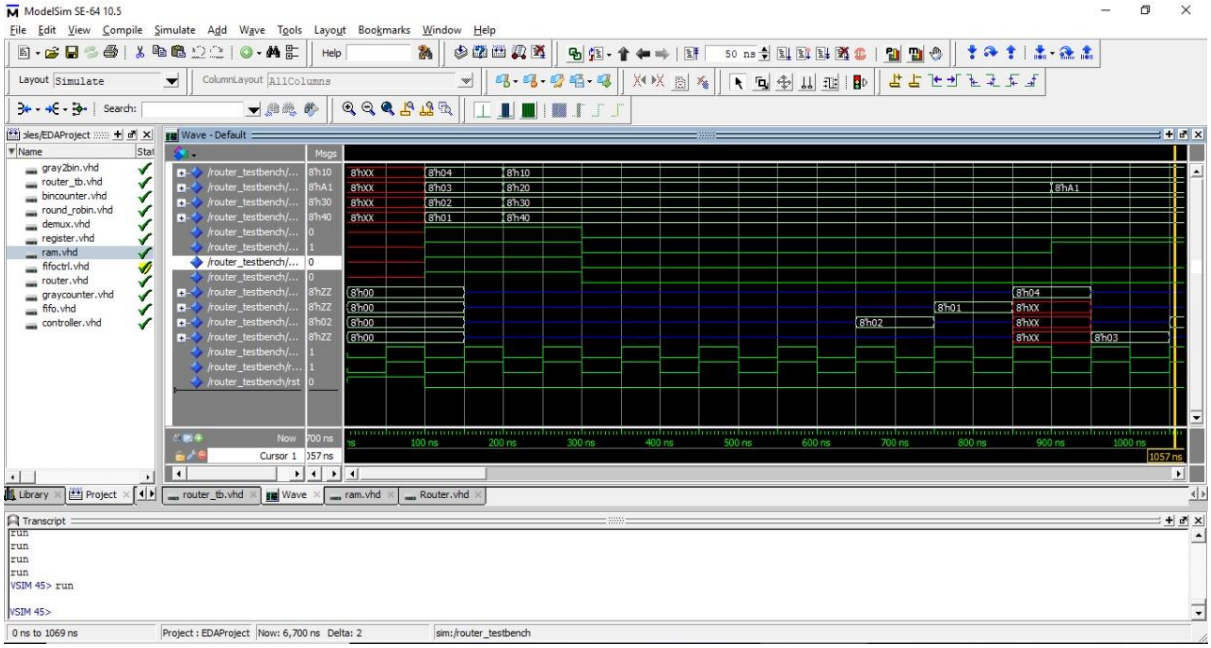


Figure 3 Example of Wave Form in Modelsim

4. Design Synthesis

We put our VHDL codes and the design constraints to the synthesis tool (Xilinx Synthesis Technology (XST), Precision from Mentor Graphics Inc. ,Synplify and Synplify Pro from Synplicity Inc.)to translate the RTL code to a complete circuit eith logic elements as we get an optimized gate level netlist, as we get a netlist of all the synthesizable modules to check that all of them are synthesizable, we get also a synthesis report to be checked in case of improvement or any violation in the design constraints , we also can view a schematic of the synthesis result.

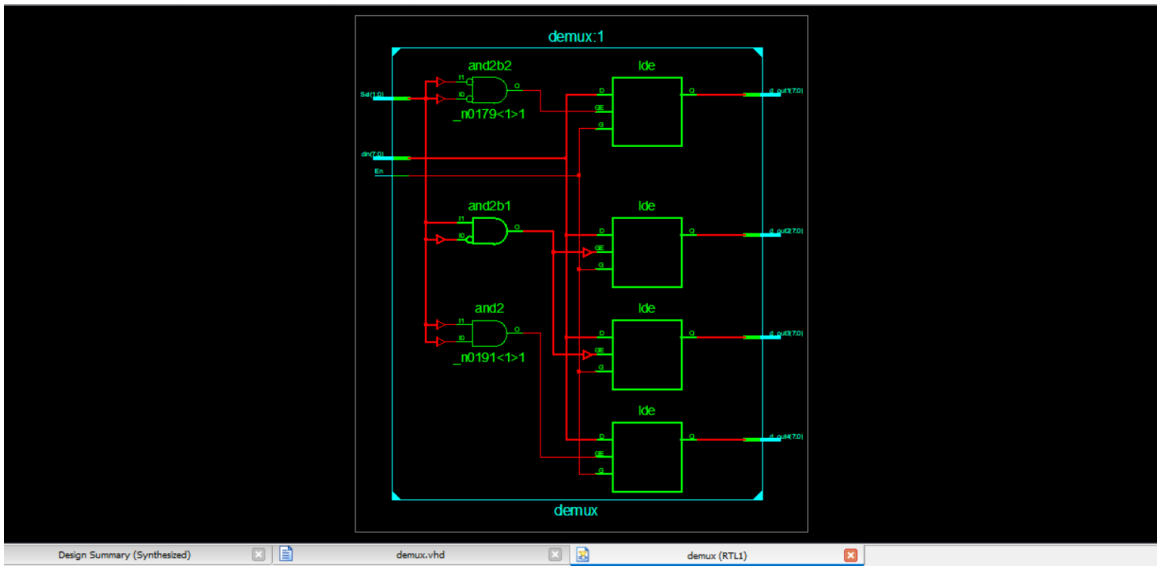


Figure 4 Example in Xilinx ISE

5. Design Implementation

This includes of 3 steps

a) Translate

We translate the generated netlist files and the constraints files to one file and by using NGD program, we save the file as native generic database and make port assignment to real physical (ports, switches, buttons) of the FPGA, and after the translate, we can make the functional simulation by using the SIMPRIM library to check the circuit's functionality is as expected or not.

b) Map

We divide the circuit into sub-blocks and map them to the real FPGA blocks and we can use the map program and we can generate native circuit description file for the representation of the mapped design to the actual FPGA components.

c) Place and Route

We place the sub blocks from the previous step into the logic blocks (designated FPGA blocks) and the connect the blocks, while doing this process, we should take into consideration the constraints as if we place a block close to the I/O pins so this will save time and we can use the PAR program for this step.

6. *Static Timing Analysis*

This step is made after the design implementation to check the timing constraints, and check the signal propagation delays path and check the delay of the longest path(critical) to check the set-up violation and we also check if there's a hold time violation.

7. *Back-annotation*

We check the reports after the implementation for any timing or physical violations, we change either the design or the constrains, we can make iterations of re-synthesizing and re-implementing the design till we meet all we need.

8. *Device Programming*

In order the FPGA platform understand the and accept the design, we need to convert the implemented design to Bitstream using BITGEN program and its stored on FPGA memory card then we connect the device to board by the USB, as the bitstream file programs the board directly

9. *In-circuit verification*

In this step, we want to ensure that the right circuit implementation has occurred after the bitstream file is uploaded on the FPGA, this's can be done by the hardware debugging Ips in the FPGA.

In this project we first used ModelSim to write and compile our VHDL codes and then get the wave forms, then to check if the code is synthesizable and get the gate count we used Xilinx ISE. More about the modules' implementation can be found in section 4 and the synthesis results in appendix D.

3. LITERATURE REVIEW

Multiprocessor's architecture depends on concurrency and synchronization in both hardware and software to enhance the performance of the system and the design productivity. Some of the key problems in the deep sub-micron technologies characterized by the gate lengths in the 60-90 nm range, with the rapidly approaching billion transistors age, would arise from non-scalable wire arrays, signals integrity errors and un-synchronized communications. The use of network chip (NOC) architecture can overcome these issues. Traditional system designs are based on trees and critical paths which contribute to an increased power consumption so SoC (System on Chip) which is synchronized by global clock signal is not power efficient. Asynchronous designs don't suffer from this issues but they are complex to design comparing to synchronous designs so researches found a solution by combining synchronous and asynchronous designs creating a strategy called GALS (globally asynchronous and locally synchronous) design. One GALS' solution is NOC, which can improve productivity of the design by supporting modularity and reuse of complex cores. The article referenced represents more on how the use of NOC can solve many problems and that it also enables higher level of abstraction in the architectural modeling of future systems if needed. [1]

The book referenced provides the historical perspective of routers and switch architectures. Although it offers a radical exploration of the NOC design space, several emerging technology trends or techniques indicate numerous new topics for future research within the NOC field. Exploring the NOC structure to style a quick communication structure for synchronization signals. Providing NOC features for transactional memory programming paradigm.it also describes the NOC logic implementation and shows how buffers are shared among VCs showing the efficient design of NOC buffers based on hierarchal bit line that has a low power overhead and considered as flexible according to the traffic that causes a change in buffers structure.[2]

The research referenced represents the design and evaluation of a highly configurable NOC employed in AcENocs (Accelerated Emulation Platform for NOCs), a versatile and cycle accurate field programmable gate array (FPGA) emulation platform for validating NOC architectures. It also represents the design of the router using Verilog hardware description language (VHDL), and how the packet NOC is constructed by connecting the routers in either 2D-tours or 2D-mech topology and how it is integrated in the AcENocs platform and prototyped on Xilinx Virtex-5 FPGA.[3]

4. DESIGN IMPLEMENTATION

The following figures represents the block diagram of the 4-port router that was implemented in this project.

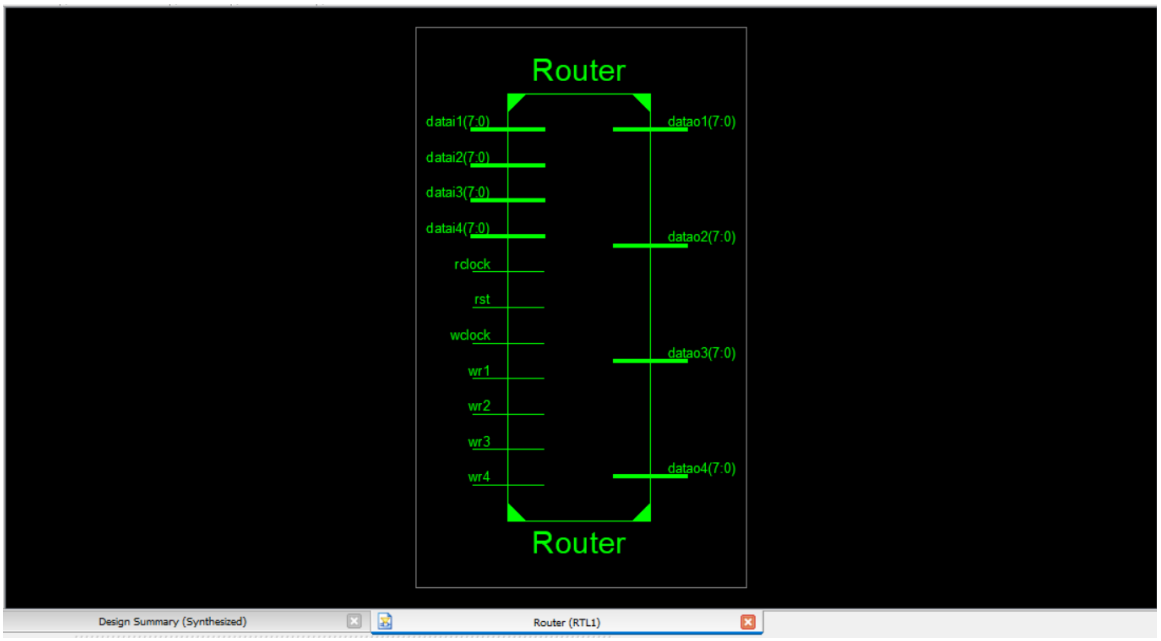


Figure 5 Router Block

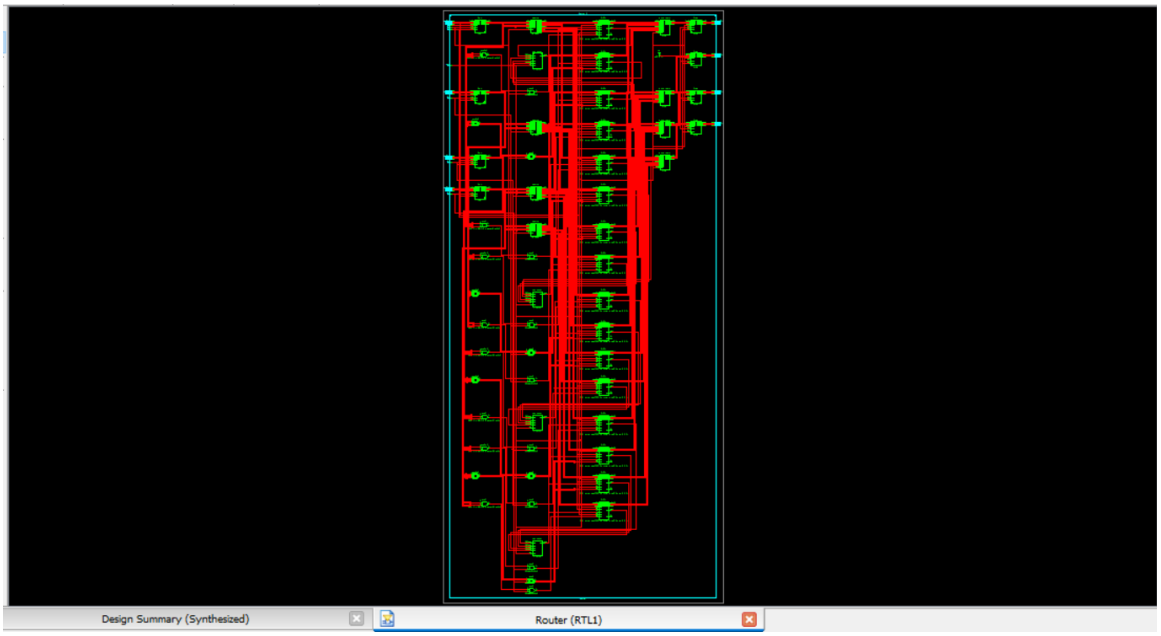


Figure 6 Router Block Content

Modules and Functions

This section contains all the modules that were used in the project along with each of their functions.

	Module	Function
1	8-bit Register (M-ROU-01)	Used in input buffer to store packet when it arrives. Used in output buffer to deliver the packet to the output port after picking it up from the output queue.
2	8-bit Demux (M-ROU-02)	Makes up the switch fabric and provide full connectivity between the switch's input and output, it also takes the packet and propagate it to the output queue with its specific address with the help of the controller.
3	Block RAM (M-ROU-03)	Used to store array of data (data in 8 bits), we can also read and write from it since it is synchronized by 2 clocks.
4	Gray Counter (M-ROU-04)	Counts up and outputs it in gray code.
5	Gray to Binary Converter (M-ROU-05)	Takes in 4 bits of data represented in Gray and converts it to 4 bits of data represented in Binary.
6	FIFO Controller (M-ROU-06)	It reads the packet header and configure the switch fabric, synchronize the incoming packets with the internal modules of the router, control flow of data inside the router and apply the round robin scheduler. It receives read and write requests and checks the validity of the read and write operations.
7	FIFO (M-ROU-07)	Hold the data in RAM using controller and makes them work with the First In First Out sequence.
8	Round Robin Scheduler (M-ROU-08)	Allow data from the queue to pass one at a time every clock cycle, serving all ports.
9	4-port Router (M-ROU-09)	Takes an input from the input port, packet that has a fixed length, and use the first 2 bits that represent the output port address and sends the packet to that specific output port.

Module Implementation

The code can be found in Appendix A, the gate count, timing summary, and device utilization summary can be found in Appendix D.

1. 8-bit Register

This module was implemented so that if the clock (Clock) has a positive edge, the enable (Clock_En) is 1, and the reset (Reset) is 0, the data will be transferred from input port (data_in) to output port (data_out). Both input and output ports have 8 bits, hence why it's called an 8-bit Register. However, if the (Clock_En) is 0 the output port will stay the same, and if the

(Reset) is 1 then the value in the output port is reset to 0 (in 8 bits). Since the (Reset) is asynchronous, both it, the clock and the clock_en are placed in the sensitivity list of the process. This module has only one process.

2. 8-bit Demux

To implement the 1 to 4 demultiplexer, we created a process with the data in (d_in), enable (En), and selector (Sel) in the sensitivity list of this process. Whenever any of them changes first the process checks if the (En) is 1 then it enters the case statements and search for the specific (Sel) it has and hence the (d_in) is routed to the specific (d_out) depending on whether the (Sel) is 00,01,10, or 11, there is a fifth case, others, which is null but added to handle a Sel value if it's not in the 4 normal cases. Data in and data out are represented in 8 bits. If the (En) is 0 then the output ports will not change their values.

3. Block RAM

To implement the dual ram module, we created 2 process with 2 Clocks, one for reading and the other for writing. The first process, has (CLKA) in its sensitivity list and is responsible to write 8 bits of data in the 2D array, first, it checks if (CLKA) is at a rising edge, then if the write enable (WEA) is 1, the address in (ADDRA) is converted to integer and is used as the index of the 2D array (ram) and the data entered (d_in) which is in 8-bits is written in that specific index in the array.

The second process has the (CLKB) in its sensitivity list, this process is responsible for reading from the 2D array, first if (CLKB) is at a rising edge then it checks if the read enable (REA) is 1 the address (ADDRB) is taken converted to integer, this integer is used as the index of the 2D array and the value in that index is assigned to the data out (d_out) to be read. However, if the (REA) is 0 then high impedance is output.

4. Gray Counter

To implement the gray counter module, we first implemented a binary counter module. The binary counter module has one process with the (clk) in its sensitivity list, if a rising edge occur then it checks if the reset is one then the counter, unsigned 4-bit signal, is reset to 0000; however, if reset is not 1 then it checks if enable is one, if it is then the counter is incremented and finally the number in the counter is assigned to the count_out which is the output represented in 4 bits.

Later to implement the gray counter we mapped the bin_counter component in the gray counter and created one process where b_counter, a 4 bit signal, is in the sensitivity list, then the most significant bit (MSB) of the b_counter is assigned to the MSB of the count_out, which represent a gray number, then we enter a for loop to get the 3 bits left of the gray code by using XOR between b_counter(i+1) and b_counter (i), binary 4 bit code, to get the count_out(1), gray 4 bit code.

5. Gray to Binary Converter

We implemented the Gray to binary converter by creating a process that has gray_in, a 4-bit gray input, in its sensitivity list to go through the process every time the input changes. Then we assigned the MSB of the gray_in to the MSB of the bin_out, a 4-bit binary output, then to get the bin_output(2) we used XOR between gray_in(3) and gray_in(2), then to get the bin_output(1) we used XOR between gray_in(3), gray_in(2), and gray_in(1), and finally to get the bin_output(0) we used XOR between gray_in(3), gray_in(2), gray_in(1), and gray_in(0). This method is similar to the method we used in the gray counter except that in gray counter it was from binary to gray and in this module we are converting from gray to binary.

6. FIFO Controller

To implement the FIFO controller first we wrote in the entity the inputs, like the read and write clocks, the read and write requests signals, and the reset, then we wrote the outputs, like read valid and write valid indications, the read and write address which are 4 bits, and finally the empty and full flags. In the architecture, we first added some read and write signals, these signals were then used to port map the components Gray to binary converter and the gray counter to gray_bin_rd, gray_bin_wr, gray_count_rd, and gray_count_wr. We also created a shared variable of 8 bits to represent the FIFO, and an s_write_reset as well as a s_read_reset and set them to one initially. We also assigned s_rd_ptr and s_wr_ptr signals rd_ptr and wr_ptr respectively.

We created one process with read request, write request, reset, read and write clocks, s_rd_ptr_gray, and s_wr_ptr_gray in its sensitivity list. First, we check if the reset is 1 then we reset everything to their default values which are s_read_reset and s_write_reset are 1, full is 0, empty is 1, write valid and read valid are zero, and the fifo has 00000000 assigned to it. If the reset is not 1 then it checks if the read and write pointers do not have the highest address of 0111 the read and write reset are 0 if they do have the highest address, then the read and write reset becomes 1.

After that, if the rising edge of the read clock occurred then we check if the read request is 1 then if the FIFO is not empty then the read valid gets a 1 and the place of that address in the FIFO gets a 0 otherwise the read valid indication is set to 0. On the other hand, if the rising edge of the write clock occurred then we check if the write request is 1 then if the FIFO is not full then the write valid gets a 1 and the place of that address in the FIFO gets a 1 otherwise the write valid indication is set to 0.

Finally, if the FIFO is empty then the empty flag is set to 1 and the full flag is set to 0, if the FIFO is full then the full flag is set to 1 and the empty flag is set to 0, if neither empty nor full then both flags get a zero.

7. FIFO

To implement this module we had to implement the FIFO controller and the RAM modules first then in the entity we added reset, rclk, wclk, rreq, wreq, and dataIn (8 bits) as inputs, and dataOut(8 bits) , full and empty as outputs, then we added the FIFO controller and Block ram components with the ports in the architecture. Then we created ADDRA_ptr and ADDRB_ptr which are both signals in 4 bits; in addition, we created write_valid_signal and read_valid_signal then finally we port mapped the controller and the ram using the signals that we have created plus the inputs and outputs that we already had in the FIFO entity.

8. Round Robin Scheduler

This module takes in 4 data inputs of 8 bits each, and a clock, it then outputs only one data output of 8 bits. To implement the round robin module, we first created 4 states from s1 to s4 then declared signals NS and CS of state_type and set CS signal initially to s1. After that, we created two processes the first process with clock in its sensitivity list to assign the data in the NS to the CS when a rising edge occurs, the CS is assigned to itself when there is no rising edge, the second process has the CS in it sensitivity list to go through the process whenever the CS changes, this process has cases to check when CS is s1 then assigns s2 to NS and assigns data input 1 to the data out to be output, and when CS is s2 then assigns s3 to NS and assigns data input 2 to the data out to be output and so on till when CS is s4 then s1 gets assigned to NS to complete the circle and outputs the data from input 4.

9. 4-port Router

To implement the router first we implemented a controller module which could have been directly written in the Router module but for simplicity we wrote it on its own. The controller module has rdclk, Fifo_1_empty, Fifo_2_empty, Fifo_3_empty, and Fifo_4_empty as inputs and rd_req as output. In the architecture created 4 states from fifo1 to fifo4; in addition to 2 signals state current_fifo and next_fifo. It has 2 process the first checks id rising edge occurred and assigns next_fifo to current_fifo. The second process has current_fifo and Fifo_1_empty to Fifo_4_empty in its sensitivity list then when current_fifo is fifo1 then next_fifo is fifo2 and if Fifo_1_empty is 0 then the read request is set to 1 otherwise read request is set to zero, similarly for cases when fifo2 and fifo3 are current_fifo, the only difference in case when fifo4 is that its next state is fifo1 to connect the loop.

After implementing the controller, we started implementing the Router itself which is made out of input and output buffers which are registers, a switch fabric made of demux, and an output queue (FIFO), along with a round robin scheduler. First in the entity we have datai1, datai2, datai3, and datai4 as inputs of 8 bits, in addition to the previous inputs we also have wr1, wr2, wr3, wr4, wclock, rclock, rst. For the outputs for this router we have datao1, datao2, datao3, and datao4.

We know that the input arrives at the register in the input buffer then when rising edge occurs it is output from the buffer and input into the demux which checks its first 2 bits to figure out the address of the output port required in order to output this packet to the FIFO that is connected from that demux and to the required output port, in order to be output depending on the round robin.

Knowing this, we added the components and ports of the register, fifo, demux, round robin, and controller modules. Then we added some arrays and signals that we will use later on while porting. The process has data_in, wrreq_arr, and wclock in it sensitivity list followed by 2 for loops to set the value of that array to either 1 or 0. Then we port mapped 4 registers from IB1 to IB4, port mapped 4 demux from DeMux1 to DeMux4, then port mapped 4 round robin from RR1 to RR4, later we port mapped 4 controllers from CTRL1 to CTRL4, and to port map the the fifo we used 2 for loops one of the outer loop and the other for the inner loop. Then, we mapped 4 more registers for OB1 to OB4. Finally, we assigned the OBOutput(0) to datao1 and OBOutput(1) to datao2 and so on till datao4.

5. SCHEDULAR DESIGN and FSM IMPLEMENTATION

FSM (Finite State Machine) is a model of computation based on hypothetical machine made from one or more states where only one state is active at the same time, therefore the machine must transition from one state to another in order to perform different actions. It is commonly used to organize and represent an execution flow. [4]

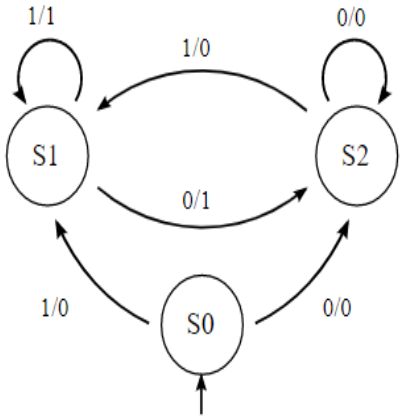


Figure 7 FSM

FSM is represented by this figure where the nodes are the states and edges are the transitions and each edge has a label that informs when the transition should happen, like(0/1) where a transition between state ’1’ and ‘2’ should happen when input ‘0’ is entered given ‘1’ as output.

In the scheduler module we use the Moore FSM implementation type as the output depends on the current state and transitions between states happens every clock cycle.

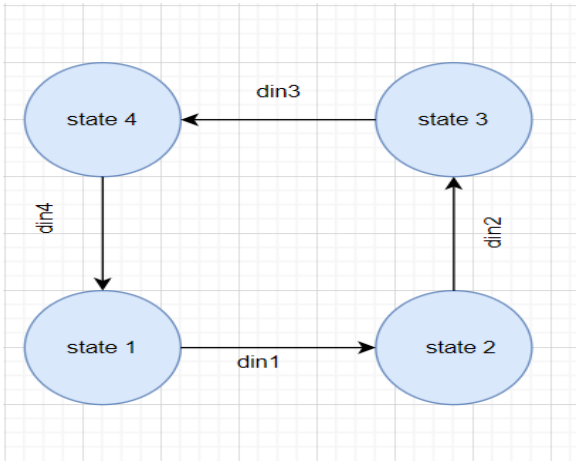


Figure 8 FSM of Scheduler

Comparison

Single process	2-process	3-process
It is a fully synchronous process. Advantages: easier to debug and easier to meet time requirements during place and route.	FSM code is divided into two processes, one synchronous and the other combinatorial. There are at least three ways that two process state machine can be implemented. Advantages: looks better. Disadvantages: more complicated.	Code is divided into three processes where each signal have its own process.

Timing analysis

Using Xilinx ISE, to get an accurate timing analysis for the scheduler circuit which enable us to calculate maximum frequency.

```
========================================================================
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  |
-----------------------------------+-------------------------+-------+
clock                              | 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)

========================================================================
```

Figure 9 Time Analysis of Scheduler

Synthesis analysis

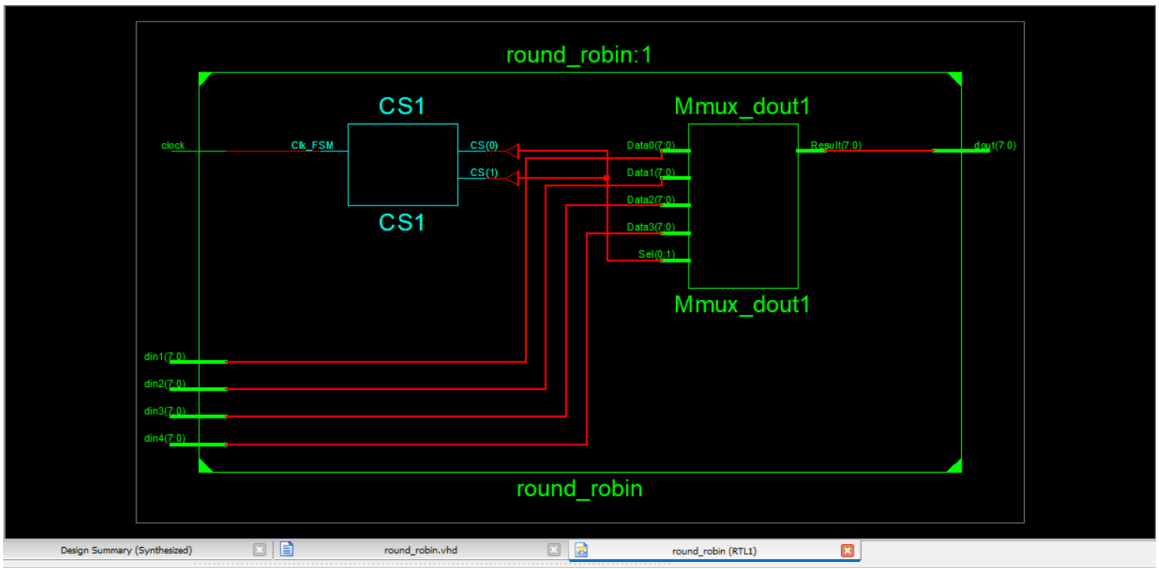


Figure 10 Round Robin RTL

```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:              2  out of   4800     0%
 Number of Slice LUTs:                  10  out of   2400     0%
    Number used as Logic:               10  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    10
   Number with an unused Flip Flop:      8  out of     10    80%
   Number with an unused LUT:            0  out of     10     0%
   Number of fully used LUT-FF pairs:    2  out of     10    20%
   Number of unique control sets:        1

IO Utilization:
 Number of IOs:                         41
 Number of bonded IOBs:                 41  out of    102    40%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:              1  out of     16     6%
```

Figure 11 Synthesis Report

6. TEST and SIMULATION RESULTS

In this section. We'll discuss the testing to the router and the testing strategies we've used. First, we test each module separately by using the functional simulation that've discussed in the section of the design flow, as we make for each module a simulation by forcing the inputs with specific values and determining the duty of clock and the period time of the clock then we observe the output wave form of each module to observe the output signal with respect to the forced input signals to check that the module behave functionally correct, we've also make simulation for the router module and we've provided the screenshots of the waveform of each module in Appendix C.

After we connect our modules together, we made the second test to the router which is the test bench and we've provided the code of the test bench in Appendix B.

Test cases:

1. First, we test when the reset is one and wait for 100ns
2. We put data inputs on the four inputs ports plus making the reset equals to zero and making the write enables equals to one and then wait for 100ns
3. We change the data in the four input ports and wait for 100ns
4. We change all the write enables to zero and wait for 100 ns
5. We check here the first test case by assert statement at the output port at datao2
6. We continue to check the first test case by waiting 100ns and check the output port at datao1

7. We then wait for 100ns and check the output port at datao4 to check the first test case and check the output port at datao1 to check the second test case
8. We wait for 100 ns the output at datao3 (check for the first test case) and check the output at datao1(check for the second test case)
9. We wait for 100ns to check the output at datao1 (check for the second test case)
10. We wait again for 100 ns to check the output at datao1(check for the second test case)
11. Now we begin the test case 3, we put data at input port datai2 and the write enable 2 equals to one and wait for 100*10 ns and change the input at datai2 and wait for 100 ns and then make the right enable 2 equals to then wait for 100*30 ns then we check the output at datao2 that's supposed to be equals to the data we entered at datai2 at the beginning of test case 2.
12. In test case four, we put data at input datai4 and make the write enable 4 equals to one then we wait for 100ns*2 and make the reset (rst) equals to one and the write enable 4 equals to zero then we wait for 100ns and put data at input darai4 and make the rst equals to zero and the write enable 4 equals to one then we wait for 100*2 ns to check output datao4 that's supposed to be equal to the last value we put at datai4

7. CONCLUSION

In conclusion, we achieved the goal of this project were we managed to build the NOC router, which is a simple router but at the beginning we faced some challenges such as understanding the concept behind each module, understanding the behavior of the router, the details of implementations and how these modules can be mapped together to build the router. Facing this challenges, we ended up implementing the router and getting the required results which are represented in Appendix D, also to get this results we made a test strategies represented in outline 6.

For the FSM of the scheduler module, we found that the best implementation is style is 2 process, it makes the code easier to read and reusable, and for the FSM of the FIFO we used one process implementation style.

For future work, we could find better methodologies for the output queues, buffers and switch fabric of the NOC router. We found that the method we used which offers one large queue for each output to do work for both queue and buffer using FIFO helped us facing the problem of dropping out some packets when the queue is full, and by doing more researches we could find better method as there is always development in the NOC architecture.

As for the challenges, we weren't able to decide when will the router work in the same module so a different module was made which is the controller to control the router and this allowed encapsulation of the modules as well so it was easier to identify which module wasn't working properly when running the router.

8. TASK DISTRIBUTION LIST

The project workload is distributed equally among all team members. We worked together on multiple online meetings where every member took their turn in writing some code and sharing ideas, then we started working on the report to document our project.

REFERENCES

1. *Survey of network on chip NOC architectures and contributions. Journal of Engineering, Computing and Architecture.*
2. *Medhi, D. and Ramasamy, K., 2018. IP routing and distance vector protocol family. In Network Routing (Second Edition) (pp. 160-182). Morgan Kaufmann.*
3. *Lotlikar, S.S., 2011. Design, Implementation and Evaluation of a Configurable NoC for AcENocs FPGA Accelerated Emulation Platform (Doctoral dissertation, Texas A & M University).*
4. *https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867*

APPENDIX A (VHDL Code)

VHDL source code for modules from 1 to 9 plus binary counter and router controller.

1. 8-bit Register (M-ROU-01)

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

entity Reg is
  port(clk, Reset,clk_en : in std_logic;
       D_in : in  std_logic_vector (7 downto 0);
       D_out : out std_logic_vector (7 downto 0));
end entity Reg;

architecture Reg8 of Reg is
begin
p1: process (clk_en,clk,Reset)
     begin
       if (Reset='1') then
         D_out <= "00000000";
       elsif (rising_edge(clk) AND clk_en= '1')then
          D_out <= D_in;
       end if;
   end process p1;
end Reg8;
```

2. 8-bit Demux (M-ROU-02)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity demux is
 port(

 din : in STD_LOGIC_VECTOR (7 downto 0);
 Sel: in STD_LOGIC_VECTOR (1 downto 0);
 En: in STD_LOGIC;
 d_out1, d_out2, d_out3, d_out4 : out STD_LOGIC_VECTOR (7 downto 0)
 );
end demux;

architecture demux8 of demux is
begin

p1: process (En,Sel,din) is
begin
if(En = '1') then
case sel is
when "00"=> d_out1<=din;
when "01"=> d_out2<=din;
when "10"=> d_out3<=din;
when"11"=> d_out4<=din;
when others=> null;
end case;

end if;
end process p1;
end demux8;
```

3. *Block RAM (M-ROU-03)*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity block_ram 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	: in std_logic := '1';
	  REA  : in std_logic := '1';
	  CLKA		: in std_logic;
		CLKB	: in std_logic;
		d_out	: out std_logic_vector(7 downto 0)
	);

end block_ram;

architecture ramdual of block_ram is

	-- Array ram
	subtype wordd is std_logic_vector(7 downto 0);
	type mem is array(7 downto 0) of wordd;

	signal ram : mem;

begin

	process(CLKA)
	begin
		if(rising_edge(CLKA)) then
```

```
                    if(WEA = '1') then
                        ram(to_integer(unsigned(ADDRA))) <= d_in;
                    end if;
                end if;
        end process;

        process(CLKB)
        begin
                if(rising_edge(CLKB)) then
            if(REA ='1') then
                    d_out <= ram(to_integer(unsigned(ADDRB)));
elsif(REA ='0') then
                    d_out<="ZZZZZZZZ";
            end if;
                end if;
        end process;

end architecture ramdual;
```

4. *Gray Counter (M-ROU-04)*

```
LIBRARY ieee;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
ENTITY grayCounter IS
port(clk,en,reset:in std_logic;
count_out:out  std_logic_vector(3 downto 0));
end;
architecture behav of grayCounter is
component bin_counter is
port(clk,en,reset:in std_logic;
count_out:out unsigned(3 downto 0));
end component;
FOR ALL: bin_counter USE ENTITY WORK.bin_counter(behav);
SIGNAL b_counter:unsigned(3 DOWNTO 0);
begin
b_count: bin_counter port map(clk=>clk,en=>en,reset=>reset,count_out=>b_counter);
p1: process(b_counter) is
begin
count_out(3)<=b_counter(3);
for i in 2 downto 0 loop
count_out(i)<=b_counter (i+1) XOR b_counter (i);
end loop;
end process p1;
end architecture;
```

Binary Counter

```
LIBRARY ieee;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY bin_counter IS
port(clk,en,reset:in std_logic;
count_out:out unsigned(3 downto 0));
end;

architecture behav of bin_counter is
SIGNAL counter:unsigned (3 DOWNTO 0);
begin
p1:process (clk) is
begin
IF rising_edge (clk) THEN
if reset='1' then counter<="0000";
elsif en='1'
then counter<=counter+1;
end if;
end if;
end process p1;
count_out<=counter;
end architecture behav;
```

5. *Gray to Binary Converter (M-ROU-05)*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity grayToBin is
port(
gray_in:in std_logic_vector(3 downto 0);
bin_out:out std_logic_vector(3 downto 0));
end;
architecture behav of grayToBin is
begin
p1:process(gray_in) is
begin
bin_out(3)<=gray_in(3);
bin_out(2)<=gray_in(3) XOR gray_in(2) ;
bin_out(1)<=gray_in(3) XOR gray_in(2) XOR gray_in(1);
bin_out(0)<=gray_in(3) XOR gray_in(2) XOR gray_in(1) XOR gray_in(0);
end process p1;
end architecture behav;
```

6. FIFO Controller (M-ROU-06)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity fifo_ctrl is
port(
rdclk, wrclk, reset: in std_logic;
rreq, wreq: in std_logic;
write_valid, read_valid : out std_logic;
wr_ptr, rd_ptr : out std_logic_vector(3 downto 0);
empty, full : out std_logic
);
end entity fifo_ctrl;

architecture fifo_arch of fifo_ctrl is
signal s_wr_ptr_gray : std_logic_vector(3 downto 0) ;
signal s_rd_ptr_gray : std_logic_vector(3 downto 0) ;
signal s_wr_ptr : std_logic_vector(3 downto 0) ;
signal s_rd_ptr : std_logic_vector(3 downto 0) ;
shared variable fifo: std_logic_vector(7 downto 0):="00000000" ;
signal s_write_reset : std_logic:='1';
signal s_read_reset : std_logic:='1';

component grayToBin is
port(gray_in:in std_logic_vector(3 downto 0);
bin_out:out std_logic_vector(3 downto 0));
end component;
FOR ALL: grayToBin USE ENTITY WORK.grayToBin(behav);
component grayCounter is
port(clk,en,reset:in std_logic;
count_out:out  std_logic_vector(3 downto 0));
end component;
```

```
FOR ALL: grayCounter USE ENTITY WORK.grayCounter(behav);

begin
gray_bin_rd: grayToBin port map(s_rd_ptr_gray, s_rd_ptr);
gray_bin_wr: grayToBin port map(s_wr_ptr_gray, s_wr_ptr);
gray_count_rd: grayCounter port map(rdclk, rreq, s_read_reset, s_rd_ptr_gray);
gray_count_wr: grayCounter port map(wrclk, wreq, s_write_reset, s_wr_ptr_gray);

rd_ptr<=s_rd_ptr;
wr_ptr<=s_wr_ptr;

p_request: process (rreq,wreq,reset,rdclk,wrclk,s_rd_ptr_gray,s_wr_ptr_gray) is
begin
if reset='1' then
 s_read_reset<='1';
 s_write_reset<='1';
 full<='0';
 empty<='1';
 write_valid<='0';
 read_valid<='0';
 fifo:="00000000";
--end if;
--if reset='0' then
else
if s_wr_ptr/="0111" then s_write_reset<='0';
 else s_write_reset<='1';
end if;

if s_rd_ptr/="0111" then s_read_reset<='0';
 else s_read_reset<='1';
end if;

if rising_edge (rdclk) then
 if rreq='1' then
```

```
 if fifo/="00000000" then
 read_valid<='1';
 fifo(to_integer (unsigned(s_rd_ptr))):='0';
 else read_valid<='0';
 end if;
 end if;
end if;

if rising_edge (wrclk) then
 if wreq='1' then
 if fifo/="11111111" then
 write_valid<='1';
 fifo(to_integer (unsigned(s_wr_ptr))):='1';
 else write_valid<='0';
 end if;
 end if;
end if;

if fifo="00000000" then
 empty<='1';
 full<='0';

elsif fifo="11111111" then
 full<='1';
 empty<='0';
else
empty<='0';
full<='0';
end if;

end if;
end process p_request;
end architecture fifo_arch;
```

7. FIFO (M-ROU-07)

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity FIFO is
port( reset, rclk, wclk, rreq, wreq: in std_logic;
      dataIn:in std_logic_vector(7 downto 0);
      dataOut:out std_logic_vector(7 downto 0);
      full,empty: out std_logic);
end FIFO;

Architecture struct of FIFO is

component block_ram 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: in std_logic;
     REA: in std_logic;
     CLKA: in std_logic;
     CLKB: in std_logic;
     d_out: out std_logic_vector(7 downto 0));
end component;
FOR ALL: block_ram USE ENTITY WORK.block_ram(ramdual);
component fifo_ctrl is
port(rdclk, wrclk, reset: in std_logic;
     rreq, wreq: in std_logic;
     write_valid, read_valid: out std_logic;
     wr_ptr, rd_ptr: out std_logic_vector(3 downto 0);
     empty, full: out std_logic);
```

```
end component;

FOR ALL: fifo_ctrl USE ENTITY WORK.fifo_ctrl(fifo_arch);

signal ADDRA_ptr,ADDRB_ptr: std_logic_vector(3 downto 0);

signal write_valid_signal, read_valid_signal: std_logic;


BEGIN


controller: fifo_ctrl

port map( rdclk=> rclk, wrclk=>wclk, reset=> reset, rreq=>rreq, wreq=>wreq, write_valid=>write_valid_signal, read_valid=>read_valid_signal, wr_ptr=>ADDRA_ptr, rd_ptr=>ADDRB_ptr, empty=>empty, full=>full);


ram: block_ram

port map( d_in=> dataIn, ADDRA=> ADDRA_ptr(2 downto 0), ADDRB=> ADDRB_ptr(2 downto 0), WEA=> write_valid_signal, REA=> read_valid_signal, CLKA=> wclk, CLKB=> rclk, d_out=> dataOut);


END Architecture struct;
```

8. Round Robin Scheduler (M-ROU-08)

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

entity round_robin is
        port(
        din1, din2, din3, din4: in std_logic_vector(7 downto 0);
      clock: in std_logic;
        dout: out std_logic_vector(7 downto 0));
end entity round_robin;



architecture behav of round_robin is
        type state_type is (s1,s2,s3,s4);
      signal NS: state_type;
        signal CS: state_type := s1;
begin

        p1: process(clock)
begin

 if rising_edge(clock) then
CS<=NS;
else
CS<=CS;
        end if;
        end process p1;

  p2: process(CS)
begin
        case CS is
  when s1 =>
 NS <= s2; dout<= din1;
```

```
 when s2 =>
  NS <= s3;  dout <= din2;
when s3 =>
 NS <= s4; dout<=din3;
when s4 =>
 NS <= s1;  dout<= din4;

 end case;
 end process p2;


end architecture behav;
```

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

ENTITY controller IS
port (      rdclk: in std_logic;
            Fifo_1_empty,Fifo_2_empty,Fifo_3_empty,Fifo_4_empty: in std_logic;
            rd_req: out std_logic

    );
END ENTITY controller;

ARCHITECTURE controller OF controller is
type state is (fifo1,fifo2,fifo3,fifo4);
signal current_fifo: state;
signal next_fifo: state;

begin
p1:process (rdclk) is
begin
 if rising_edge(rdclk) then
 current_fifo <= next_fifo;
 end if;
end process p1;

p2:process (current_fifo,Fifo_1_empty,Fifo_2_empty,Fifo_3_empty,Fifo_4_empty) is
begin
case current_fifo is
when fifo1=>
  next_fifo<= fifo2;
  if Fifo_1_empty='0' then
  rd_req<='1';
  else rd_req<='0';
```

```
  end if;
when fifo2=>
  next_fifo<= fifo3;
  if Fifo_2_empty='0' then
  rd_req<='1';
  else rd_req<='0';
  end if;
when fifo3=>
  next_fifo<= fifo4;
  if Fifo_3_empty='0' then
  rd_req<='1';
  else rd_req<='0';
  end if;
when fifo4=>
  next_fifo<= fifo1;
  if Fifo_4_empty='0' then
  rd_req<='1';
  else rd_req<='0';
  end if;
end case;
end process p2;
end architecture controller;
```

9. 4-port Router (M-ROU-09)

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity Router is
port(datai1,datai2,datai3,datai4: in std_logic_vector(7 downto 0);
wr1, wr2, wr3, wr4: in std_logic;
datao1,datao2,datao3,datao4: out std_logic_vector(7 downto 0);
wclock, rclock, rst: in std_logic);
end Router;

architecture behav of Router is

component REG is
port(clk, Reset,clk_en : in std_logic;
      D_in : in  std_logic_vector (7 downto 0);
      D_out : out std_logic_vector (7 downto 0));
end component;
FOR ALL: REG USE ENTITY WORK.REG(Reg8);

component fifo is
port( reset, rclk, wclk, rreq, wreq: in std_logic;
     dataIn:in std_logic_vector(7 downto 0);
     dataOut:out std_logic_vector(7 downto 0);
     full,empty: out std_logic);
end component;
--FOR ALL: fifo USE ENTITY WORK.fifo(struct);

component demux is
 port(
```

```
 din : in STD_LOGIC_VECTOR (7 downto 0);
 Sel: in STD_LOGIC_VECTOR (1 downto 0);
 En: in STD_LOGIC;
 d_out1, d_out2, d_out3, d_out4 : out STD_LOGIC_VECTOR (7 downto 0)
 );
end component;
FOR ALL: demux USE ENTITY WORK.demux(demux8);

component round_robin is
        port(
        din1, din2, din3, din4: in std_logic_vector(7 downto 0);
      clock: in std_logic;
        dout: out std_logic_vector(7 downto 0));
end component;
FOR ALL: round_robin USE ENTITY WORK.round_robin(behav);

component controller is
port (      rdclk: in std_logic;
            Fifo_1_empty,Fifo_2_empty,Fifo_3_empty,Fifo_4_empty: in std_logic;
            rd_req: out std_logic);
end component;
FOR ALL: controller USE ENTITY WORK.controller(controller);

type arrOfVectors_sig is array(0 to 3) of std_logic_vector (7 downto 0);
type arrOfVectors_2D_sig is array (0 to 3,0 to 3) of std_logic_vector(7 downto 0);
type arr_2D_sig is array(0 to 3,0 to 3) of std_logic;
type arr4x1 is array (0 to 3) of std_logic;

signal RegOutput: arrOfVectors_sig;
signal OBOutput: arrOfVectors_sig;
signal DeMuxArrOutput: arrOfVectors_2D_sig;
signal FifoOutput: arrOfVectors_2D_sig;
signal FifotoRR: arrOfVectors_2D_sig;
signal Empty: arr_2D_sig;
```

```
signal Full: arr_2D_sig;
--signal Full: arr4x1;
signal controllerFlag: arr4x1;
signal Wrreq_arr: arr4x1;
--signal Rdreq: arr_2D_sig;
signal Wrreq:arr_2D_sig;
signal Data_in:arrOfVectors_sig;
signal Data_out:arrOfVectors_sig;

Begin

Wrreq_arr<=(wr1,wr2,wr3,wr4);
Data_in<=(datai1,datai2,datai3,datai4);

p1:process (Data_in,Wrreq_arr,wclock)
is begin
loop1:for i in 0 to 3 loop
loop2:for j in 0 to 3 loop
 if Wrreq_arr(i)= '1' and RegOutput(i)(1 downto 0)=j then
 Wrreq(i,j) <='1';
 else Wrreq(i,j) <='0';
end if;
end loop loop2;
end loop loop1;
end process p1;

IB1 : REG
port map(clk=> wclock, D_in=> datai1, reset=>rst, clk_en=>wr1, D_out=>RegOutput(0));
IB2 : REG
port map(clk=> wclock, D_in=> datai2, reset=>rst, clk_en=>wr2, D_out=>RegOutput(1));
IB3 : REG
port map(clk=> wclock, D_in=> datai3, reset=>rst, clk_en=>wr3, D_out=>RegOutput(2));
IB4 : REG
port map(clk=> wclock, D_in=> datai4, reset=>rst, clk_en=>wr4, D_out=>RegOutput(3));
```

DeMux1: DEMUX

port map(din=> RegOutput(0), Sel=> (RegOutput(0)(1 downto 0)), En=> wr1, d_out1=>DeMuxArrOutput(0,0),

d_out2=>DeMuxArrOutput(0,1),d_out3=>DeMuxArrOutput(0,2),d_out4=>DeMuxArrOutput(0,3));

DeMux2: DEMUX

port map(din=> RegOutput(1), Sel=> (RegOutput(1)(1 downto 0)), En=> wr2, d_out1=>DeMuxArrOutput(1,0),

d_out2=>DeMuxArrOutput(1,1),d_out3=>DeMuxArrOutput(1,2),d_out4=>DeMuxArrOutput(1,3));

DeMux3: DEMUX

port map(din=> RegOutput(2), Sel=> (RegOutput(2)(1 downto 0)), En=> wr3, d_out1=>DeMuxArrOutput(2,0),

d_out2=>DeMuxArrOutput(2,1),d_out3=>DeMuxArrOutput(2,2),d_out4=>DeMuxArrOutput(2,3));

DeMux4: DEMUX

port map(din=> RegOutput(3), Sel=> (RegOutput(3)(1 downto 0)), En=> wr4, d_out1=>DeMuxArrOutput(3,0),

d_out2=>DeMuxArrOutput(3,1),d_out3=>DeMuxArrOutput(3,2),d_out4=>DeMuxArrOutput(3,3));

RR1: round_robin

port map(clock => rclock, din1=> FifotoRR(0,0), din2=> FifotoRR(0,1), din3=> FifotoRR(0,2), din4=> FifotoRR(0,3), dout=> Data_out(0));

RR2: round_robin

port map(clock => rclock, din1=> FifotoRR(1,0), din2=> FifotoRR(1,1), din3=> FifotoRR(1,2), din4=> FifotoRR(1,3), dout=> Data_out(1));

RR3: round_robin

port map(clock => rclock, din1=> FifotoRR(2,0), din2=> FifotoRR(2,1), din3=> FifotoRR(2,2), din4=> FifotoRR(2,3), dout=> Data_out(2));

RR4: round_robin

port map(clock => rclock, din1=> FifotoRR(3,0), din2=> FifotoRR(3,1), din3=> FifotoRR(3,2), din4=> FifotoRR(3,3), dout=> Data_out(3));

CTRL1: controller

port map(rdclk=> rclock, Fifo_1_empty=> Empty(0,0), Fifo_2_empty=> Empty(0,1), Fifo_3_empty=> Empty(0,2), Fifo_4_empty=> Empty(0,3), rd_req=> controllerFlag(0));

CTRL2: controller

port map(rdclk=> rclock, Fifo_1_empty=> Empty(1,0), Fifo_2_empty=> Empty(1,1), Fifo_3_empty=> Empty(1,2), Fifo_4_empty=> Empty(1,3), rd_req=> controllerFlag(1));

CTRL3: controller

port map(rdclk=> rclock, Fifo_1_empty=> Empty(2,0), Fifo_2_empty=> Empty(2,1), Fifo_3_empty=> Empty(2,2), Fifo_4_empty=> Empty(2,3), rd_req=> controllerFlag(2));

CTRL4: controller

port map(rdclk=> rclock, Fifo_1_empty=> Empty(3,0), Fifo_2_empty=> Empty(3,1), Fifo_3_empty=> Empty(3,2), Fifo_4_empty=> Empty(3,3), rd_req=> controllerFlag(3));

Fifo_outer_loop: FOR i IN 0 TO 3 GENERATE

begin

Fifo_inner_loop: FOR j IN 0 TO 3 GENERATE

FOR ALL: fifo USE ENTITY WORK.fifo(struct);

begin

routerFIFO: fifo

port map(reset=> rst, rclk=> rclock, wclk=> wclock, rreq=> controllerFlag(i), wreq=> Wrreq(i,j), dataIn=> DeMuxArrOutput(j,i), dataOut=> FifotoRR(i, j), full=> Full(i,j), empty=> Empty(i,j));

END GENERATE Fifo_inner_loop;

END GENERATE Fifo_outer_loop;

OB1 : REG

port map(clk=> rclock, D_in=> Data_out(0), reset=>rst, clk_en=>'1', D_out=>OBOutput(0));

OB2 : REG

port map(clk=> rclock, D_in=> Data_out(1), reset=>rst, clk_en=>'1', D_out=>OBOutput(1));

OB3 : REG

port map(clk=> rclock, D_in=> Data_out(2), reset=>rst, clk_en=>'1', D_out=>OBOutput(2));

OB4 : REG

port map(clk=> rclock, D_in=> Data_out(3), reset=>rst, clk_en=>'1', D_out=>OBOutput(3));

datao1<= OBOutput(0);

datao2<= OBOutput(1);

datao3<= OBOutput(2);

datao4<= OBOutput(3);

END Architecture behav;

APPENDIX B (Test Bench)

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;



entity Router_testbench is
end Router_testbench;

Architecture behav of Router_testbench IS

component Router is
port(datai1,datai2,datai3,datai4: in std_logic_vector(7 downto 0);
wr1, wr2, wr3, wr4: in std_logic;
datao1,datao2,datao3,datao4: out std_logic_vector(7 downto 0);
wclock, rclock, rst: in std_logic);
end component;
FOR ALL: Router USE ENTITY WORK.Router(behav);

signal datai1,datai2,datai3,datai4: std_logic_vector( 7 downto 0);
signal wr1, wr2, wr3, wr4: std_logic;
signal datao1,datao2,datao3,datao4: std_logic_vector( 7 downto 0);
signal wclock, rclock, rst: std_logic;
  CONSTANT clk_period : TIME := 100 ns;

Begin

DUT : Router
port map( rst=> rst, rclock=> rclock, wclock=> wclock,
        datai1=> datai1, datai2=> datai2, datai3=> datai3, datai4=> datai4,
        wr1=> wr1, wr2=> wr2, wr3=> wr3, wr4=> wr4,
```

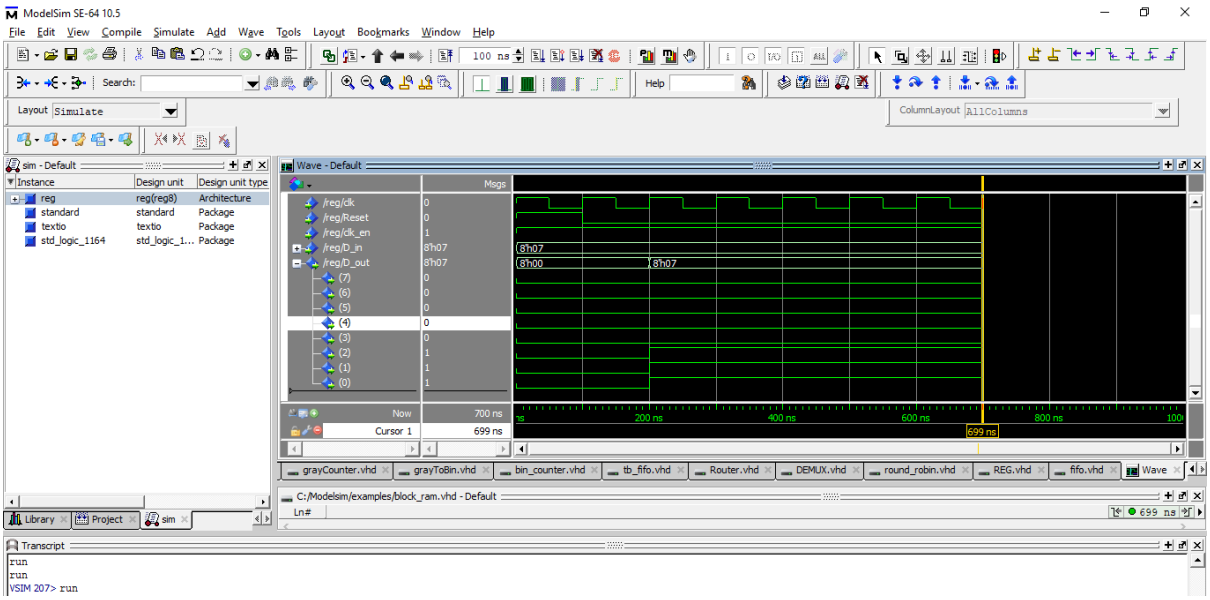
```
      datao1=> datao1, datao2=> datao2, datao3=> datao3, datao4=> datao4);
 rclk_process : PROCESS
 BEGIN
    rclock <= '0';
    WAIT FOR clk_period/2;
    rclock <= '1';
    WAIT FOR clk_period/2;
 END PROCESS;

 wrclk_process : PROCESS
 BEGIN
    wclock <= '0';
    WAIT FOR clk_period/2;
    wclock <= '1';
    WAIT FOR clk_period/2;

 END PROCESS;

 stim_p : PROCESS
 BEGIN
    rst <= '1';
    WAIT FOR clk_period;
    datai1 <= "00000100";
    datai2 <= "00000011";
    datai3 <= "00000010";
    datai4 <= "00000001";
    rst <= '0';
    wr1 <= '1';
    wr2 <= '1';
    wr3 <= '1';
    wr4 <= '1';
    WAIT FOR clk_period;
    datai1 <= "00010000";
    datai2 <= "00100000";
```

```
datai3 <= "00110000";
datai4 <= "01000000";
WAIT FOR clk_period;
wr1 <= '0';
wr2 <= '0';
wr3 <= '0';
wr4 <= '0';
WAIT FOR clk_period;
ASSERT datao2 = "00000001"
   SEVERITY Error;

WAIT FOR clk_period;
ASSERT datao1 = "00000100"
   SEVERITY Error;

WAIT FOR clk_period;
ASSERT datao4 = "00000011"
   SEVERITY Error;
ASSERT datao1 = "00100000"
   SEVERITY Error;

WAIT FOR clk_period;
ASSERT datao3 = "00000010"
   SEVERITY Error;
ASSERT datao1 = "00110000"
   SEVERITY Error;

WAIT FOR clk_period;
ASSERT datao1 = "01000000"
   SEVERITY Error;

WAIT FOR clk_period;
ASSERT datao1 = "00010000"
   SEVERITY Error;
```

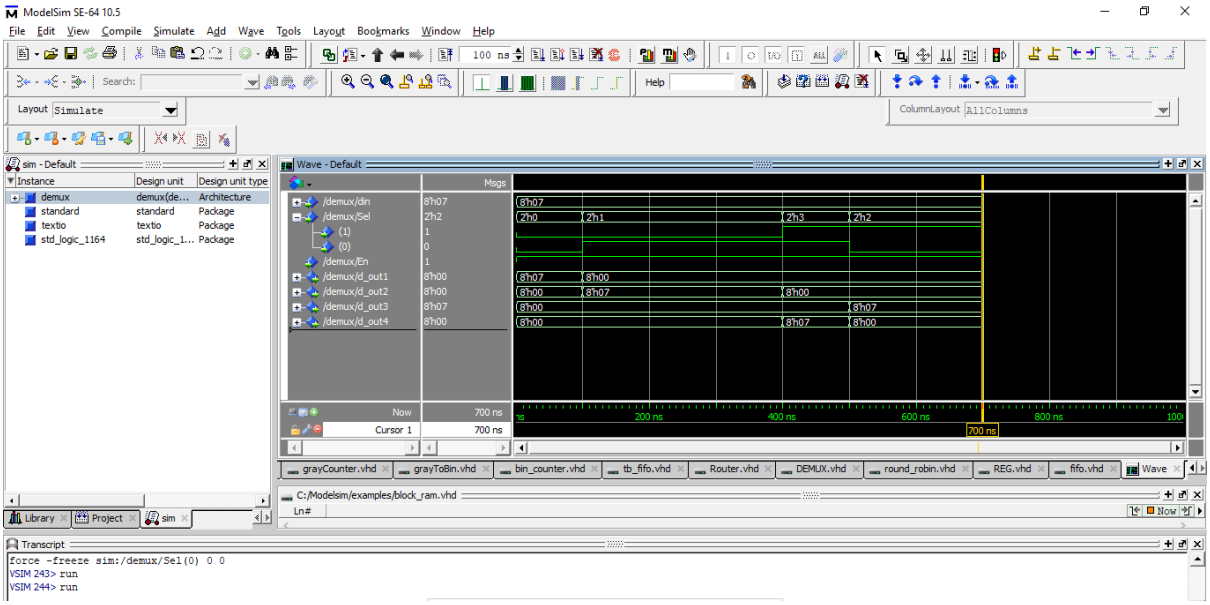
```
        wr2 <= '1';
        datai2 <= "10100001";
        WAIT FOR clk_period * 10;
        datai2 <= "11110001";
        WAIT FOR clk_period;
        wr2 <= '0';

        WAIT FOR clk_period * 30;
        ASSERT datao2 = "10100001"
           SEVERITY Error;

        datai4 <= "11110011";
        wr4 <= '1';
        WAIT FOR clk_period * 2;
        rst <= '1';
        wr4 <= '0';
        WAIT FOR clk_period;
        datai4 <= "11010011";
        wr4 <= '1';
        rst <= '0';
        WAIT FOR clk_period;
        wr4 <= '0';
        WAIT FOR clk_period * 2;
        ASSERT datao4 = "11010011"
           SEVERITY Error;

        WAIT;
    END PROCESS;

END;
```

APPENDIX C (Wave Form)

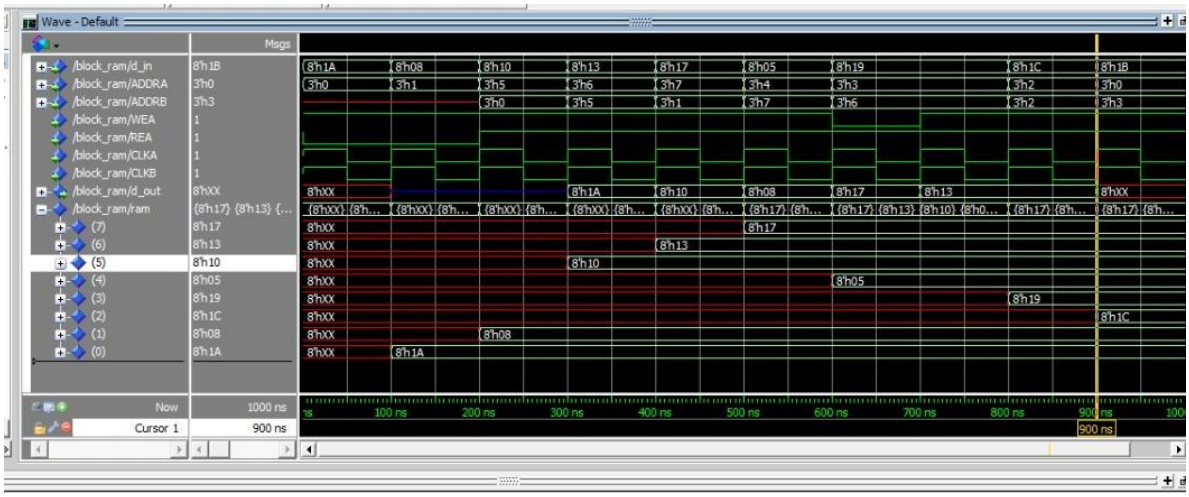
1. 8-bit Register (M-ROU-01)



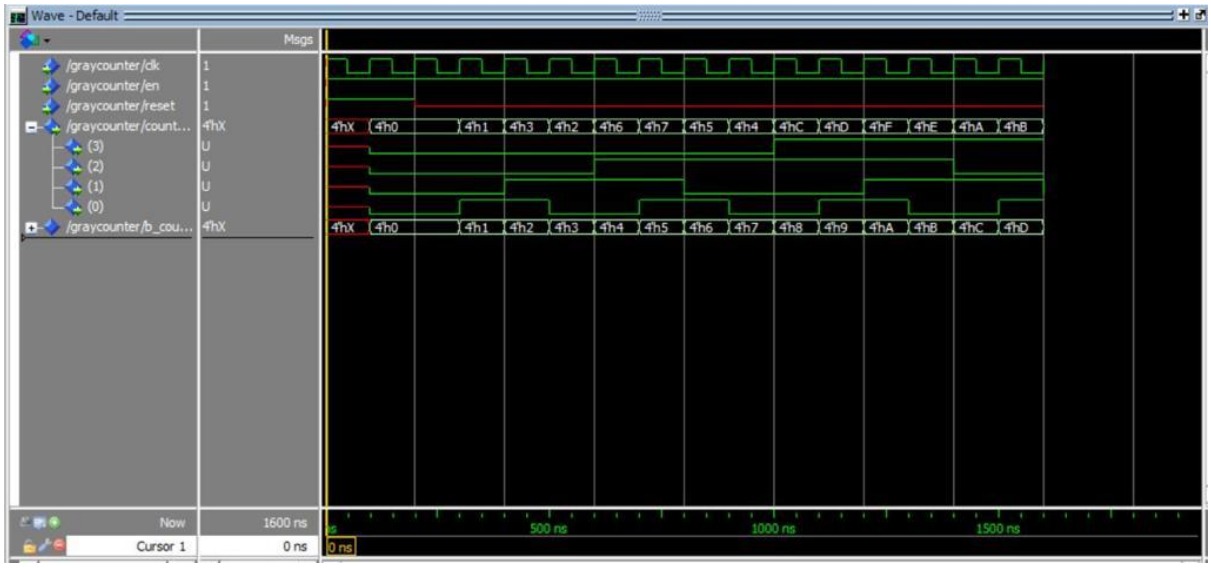
2. 8-bit Demux (M-ROU-02)



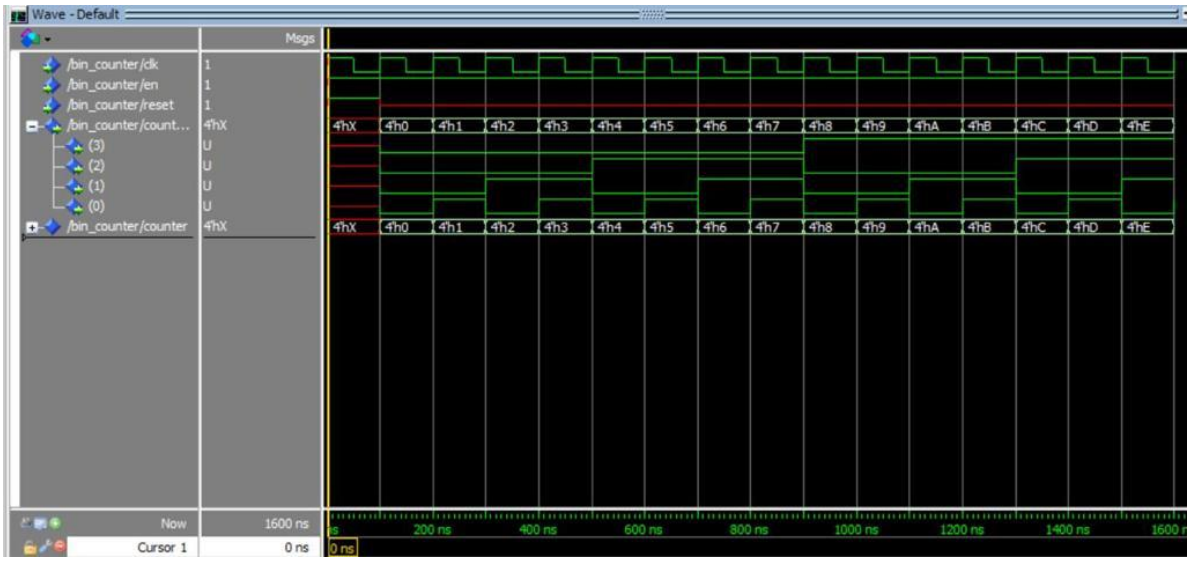
3. *Block RAM (M-ROU-03)*



4. *Gray Counter (M-ROU-04)*

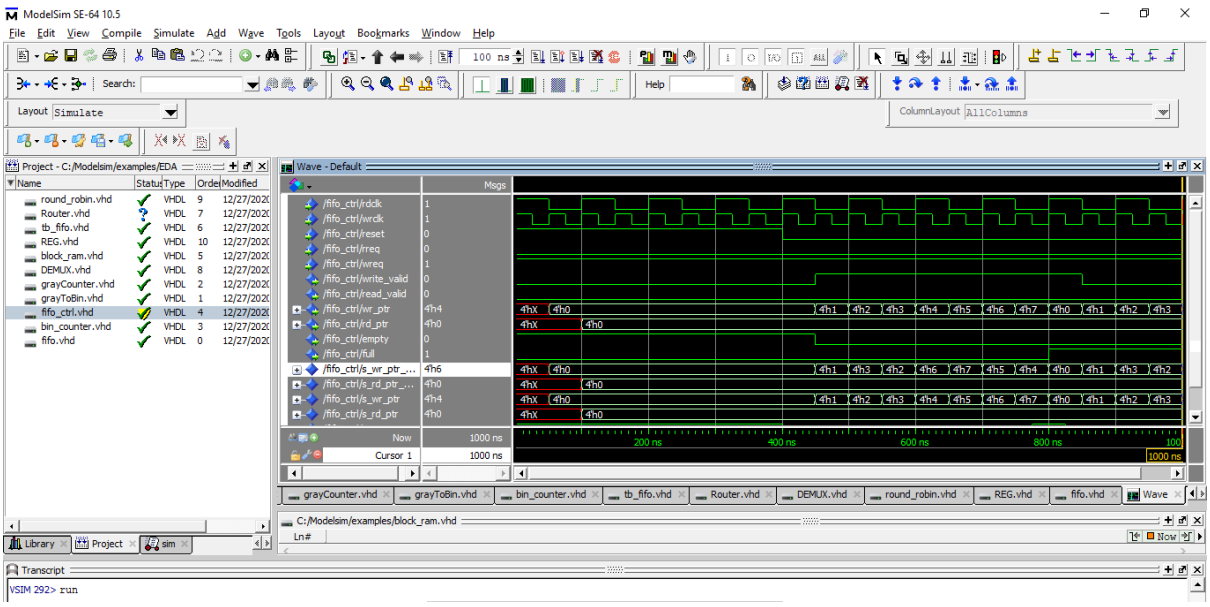


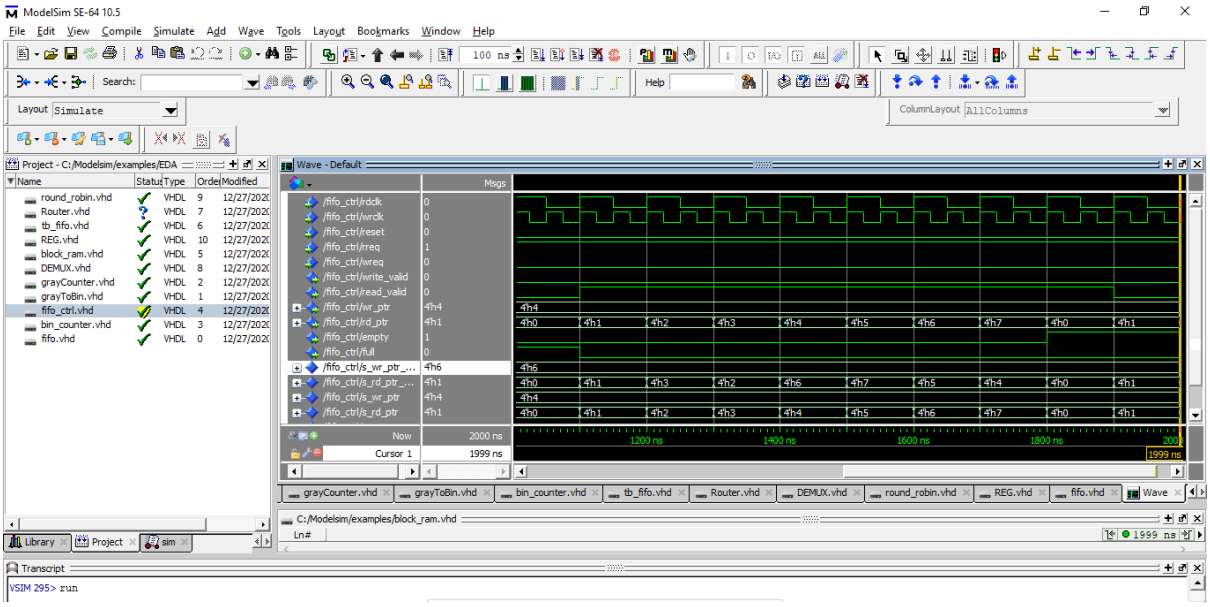
Binary Counter



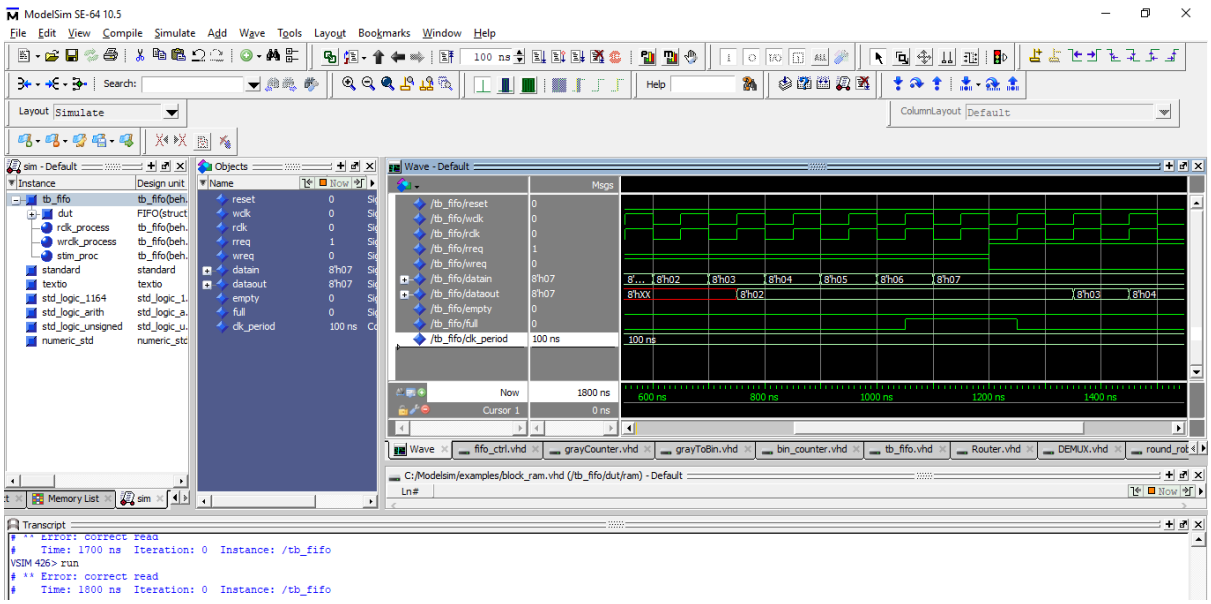
5. *Gray to Binary Converter (M-ROU-05)*

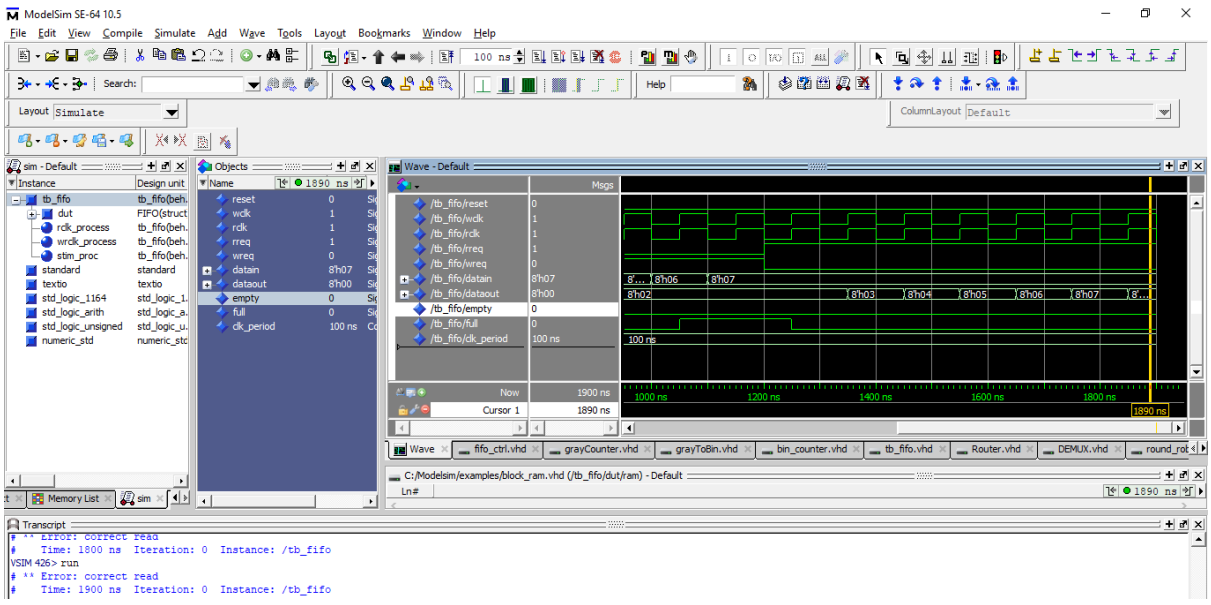
6. *FIFO Controller (M-ROU-06)*



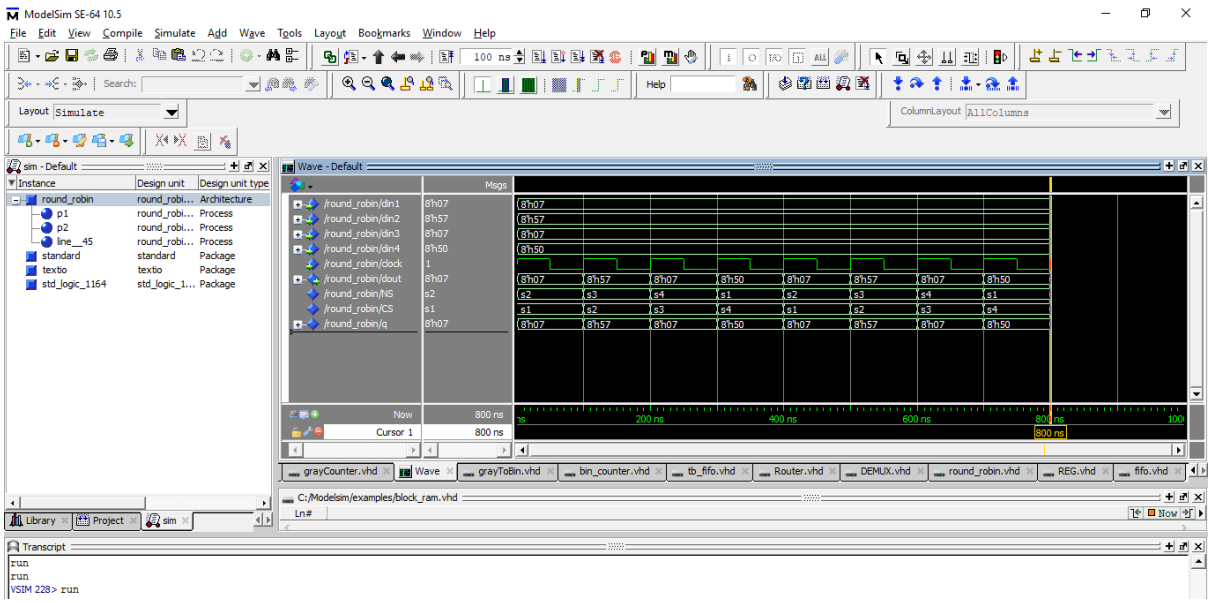


7. FIFO (M-ROU-07)

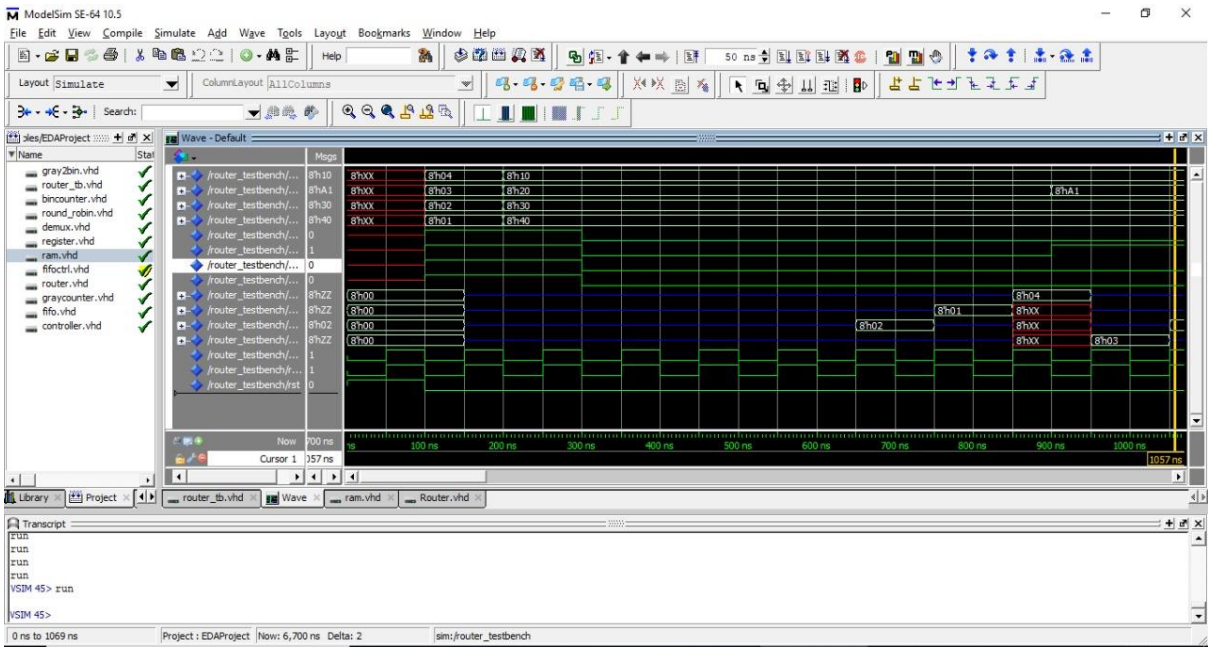


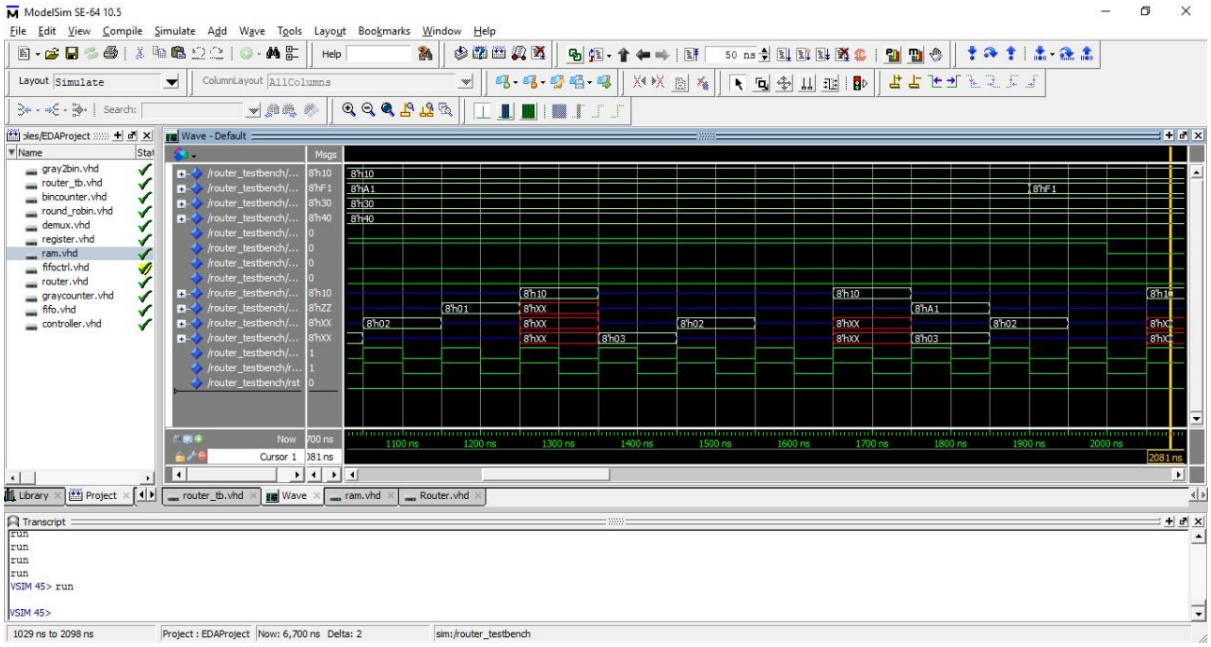


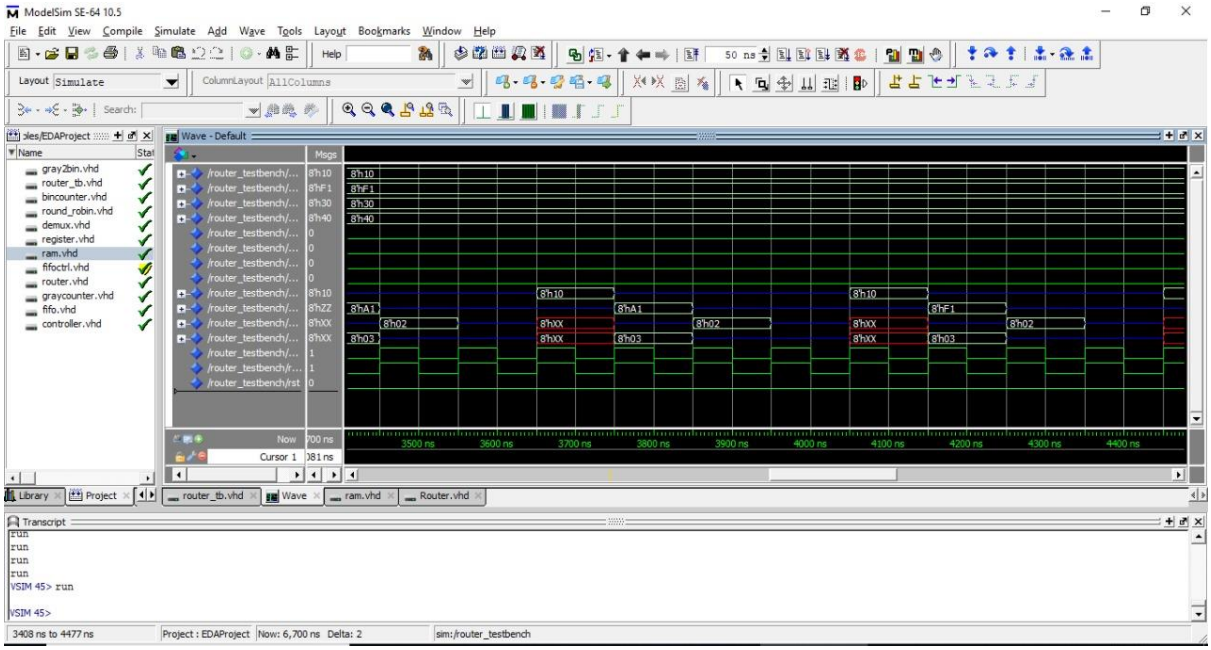
8. Round Robin Scheduler (M-ROU-08)



9. 4-port Router (M-ROU-09)



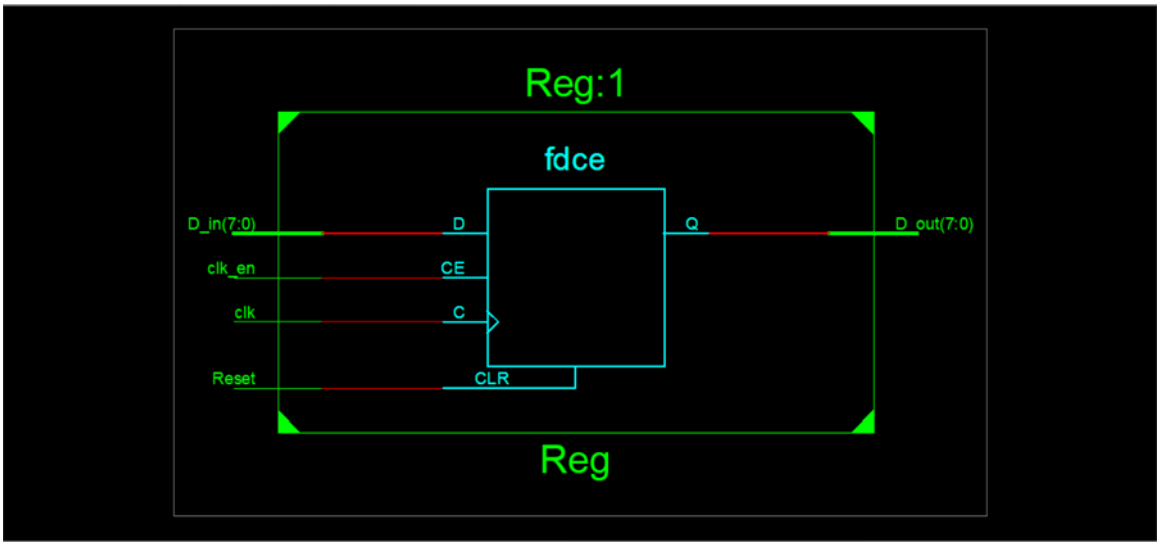

ModelSim SE-64 10.5
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
gray2bin.vhd
router_tb.vhd
bincounter.vhd
round_robin.vhd
demux.vhd
register.vhd
ram.vhd
fifoctrl.vhd
router.vhd
graycounter.vhd
fifo.vhd
controller.vhd
Transcript
run
run
run
run
VSIM 45> run
VSIM 45>
1029 ns to 2098 ns
Project : EDAProject
Now: 6,700 ns Delta: 2
sim:/router_testbench


ModelSim SE-64 10.5
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
gray2bin.vhd
router_tb.vhd
bincounter.vhd
round_robin.vhd
demux.vhd
register.vhd
ram.vhd
fifoctrl.vhd
router.vhd
graycounter.vhd
fifo.vhd
controller.vhd
Transcript
run
run
run
run
VSIM 45> run
VSIM 45>
3408 ns to 4477 ns
Project : EDAProject
Now: 6,700 ns Delta: 2
sim:/router_testbench

Appendix D (Synthesis)

This Appendix contains gate count, device utilization summary and and timing summary for all the modules.

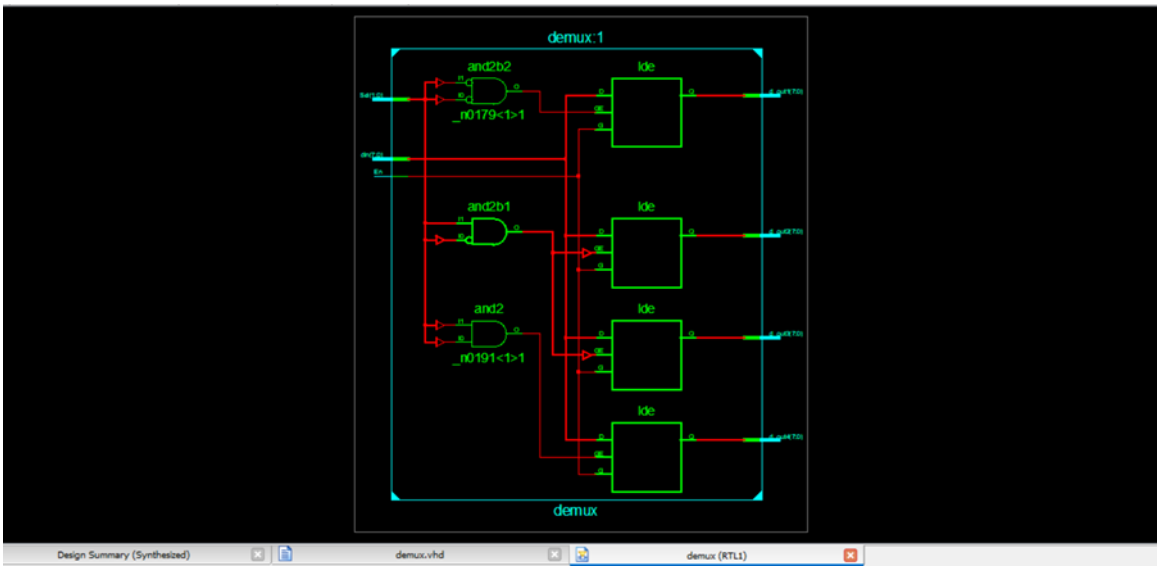
1. *8-bit Register (M-ROU-01)*




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      0
   Number with an unused Flip Flop:       0  out of      0
   Number with an unused LUT:             0  out of      0
   Number of fully used LUT-FF pairs:     0  out of      0
   Number of unique control sets:         1

IO Utilization:
 Number of IOs:                          19
 Number of bonded IOBs:                  19  out of    102    18%
    IOB Flip Flops/Latches:               8

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: 2.454ns
   Maximum output required time after clock: 3.597ns
   Maximum combinational path delay: No path found
```

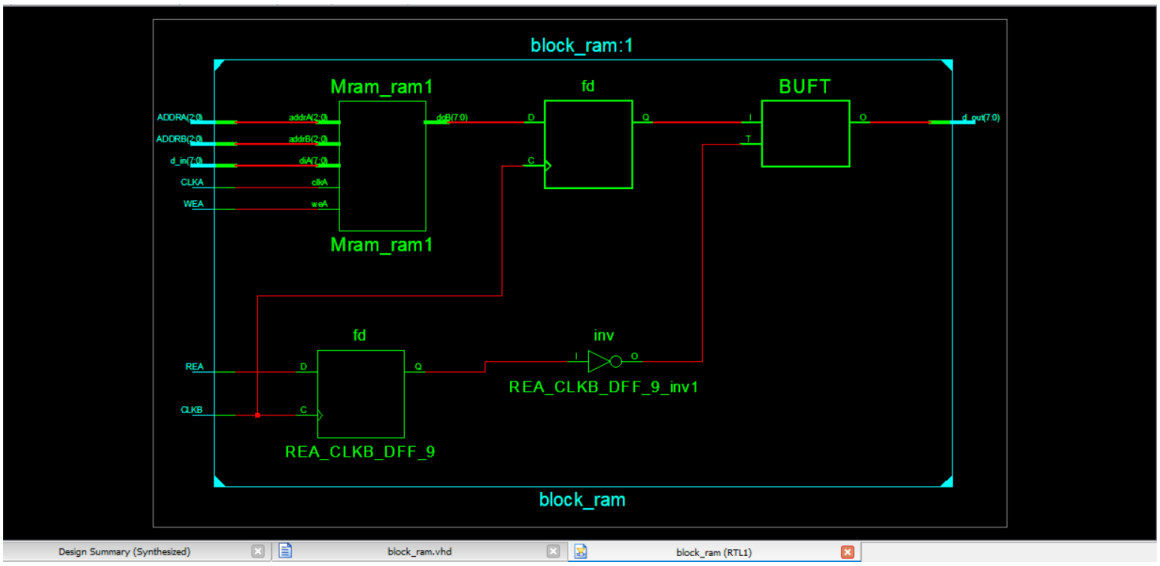
2. 8-bit Demux (M-ROU-02)




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice LUTs:                    4  out of   2400     0%
    Number used as Logic:                 4  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      4
   Number with an unused Flip Flop:       4  out of      4   100%
   Number with an unused LUT:             0  out of      4     0%
   Number of fully used LUT-FF pairs:     0  out of      4     0%
   Number of unique control sets:         4

IO Utilization:
 Number of IOs:                          43
 Number of bonded IOBs:                  43  out of    102    42%
    IOB Flip Flops/Latches:              32

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: 3.336ns
   Maximum output required time after clock: 3.648ns
   Maximum combinational path delay: No path found
```

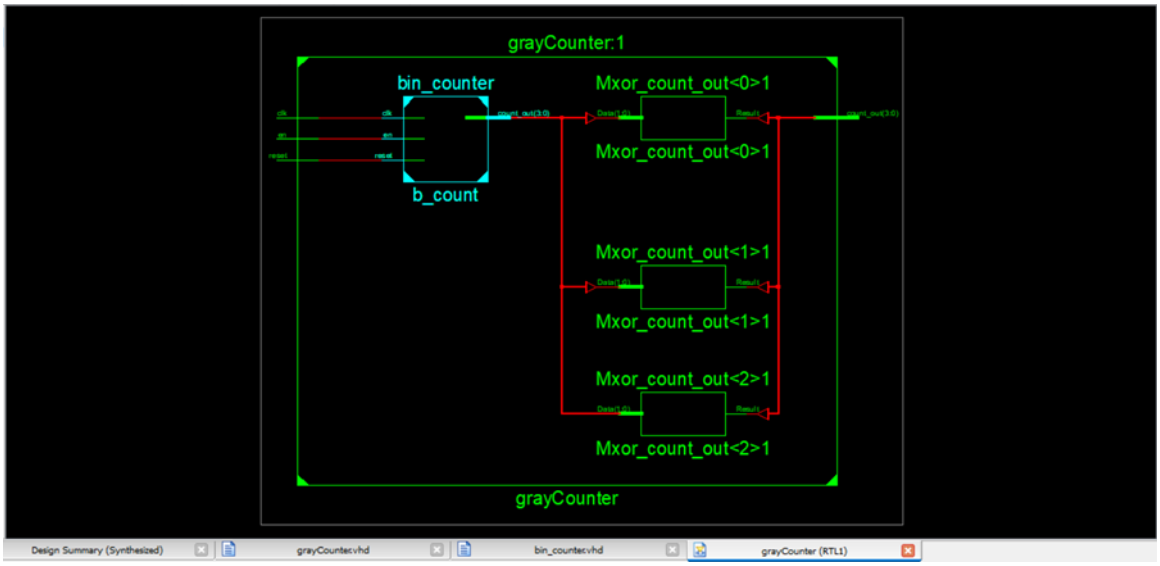
3. *Block RAM (M-ROU-03)*




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice LUTs:                    9  out of   2400     0%
    Number used as Logic:                 1  out of   2400     0%
    Number used as Memory:                8  out of   1200     0%
       Number used as RAM:                8

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      9
   Number with an unused Flip Flop:       9  out of      9   100%
   Number with an unused LUT:             0  out of      9     0%
   Number of fully used LUT-FF pairs:     0  out of      9     0%
   Number of unique control sets:         1

IO Utilization:
 Number of IOs:                          26
 Number of bonded IOBs:                  26  out of    102    25%
    IOB Flip Flops/Latches:               9

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               2  out of     16    12%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: 2.822ns
   Maximum output required time after clock: 4.604ns
   Maximum combinational path delay: No path found
```

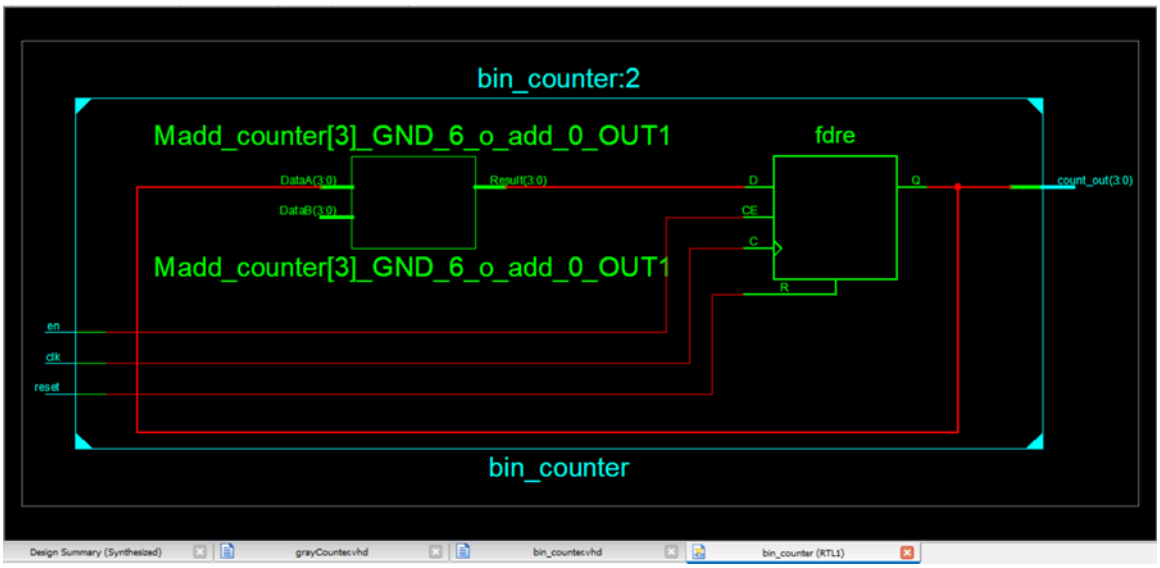
4. *Gray Counter (M-ROU-04)*




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:              4  out of   4800     0%
 Number of Slice LUTs:                   6  out of   2400     0%
    Number used as Logic:                6  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    10
   Number with an unused Flip Flop:      6  out of     10    60%
   Number with an unused LUT:            4  out of     10    40%
   Number of fully used LUT-FF pairs:    0  out of     10     0%
   Number of unique control sets:        1

IO Utilization:
 Number of IOs:                          7
 Number of bonded IOBs:                  7  out of    102     6%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               1  out of     16     6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.016ns (Maximum Frequency: 495.933MHz)
   Minimum input arrival time before clock: 2.335ns
   Maximum output required time after clock: 4.625ns
   Maximum combinational path delay: No path found
```

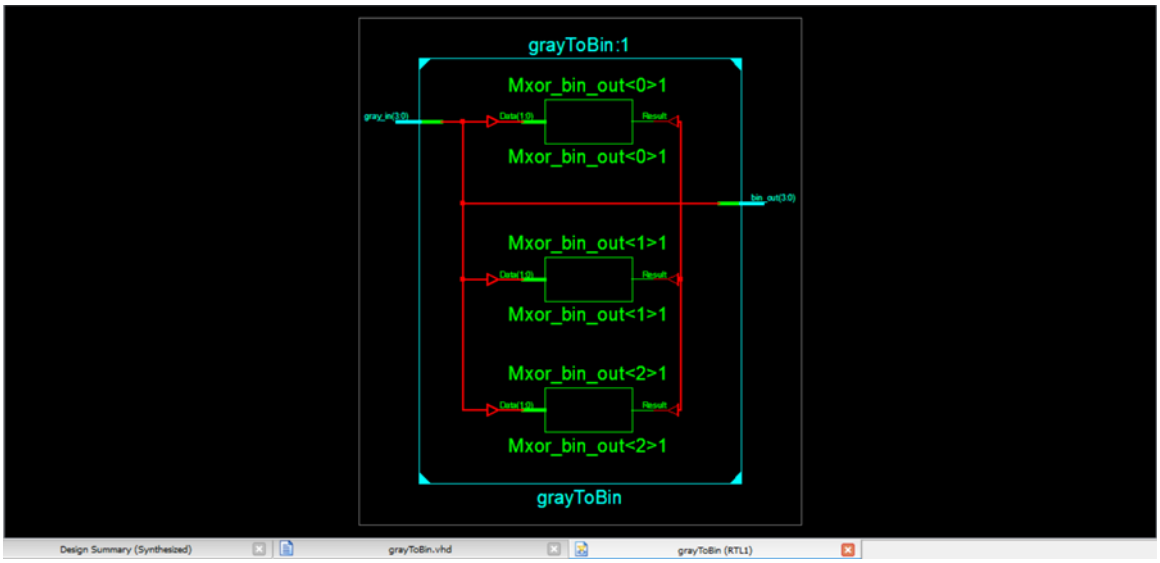
Binary Counter




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:                4  out of   4800     0%
 Number of Slice LUTs:                     4  out of   2400     0%
    Number used as Logic:                  4  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:       8
   Number with an unused Flip Flop:        4  out of      8    50%
   Number with an unused LUT:              4  out of      8    50%
   Number of fully used LUT-FF pairs:      0  out of      8     0%
   Number of unique control sets:          1

IO Utilization:
 Number of IOs:                            7
 Number of bonded IOBs:                    7  out of    102     6%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                 1  out of     16     6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.048ns (Maximum Frequency: 488.317MHz)
   Minimum input arrival time before clock: 2.335ns
   Maximum output required time after clock: 3.732ns
   Maximum combinational path delay: No path found
```

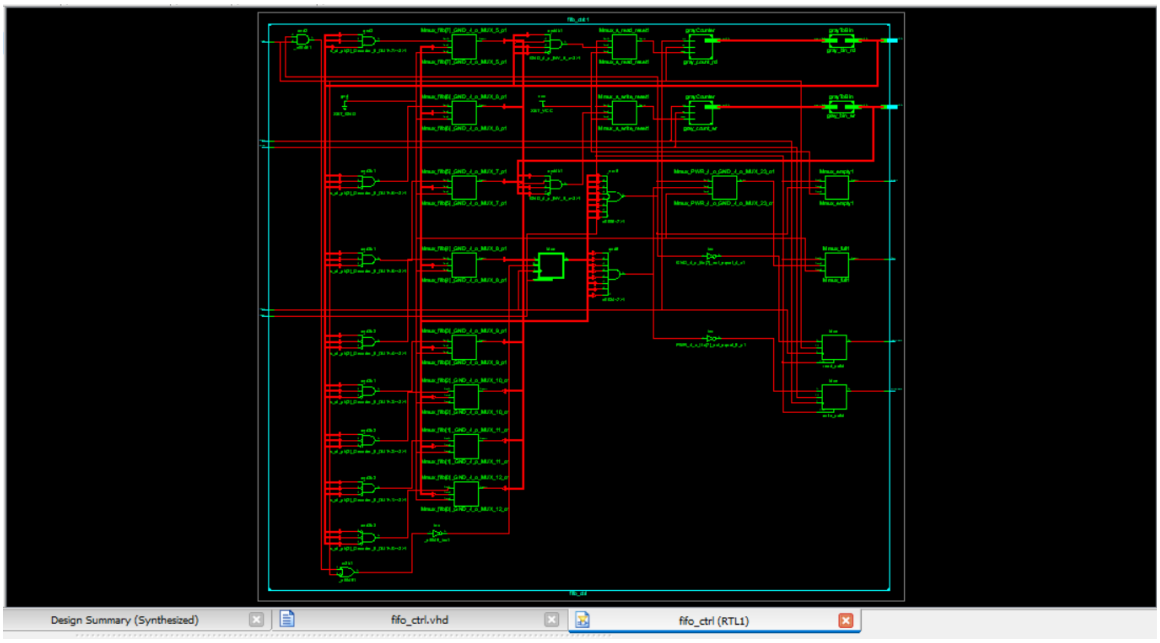
5. *Gray to Binary Converter (M-ROU-05)*




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice LUTs:                  3  out of   2400     0%
    Number used as Logic:               3  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    3
   Number with an unused Flip Flop:     3  out of      3   100%
   Number with an unused LUT:           0  out of      3     0%
   Number of fully used LUT-FF pairs:   0  out of      3     0%
   Number of unique control sets:       0

IO Utilization:
 Number of IOs:                         8
 Number of bonded IOBs:                 8  out of    102     7%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 5.422ns
```

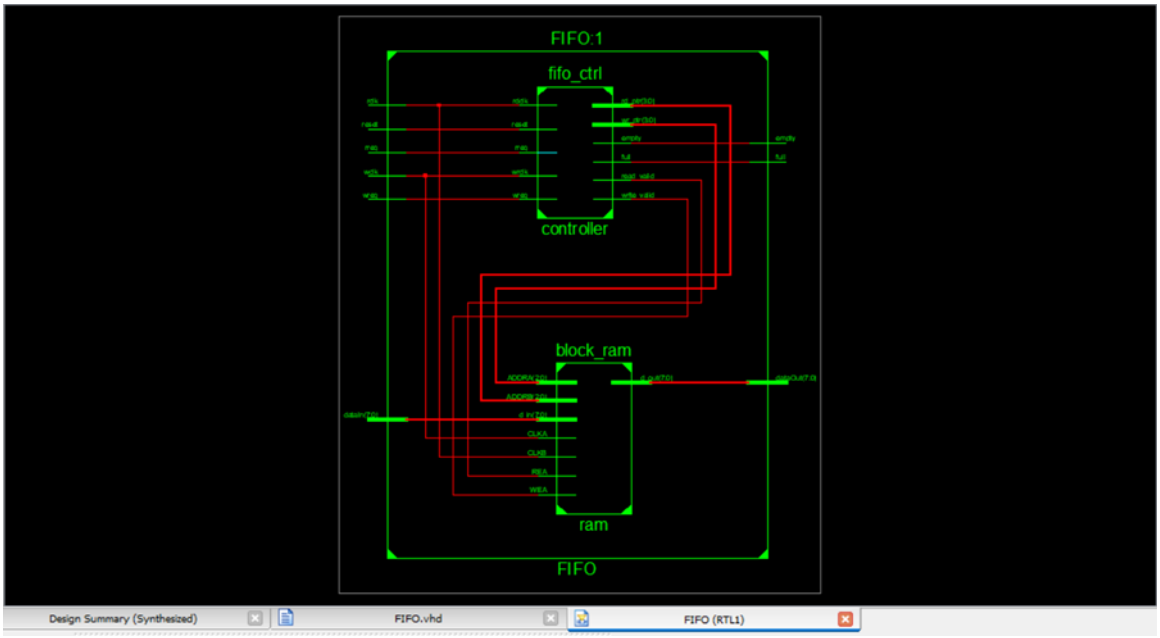
6. *FIFO Controller (M-ROU-06)*




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:                8  out of   4800     0%
 Number of Slice LUTs:                     8  out of   2400     0%
    Number used as Logic:                  8  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      16
   Number with an unused Flip Flop:        8  out of     16    50%
   Number with an unused LUT:              8  out of     16    50%
   Number of fully used LUT-FF pairs:      0  out of     16     0%
   Number of unique control sets:          3

IO Utilization:
 Number of IOs:                           17
 Number of bonded IOBs:                   17  out of    102    16%
    IOB Flip Flops/Latches:                1

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                2  out of     16    12%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 1.714ns (Maximum Frequency: 583.431MHz)
   Minimum input arrival time before clock: 2.721ns
   Maximum output required time after clock: 3.732ns
   Maximum combinational path delay: No path found
```

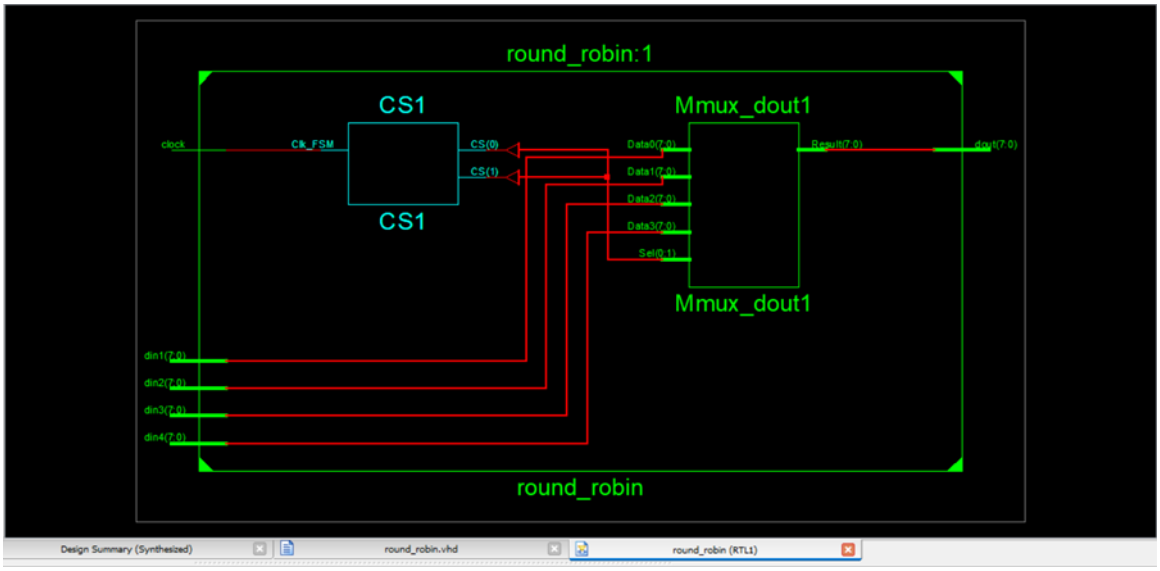
7. FIFO (M-ROU-07)




```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:             11  out of   4800     0%
 Number of Slice LUTs:                  16  out of   2400     0%
    Number used as Logic:                8  out of   2400     0%
    Number used as Memory:               8  out of   1200     0%
       Number used as RAM:               8

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    19
   Number with an unused Flip Flop:      8  out of     19    42%
   Number with an unused LUT:            3  out of     19    15%
   Number of fully used LUT-FF pairs:    8  out of     19    42%
   Number of unique control sets:        3

IO Utilization:
 Number of IOs:                         23
 Number of bonded IOBs:                 23  out of    102    22%
    IOB Flip Flops/Latches:              6

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:              2  out of     16    12%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 2.162ns (Maximum Frequency: 462.620MHz)
   Minimum input arrival time before clock: 2.721ns
   Maximum output required time after clock: 3.597ns
   Maximum combinational path delay: No path found
```

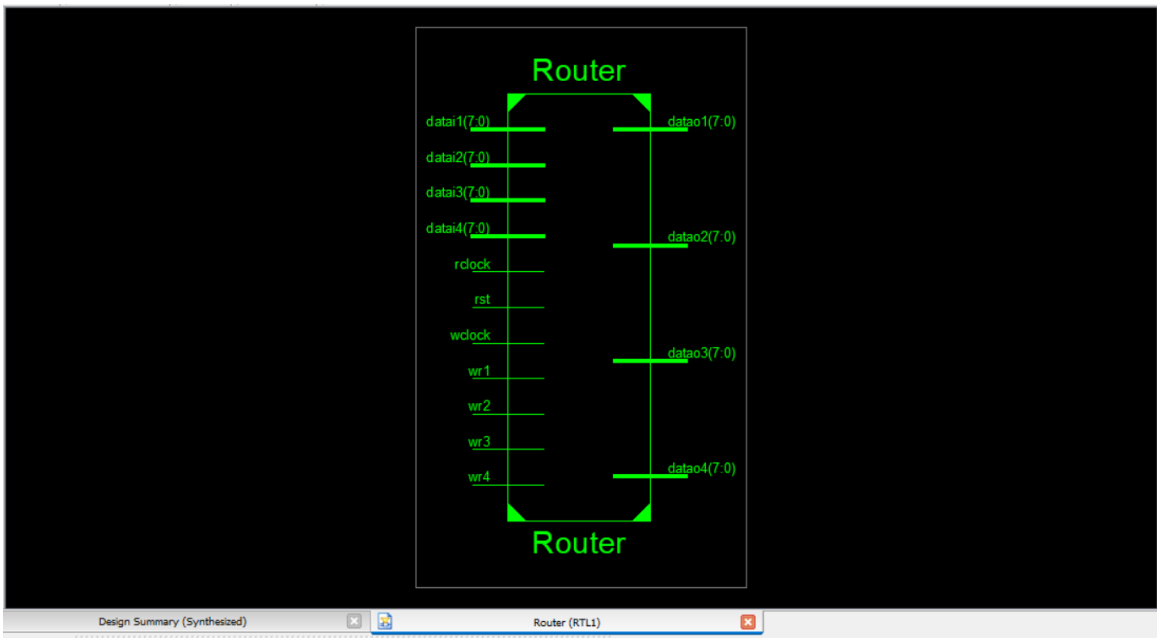
8. *Round Robin Scheduler (M-ROU-08)*

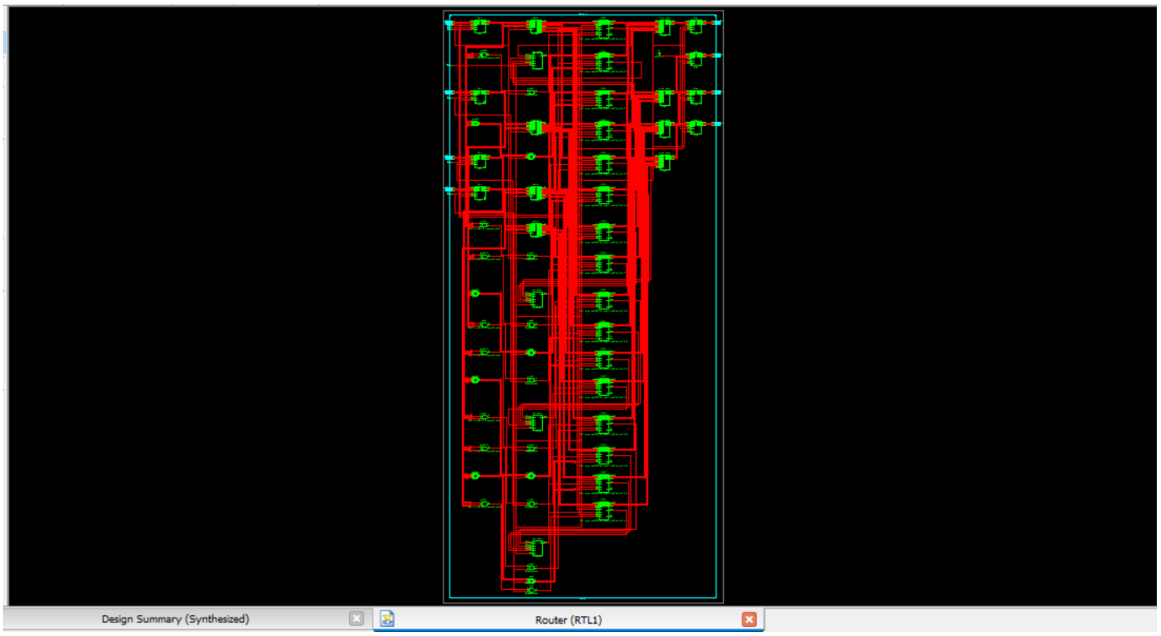



```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:
 Number of Slice Registers:               2  out of   4800     0%
 Number of Slice LUTs:                   10  out of   2400     0%
    Number used as Logic:                10  out of   2400     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     10
   Number with an unused Flip Flop:       8  out of     10    80%
   Number with an unused LUT:             0  out of     10     0%
   Number of fully used LUT-FF pairs:     2  out of     10    20%
   Number of unique control sets:         1

IO Utilization:
 Number of IOs:                          41
 Number of bonded IOBs:                  41  out of    102    40%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%
```

```
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
```

9. 4-port Router (M-ROU-09)





```
Device utilization summary:
---------------------------

Selected Device : 6slx4tqg144-3


Slice Logic Utilization:

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      0
   Number with an unused Flip Flop:       0  out of      0
   Number with an unused LUT:             0  out of      0
   Number of fully used LUT-FF pairs:     0  out of      0
   Number of unique control sets:         1

IO Utilization:
 Number of IOs:                          71
 Number of bonded IOBs:                  34  out of    102    33%
    IOB Flip Flops/Latches:              32

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16     6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: 2.943ns
   Maximum output required time after clock: 3.597ns
   Maximum combinational path delay: No path found
```