Network on Chip

***Abstract*— This project involves design and implementation of a 4 by 4 network on chip(NOC) using Verilog.**

***Index Terms*— Network on-chip, Buffer unit, Switch, Packet, Flit, Node, Router, Verilog, Verification**

I. INTRODUCTION

Network on chip (NOC) is a trending topic that enables the connection of multiple processing units on a single chip. NOC comprises switches and nodes; nodes represent processing units, and routers are modules responsible for finding a path between two nodes and transporting packets between them. In this computer assignment, we aim to design and implement a 4x4 NOC using Verilog. The topology of the NOC is mesh, and the routing algorithm employed is X-Y. In the following paragraphs, we will design different parts of this NOC.

.

II. BUFFER UNIT

Each router has 5 buffer units for five different directions: local, west, north, east, and south. The responsibility of a buffer unit is to save data in a first-in-first-out (FIFO) memory when other units try to transport data to them. In a NOC, each piece of data is divided into packets, and each packet is further split into flits. There are three types of flits: header flit, payload flit, and tail flit. Each flit is 18 bits long, containing 16 bits of data and two bits that indicate the flit type. The header flit is the first flit and contains information such as the destination node ID, source node ID, and packet number. This information is useful for finding a path between the source and destination. The tail flit contains 16 bits of data and is the last flit of the packet. Figure 1 shows the structure of different types of flit.

	17:16	15:8	7:4	3:0
Header flit	01	Packet number	Source	Destination
payload	00			
tail	10			

Fig 1. Structure of flit

A. FIFO

FIFO is a type of memory that writes data in order and, when you attempt to read data, gives you the first data that was written in the memory. To implement this, we need a memory that allows you to write data at any address. Two pointers are necessary: one for the write address and one for the read address of the memory. When you write data to the FIFO, it is written to the address indicated by the write pointer, and the pointer is incremented. Conversely, each attempt to read data leads to reading from the read pointer address and incrementing this pointer. Whenever the two pointers indicate the same address, it means that the FIFO is either full or empty, depending on the last operation. If you write something to the FIFO and the two pointers become equal, it means that the FIFO is full. In contrast, any attempt to read data that leads to the equality of the two pointers indicates that the FIFO is empty. Therefore, it is necessary to save the last operation in a register to be capable of issuing full and empty flags. Figure 2 illustrates the RTL design of a FIFO.

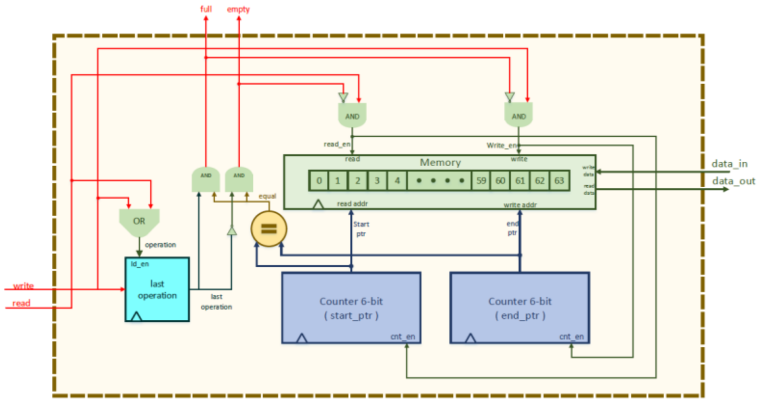
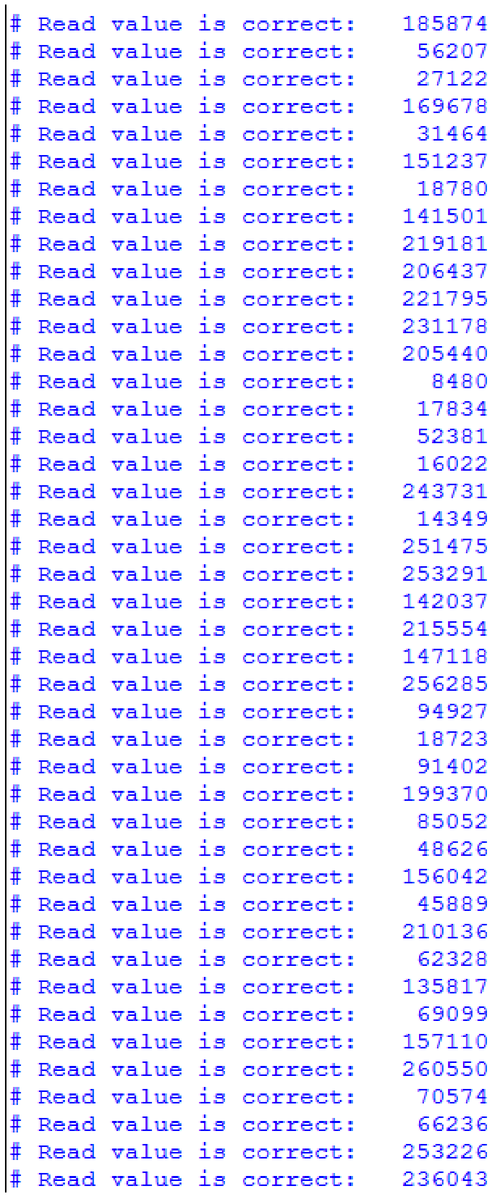


Fig 2. FIFO RTL design

After implementing the FIFO in Verilog, a testbench was developed to verify its functionality. In this testbench, we first save 64 random data values in the FIFO, then read the data one by one to check if the values are the same. Figure 3 shows the testbench results.



```
# Read value is correct:   185874
# Read value is correct:    56207
# Read value is correct:    27122
# Read value is correct:   169678
# Read value is correct:    31464
# Read value is correct:   151237
# Read value is correct:    18780
# Read value is correct:   141501
# Read value is correct:   219181
# Read value is correct:   206437
# Read value is correct:   221795
# Read value is correct:   231178
# Read value is correct:   205440
# Read value is correct:     8480
# Read value is correct:    17834
# Read value is correct:    52381
# Read value is correct:    16022
# Read value is correct:   243731
# Read value is correct:    14349
# Read value is correct:   251475
# Read value is correct:   253291
# Read value is correct:   142037
# Read value is correct:   215554
# Read value is correct:   147118
# Read value is correct:   256285
# Read value is correct:    94927
# Read value is correct:    18723
# Read value is correct:    91402
# Read value is correct:   199370
# Read value is correct:    85052
# Read value is correct:    48626
# Read value is correct:   156042
# Read value is correct:    45889
# Read value is correct:   210136
# Read value is correct:    62328
# Read value is correct:   135817
# Read value is correct:    69099
# Read value is correct:   157110
# Read value is correct:   260550
# Read value is correct:    70574
# Read value is correct:    66236
# Read value is correct:   253226
# Read value is correct:   236043
```

Fig 3. FIFO testbench result

B. Buffer Unit Controller

The buffer unit is responsible for buffering input packets and sending them to one of the neighboring routers. Therefore, a controller is designed to handle handshaking with other buffer units. Figure 4 illustrates the state diagram of the controller. It has three main parts. The first three states are dedicated to receiving a packet and buffering it in the FIFO. The fourth state is for requesting a port, and the state machine does not leave this state until it receives the grant. Finally, in the last three states, the controller attempts to transport the packet to the target buffer unit. The handshaking algorithm is based on sending a request and receiving an acknowledgment. Figure 5 shows the handshaking algorithm.

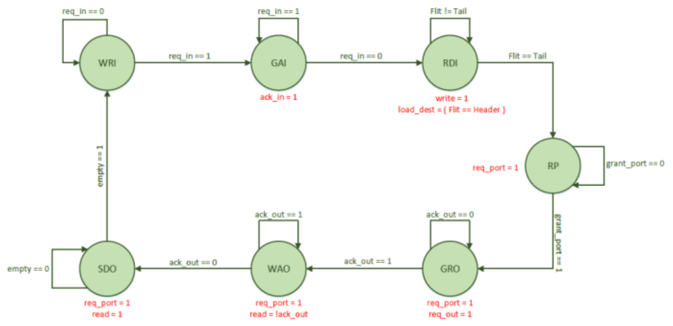


Fig 4. Buffer Unit Controller State diagram

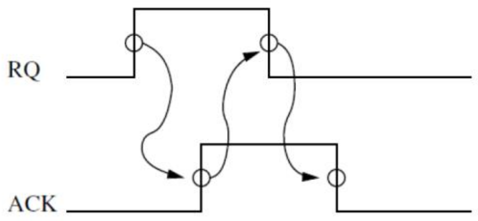


Fig 5. Buffer units handshaking

C. Buffer Unit Top Module

In the top module, the FIFO and control unit are connected to each other. Additionally, a register is added to store the destination of the packet. This is achieved using the 'load_dest' signal, which is generated by the controller whenever the header flit arrives. The RTL view of the buffer unit top module is shown in Figure 6.

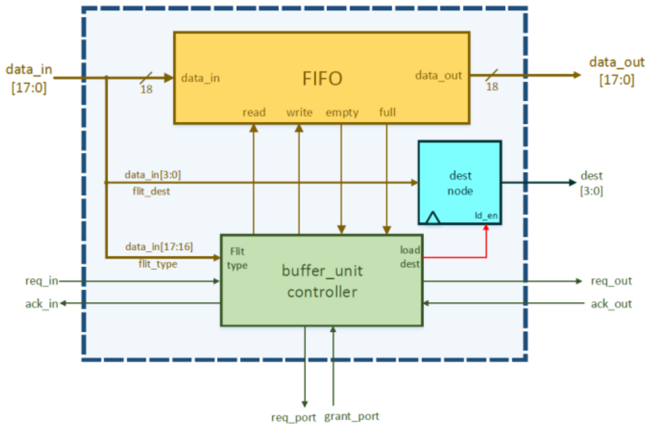


Fig 6. RTL view of the buffer unit

III. Node

After implementing the buffer unit in Verilog, it is necessary to verify this module. To achieve this, we need to write a behavioral node in Verilog that mimics the functionality of a processing unit in a NOC. This unit consists of two initial statements in Verilog. One statement is responsible for generating packets and sending them to the local buffer unit, while the other statement is prepared to receive incoming packets. The number of generated packets can be controlled using a parameter. Additionally, the node ID should be specified by another parameter. Figure 7 shows a schematic of the node.

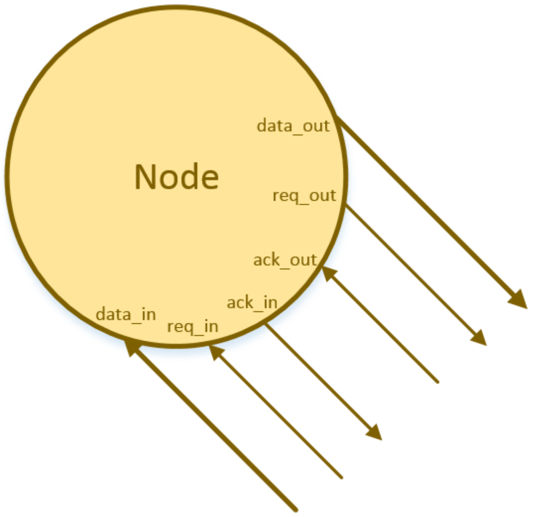
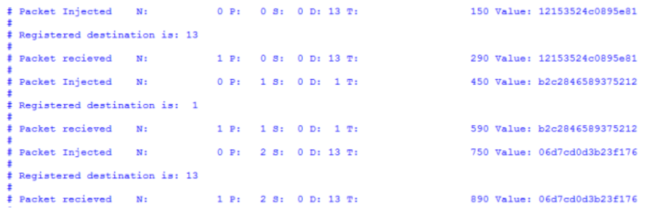


Fig 7. Schematic of node

After implementing the node, we connect two nodes to either side of a buffer unit and issue the grant port signal whenever the buffer unit activates the request port signal. We expect the packets to be generated by one node, buffered in the buffer unit, and then sent to the second node, which receives the packet. Figure 8 shows the simulation results. Each node, after generating a packet, prints some information about the packet. This information includes the generator node ID, packet number, source ID, destination ID, time of injection, and packet value. Similarly, after receiving a packet, the node prints the exact same information about the packet. As shown, three packets injected by node 0 were received by node 1. Moreover, all destinations were registered correctly.



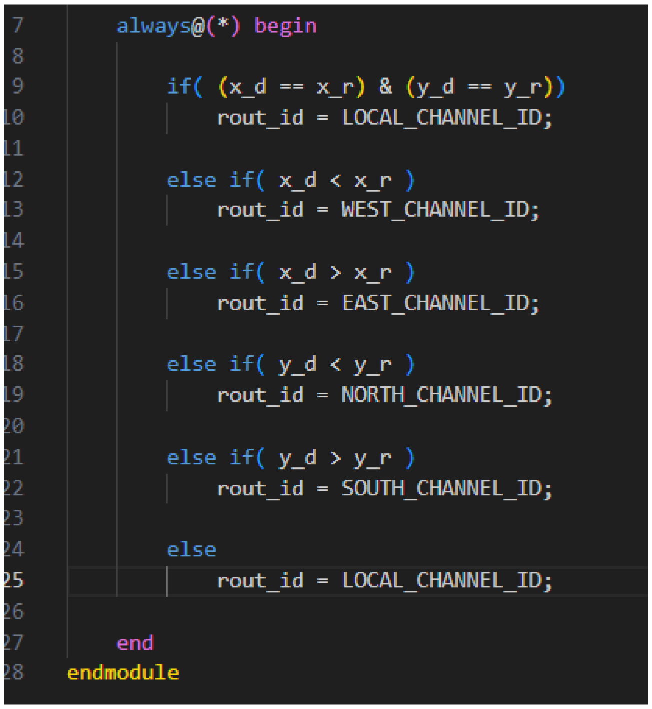
```
# Packet Injected    N:          0 P:   0 S:   0 D: 13 T:                 150 Value: 12153524c0895e81
#
# Registered destination is: 13
#
# Packet recieved    N:          1 P:   0 S:   0 D: 13 T:                 290 Value: 12153524c0895e81
#
# Packet Injected    N:          0 P:   1 S:   0 D:  1 T:                 450 Value: b2c2846589375212
#
# Registered destination is:  1
#
# Packet recieved    N:          1 P:   1 S:   0 D:  1 T:                 590 Value: b2c2846589375212
#
# Packet Injected    N:          0 P:   2 S:   0 D: 13 T:                 750 Value: 06d7cd0d3b23f176
#
# Registered destination is: 13
#
# Packet recieved    N:          1 P:   2 S:   0 D: 13 T:                 890 Value: 06d7cd0d3b23f176
```

Fig 8. Buffer unit simulation result

IV. Switching

A. Routing Unit

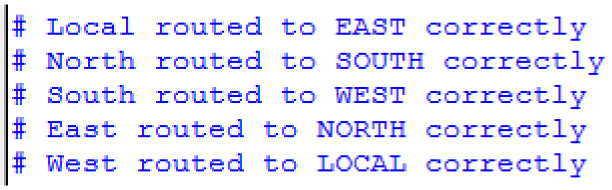
As discussed in previous sections, every buffer unit registers the packet's destination node ID. Therefore, a module is needed to find the proper output port for every packet. This unit uses the 'X-Y' algorithm to route packets. According to this algorithm, each packet first travels horizontally to match the packet's x-coordinate with the destination node's x-coordinate. Then, it travels vertically to reach the router connected to the target node. The algorithm for this unit, implemented in Verilog, is shown in Figure 9.



```
    always@(*) begin

        if( (x_d == x_r) & (y_d == y_r))
            rout_id = LOCAL_CHANNEL_ID;

        else if( x_d < x_r )
            rout_id = WEST_CHANNEL_ID;

        else if( x_d > x_r )
            rout_id = EAST_CHANNEL_ID;

        else if( y_d < y_r )
            rout_id = NORTH_CHANNEL_ID;

        else if( y_d > y_r )
            rout_id = SOUTH_CHANNEL_ID;

        else
            rout_id = LOCAL_CHANNEL_ID;

    end
endmodule
```

Fig 9. Routing algorithm in Verilog

All buffer unit destinations are sent to a module called the routing unit, which generates the proper output channel for each of them. We developed a testbench in Verilog to verify this module. The testbench results are illustrated in Figure 10.



```
# Local routed to EAST correctly
# North routed to SOUTH correctly
# South routed to WEST correctly
# East routed to NORTH correctly
# West routed to LOCAL correctly
```

Fig 10. Routing unit testbench result

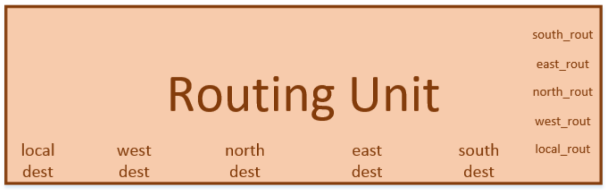


Fig 11. Routing unit schematic

B. Switch Allocator

The switch allocator is a module that allocates the output port to one of the buffer units that claim the port by issuing a request port signal. Each output port has its own switch allocator, which decides which buffer unit can access the output port. Figure 12 shows the RTL view of the switch allocator unit. As illustrated, all buffer units' request port signals are sent to this module. The module compares the route of each buffer unit, generated by the routing unit, with the output channel ID. If the route and channel ID match, the module checks whether the request port signal is on or off using an 'AND' gate and a comparator.

However, as mentioned in previous sections, the local buffer unit has the highest priority, followed by west, north, east, and south in descending order of priority. This means that if one buffer unit is granted the port, another buffer unit with lower priority cannot grant the output port. The priority is implemented using inverters and larger 'AND' gates. The inverted grant signal of a buffer unit is sent to the 'AND' gates of buffer units with lower priority to prevent their outputs from becoming high.

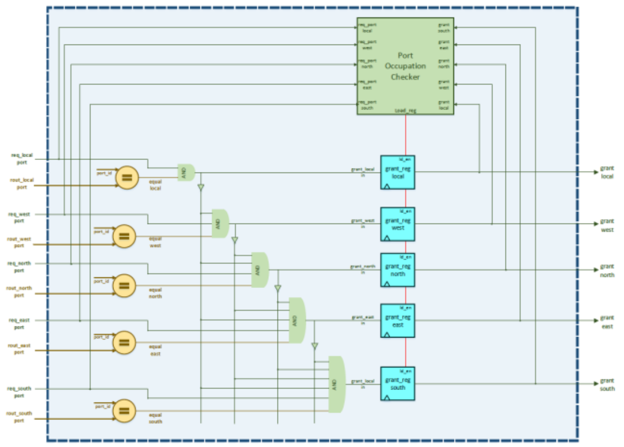


Fig 12. RTL view of switch allocator

Additionally, registers are needed to prevent collisions between two buffer units. For example, assume a buffer unit is granted the port and is sending flits. Meanwhile, another buffer unit with higher priority attempts to grant the port by issuing the request port signal. In such a situation, the switch allocator should maintain the previous state of the grant signals until the first buffer unit finishes its job and deactivates its request port signal. Therefore, registers are necessary to prevent conflicts.

To generate load signals for the registers, a unit called the 'Port Occupation Checker' is developed. The RTL view of this module is shown in Figure 13. It issues the load signal if none of the grant signals are active or if the buffer unit that grants the port deactivates the request port signal.

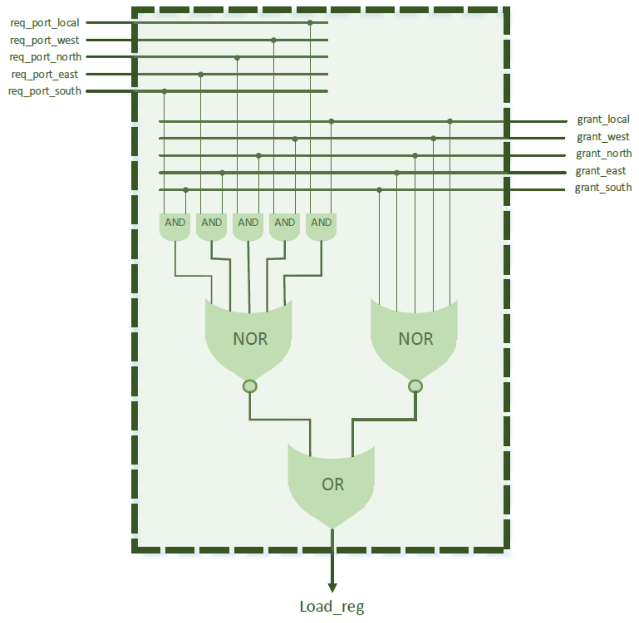


Fig 13. RTL view of port occupation checker

After implementing this unit in Verilog, a testbench was developed to verify the switch allocator module. In this testbench, all request ports are issued simultaneously. After a while, the local request port, which has the highest priority, is deactivated, allowing the other buffer units to grant the port. The other buffer units then deactivate their request ports sequentially. Figure 14 shows the results of this testbench.

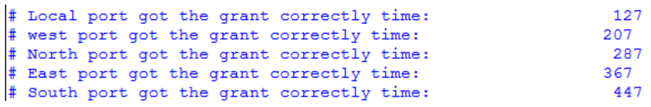

```
# Local port got the grant correctly time:              127
# west port got the grant correctly time:              207
# North port got the grant correctly time:              287
# East port got the grant correctly time:              367
# South port got the grant correctly time:              447
```

Fig 14. Testbench result of switch allocator

C. Switch Port

In a router, a switch unit is necessary to connect each buffer unit to its target output channel. The computer assignment requires us to design a single unit to handle all switching activities. However, our design differs; we use a switch unit for each output channel. All buffer units send their signals to each port switch, which then decides which signal should be selected for the output. The operations of switching and selecting data are implemented using multiplexers, demultiplexers, and the switch allocator discussed in the previous section. Figure 15 illustrates the RTL design of the port switch unit.

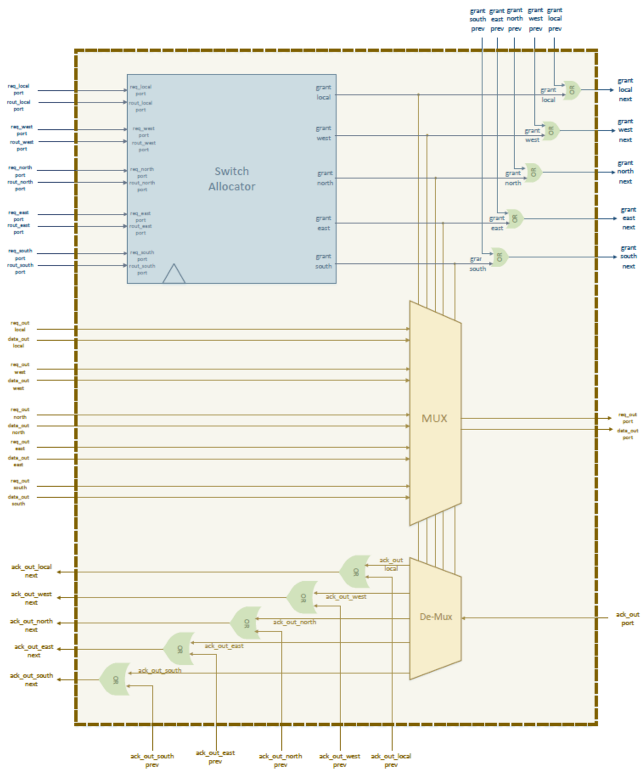
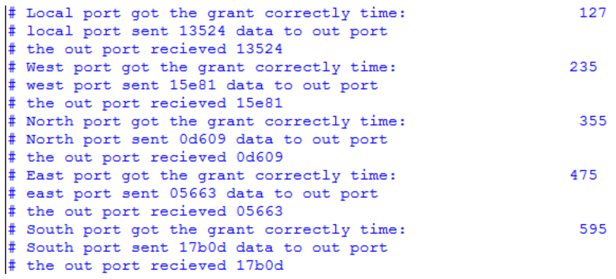


Fig 15. RTL view of switch port unit

As you know, there is only one grant signal for each buffer unit, but there are 5 switch allocators that generate 5 grant signals for each buffer unit. If any one of these five signals is on, then the grant signal for the related buffer unit should become active. To achieve this, a 5-input 'OR' gate can be used for each buffer unit to create the grant signal.

However, in our design, we use multiple 2-input 'OR' gates within the port switch unit to logically combine the input grant signals and the generated grant signals to create the output grant signals. By connecting the output grant signals of one port switch to the input grant signals of the next port switch, we ensure that, at the output of the last switch, we have the grant signals for each buffer unit.

Finally, for verification, a testbench was developed in Verilog similar to the testbench for the switch allocator, with some additional components. All buffer units activate the request port simultaneously. After granting the port one by one, they send data using the dedicated channel. Figure 16 shows the results of the testbench. As shown, the buffer units grant the port one by one, and all sent data and received data are the same.


```
# Local port got the grant correctly time:                127
# local port sent 13524 data to out port
# the out port recieved 13524
# West port got the grant correctly time:                235
# west port sent 15e81 data to out port
# the out port recieved 15e81
# North port got the grant correctly time:                355
# North port sent 0d609 data to out port
# the out port recieved 0d609
# East port got the grant correctly time:                475
# east port sent 05663 data to out port
# the out port recieved 05663
# South port got the grant correctly time:                595
# South port sent 17b0d data to out port
# the out port recieved 17b0d
```
Fig 16. Result of switch port testbench

V. Router

After implementing and verifying all the previous parts, we can now connect all these modules together to create a router. Figure 17 illustrates the design of a router.

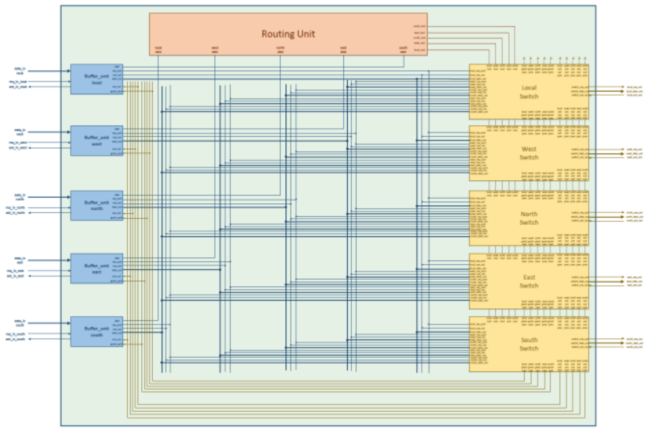
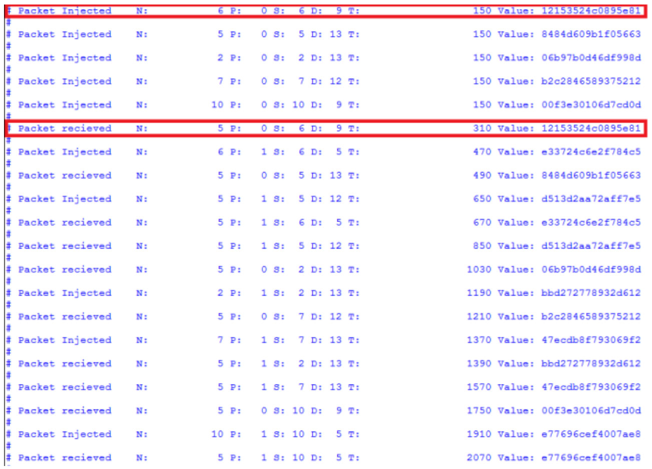


Fig 17. Design of router unit

Now we need to test the router. To achieve this, a testbench should be developed. In this testbench, we connect 5 nodes, developed in Section 3, to a router. Each node generates 2 packets for random destinations, and the router is responsible for routing the packets. The router ID is 6, and the node IDs are 2, 5, 6, 7, and 10, which correspond to the actual router numbers surrounding router 6 in a 4x4 NOC. The result of this testbench is shown in Figure 18. As shown, all packets have been routed correctly based on the 'X-Y' algorithm. For instance, the first packet is generated by node 6 and is scheduled to be sent to node 9. Since node 9 is located west of node 6 in a 4x4 NOC, the packet should be sent to the west output channel and received by node 5. The result matches our expectations exactly.



```
# Packet Injected    N:          6 P:   0 S:  6 D:  9 T:                  150 Value: 12153524c0895e81
#
# Packet Injected    N:          5 P:   0 S:  5 D: 13 T:                  150 Value: 8484d609b1f05663
#
# Packet Injected    N:          2 P:   0 S:  2 D: 13 T:                  150 Value: 06b97b0d46df998d
#
# Packet Injected    N:          7 P:   0 S:  7 D: 12 T:                  150 Value: b2c2846589375212
#
# Packet Injected    N:         10 P:   0 S: 10 D:  9 T:                  150 Value: 00f3e30106d7cd0d
#
# Packet recieved    N:          5 P:   0 S:  6 D:  9 T:                  310 Value: 12153524c0895e81
#
# Packet Injected    N:          6 P:   1 S:  6 D:  5 T:                  470 Value: e33724c6e2f784c5
#
# Packet recieved    N:          5 P:   0 S:  5 D: 13 T:                  490 Value: 8484d609b1f05663
#
# Packet Injected    N:          5 P:   1 S:  5 D: 12 T:                  650 Value: d513d2aa72aff7e5
#
# Packet recieved    N:          5 P:   1 S:  6 D:  5 T:                  670 Value: e33724c6e2f784c5
#
# Packet recieved    N:          5 P:   1 S:  5 D: 12 T:                  850 Value: d513d2aa72aff7e5
#
# Packet recieved    N:          5 P:   0 S:  2 D: 13 T:                 1030 Value: 06b97b0d46df998d
#
# Packet Injected    N:          2 P:   1 S:  2 D: 13 T:                 1190 Value: bbd272778932d612
#
# Packet recieved    N:          5 P:   0 S:  7 D: 12 T:                 1210 Value: b2c2846589375212
#
# Packet Injected    N:          7 P:   1 S:  7 D: 13 T:                 1370 Value: 47ecdb8f793069f2
#
# Packet recieved    N:          5 P:   1 S:  2 D: 13 T:                 1390 Value: bbd272778932d612
#
# Packet recieved    N:          5 P:   1 S:  7 D: 13 T:                 1570 Value: 47ecdb8f793069f2
#
# Packet recieved    N:          5 P:   0 S: 10 D:  9 T:                 1750 Value: 00f3e30106d7cd0d
#
# Packet Injected    N:         10 P:   1 S: 10 D:  5 T:                 1910 Value: e77696cef4007ae8
#
# Packet recieved    N:          5 P:   1 S: 10 D:  5 T:                 2070 Value: e77696cef4007ae8
```

Fig 18. Router testbench result

VI. 4x4 Network on-Chip

A. Design

To design a 4x4 NOC, it is necessary to connect 16 routers together both horizontally and vertically. Each router has a node connected to the local channel, and the four other channels are used for cascading routers together. Figure 19 shows the design of a 4x4 NOC.

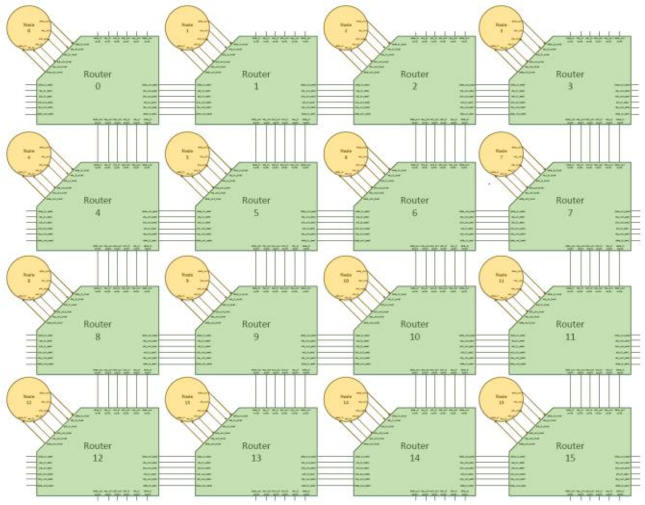
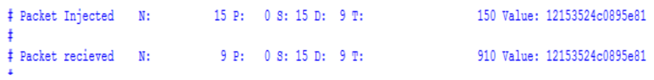


Fig 19. 4x4 NOC design

To implement this connection in Verilog, a generate statement should be utilized. The most important point in connecting the ports is that the output channel of one router should be connected to the input channel of the neighboring router. For example, the 'data_in' port of the west channel of router number 6 should be connected to the 'data_out' port of the east channel of router number 5.

B. One Injection Verification

After developing the 4x4 NOC in Verilog, we need to test its functionality under different scenarios. First, we created a testbench to inject a single packet to a random destination and check whether it reaches the target node correctly. This is possible due to the packet number parameter embedded in the node Verilog code. If this number is zero or negative, the node does not generate any packets. We subtract each router ID from 14 and use this as the node parameter. In this configuration, only node 15 generates a packet while the other nodes wait for packets to arrive. Figure 20 shows the testbench results, demonstrating that the packets arrived at the correct node.

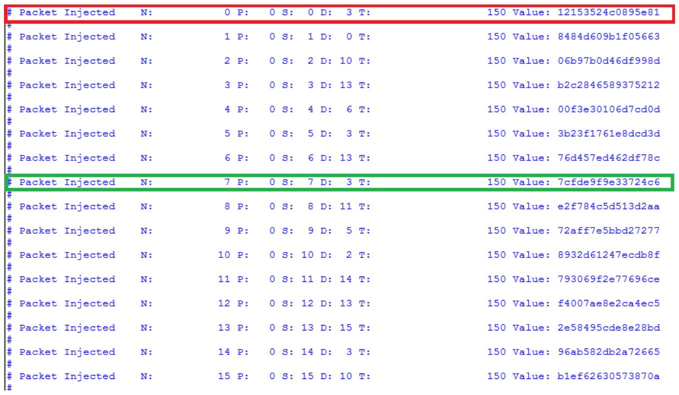


```
# Packet Injected    N:         15 P:   0 S: 15 D:  9 T:                  150 Value: 12153524c0895e81
#
# Packet recieved    N:          9 P:   0 S: 15 D:  9 T:                  910 Value: 12153524c0895e81
```

Fig 20. One injection testbench result

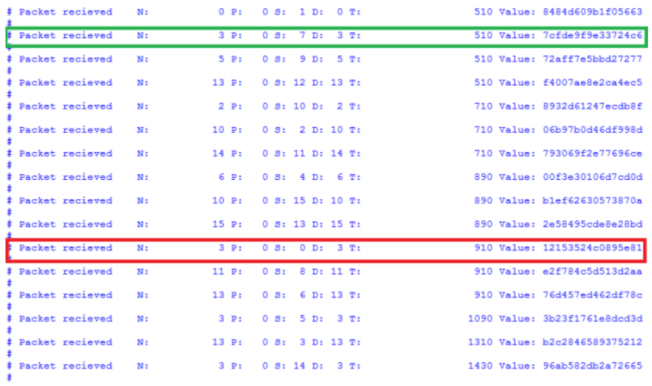
C. One Injection per Node Verification

In this scenario, each node generates only one packet for a random destination. Figure 21 shows the injected packets, and Figure 22 illustrates the information about the received packets. We can see that each packet arrived at the correct node, but at different times, and the order of packet generation does not correspond to the order of reception. For example, node 0 generates a packet for node 3, which is located at the end of the row. Interestingly, node 7 also generates a packet for node 3, which is located directly above node 7, at the same time. The results show that the packet generated by node 7 arrived at node 3 at 510 ns, while the packet generated by node 0 arrived at 910 ns. This trend is consistent for all sent packets.



```
# Packet Injected    N:          0 P:   0 S:  0 D:  3 T:                  150 Value: 12153524c0895e81
#
# Packet Injected    N:          1 P:   0 S:  1 D:  0 T:                  150 Value: 8484d609b1f05663
#
# Packet Injected    N:          2 P:   0 S:  2 D: 10 T:                  150 Value: 06b97b0d46df998d
#
# Packet Injected    N:          3 P:   0 S:  3 D: 13 T:                  150 Value: b2c2846589375212
#
# Packet Injected    N:          4 P:   0 S:  4 D:  6 T:                  150 Value: 00f3e30106d7cd0d
#
# Packet Injected    N:          5 P:   0 S:  5 D:  3 T:                  150 Value: 3b23f1761e8dcd3d
#
# Packet Injected    N:          6 P:   0 S:  6 D: 13 T:                  150 Value: 76d457ed462df78c
#
# Packet Injected    N:          7 P:   0 S:  7 D:  3 T:                  150 Value: 7cfde9f9e33724c6
#
# Packet Injected    N:          8 P:   0 S:  8 D: 11 T:                  150 Value: e2f784c5d513d2aa
#
# Packet Injected    N:          9 P:   0 S:  9 D:  5 T:                  150 Value: 72aff7e5bbd27277
#
# Packet Injected    N:         10 P:   0 S: 10 D:  2 T:                  150 Value: 8932d61247ecdb8f
#
# Packet Injected    N:         11 P:   0 S: 11 D: 14 T:                  150 Value: 793069f2e77696ce
#
# Packet Injected    N:         12 P:   0 S: 12 D: 13 T:                  150 Value: f4007ae8e2ca4ec5
#
# Packet Injected    N:         13 P:   0 S: 13 D: 15 T:                  150 Value: 2e58495cde8e28bd
#
# Packet Injected    N:         14 P:   0 S: 14 D:  3 T:                  150 Value: 96ab582db2a72665
#
# Packet Injected    N:         15 P:   0 S: 15 D: 10 T:                  150 Value: b1ef62630573870a
#
```

Fig 21. Injected packets information

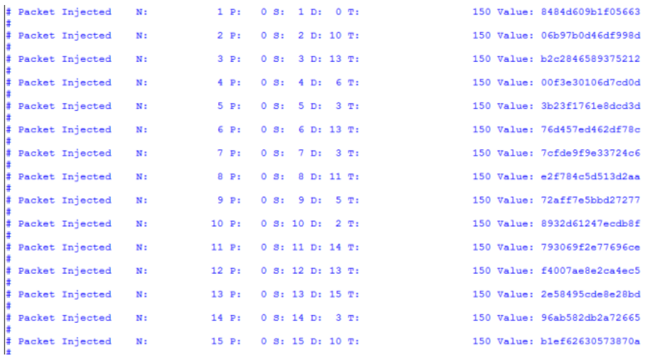


```
# Packet recieved    N:          0 P:   0 S:  1 D:  0 T:                  510 Value: 8484d609b1f05663
#
# Packet recieved    N:          3 P:   0 S:  7 D:  3 T:                  510 Value: 7cfde9f9e33724c6
#
# Packet recieved    N:          5 P:   0 S:  9 D:  5 T:                  510 Value: 72aff7e5bbd27277
#
# Packet recieved    N:         13 P:   0 S: 12 D: 13 T:                  510 Value: f4007ae8e2ca4ec5
#
# Packet recieved    N:          2 P:   0 S: 10 D:  2 T:                  710 Value: 8932d61247ecdb8f
#
# Packet recieved    N:         10 P:   0 S:  2 D: 10 T:                  710 Value: 06b97b0d46df998d
#
# Packet recieved    N:         14 P:   0 S: 11 D: 14 T:                  710 Value: 793069f2e77696ce
#
# Packet recieved    N:          6 P:   0 S:  4 D:  6 T:                  890 Value: 00f3e30106d7cd0d
#
# Packet recieved    N:         10 P:   0 S: 15 D: 10 T:                  890 Value: b1ef62630573870a
#
# Packet recieved    N:         15 P:   0 S: 13 D: 15 T:                  890 Value: 2e58495cde8e28bd
#
# Packet recieved    N:          3 P:   0 S:  0 D:  3 T:                  910 Value: 12153524c0895e81
#
# Packet recieved    N:         11 P:   0 S:  8 D: 11 T:                  910 Value: e2f784c5d513d2aa
#
# Packet recieved    N:         13 P:   0 S:  6 D: 13 T:                  910 Value: 76d457ed462df78c
#
# Packet recieved    N:          3 P:   0 S:  5 D:  3 T:                 1090 Value: 3b23f1761e8dcd3d
#
# Packet recieved    N:         13 P:   0 S:  3 D: 13 T:                 1310 Value: b2c2846589375212
#
# Packet recieved    N:          3 P:   0 S: 14 D:  3 T:                 1430 Value: 96ab582db2a72665
#
```

Fig 22. Received packets information

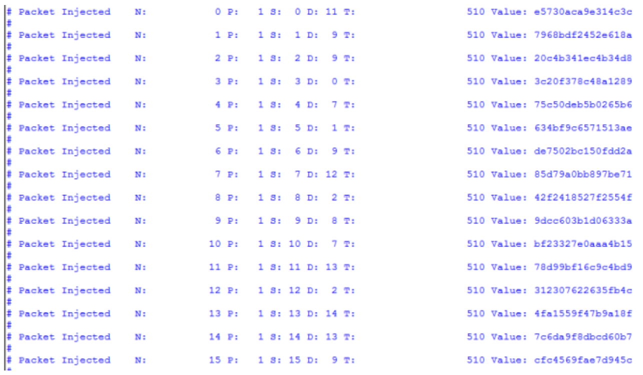
D. Two Injection per Node Verification

In this scenario, each node generates two packets. However, verification becomes more difficult because the total number of packets reaches 32, making it challenging to manually check whether each packet arrived at the correct node. Figures 23, 24, 25, and 26 show the results.



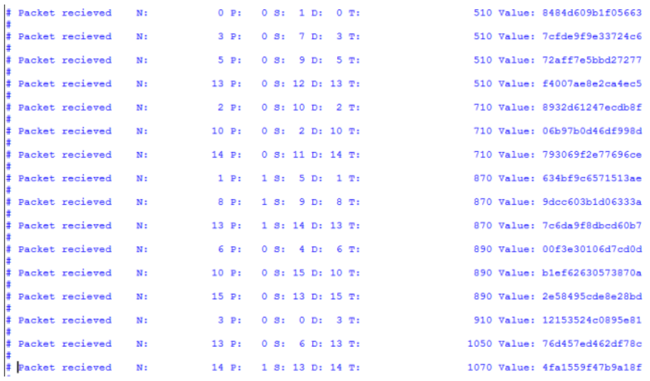
```
# Packet Injected    N:          1 P:   0 S:  1 D:  0 T:               150 Value: 8484d609b1f05663
#
# Packet Injected    N:          2 P:   0 S:  2 D: 10 T:               150 Value: 06b97b0d46df998d
#
# Packet Injected    N:          3 P:   0 S:  3 D: 13 T:               150 Value: b2c2846589375212
#
# Packet Injected    N:          4 P:   0 S:  4 D:  6 T:               150 Value: 00f3e30106d7cd0d
#
# Packet Injected    N:          5 P:   0 S:  5 D:  3 T:               150 Value: 3b23f1761e8dcd3d
#
# Packet Injected    N:          6 P:   0 S:  6 D: 13 T:               150 Value: 76d457ed462df78c
#
# Packet Injected    N:          7 P:   0 S:  7 D:  3 T:               150 Value: 7cfde9f9e33724c6
#
# Packet Injected    N:          8 P:   0 S:  8 D: 11 T:               150 Value: e2f784c5d513d2aa
#
# Packet Injected    N:          9 P:   0 S:  9 D:  5 T:               150 Value: 72aff7e5bbd27277
#
# Packet Injected    N:         10 P:   0 S: 10 D:  2 T:               150 Value: 8932d61247ecdb8f
#
# Packet Injected    N:         11 P:   0 S: 11 D: 14 T:               150 Value: 793069f2e77696ce
#
# Packet Injected    N:         12 P:   0 S: 12 D: 13 T:               150 Value: f4007ae8e2ca4ec5
#
# Packet Injected    N:         13 P:   0 S: 13 D: 15 T:               150 Value: 2e58495cde8e28bd
#
# Packet Injected    N:         14 P:   0 S: 14 D:  3 T:               150 Value: 96ab582db2a72665
#
# Packet Injected    N:         15 P:   0 S: 15 D: 10 T:               150 Value: b1ef62630573870a
#
```

Fig 23. First group of packets injected at 150 ns



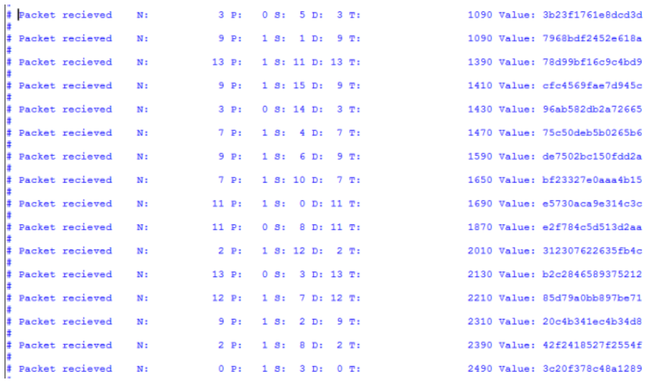
```
# Packet Injected    N:          0 P:   1 S:  0 D: 11 T:               510 Value: e5730aca9e314c3c
#
# Packet Injected    N:          1 P:   1 S:  1 D:  9 T:               510 Value: 7968bdf2452e618a
#
# Packet Injected    N:          2 P:   1 S:  2 D:  9 T:               510 Value: 20c4b341ec4b34d8
#
# Packet Injected    N:          3 P:   1 S:  3 D:  0 T:               510 Value: 3c20f378c48a1289
#
# Packet Injected    N:          4 P:   1 S:  4 D:  7 T:               510 Value: 75c50deb5b0265b6
#
# Packet Injected    N:          5 P:   1 S:  5 D:  1 T:               510 Value: 634bf9c6571513ae
#
# Packet Injected    N:          6 P:   1 S:  6 D:  9 T:               510 Value: de7502bc150fdd2a
#
# Packet Injected    N:          7 P:   1 S:  7 D: 12 T:               510 Value: 85d79a0bb897be71
#
# Packet Injected    N:          8 P:   1 S:  8 D:  2 T:               510 Value: 42f2418527f2554f
#
# Packet Injected    N:          9 P:   1 S:  9 D:  8 T:               510 Value: 9dcc603b1d06333a
#
# Packet Injected    N:         10 P:   1 S: 10 D:  7 T:               510 Value: bf23327e0aaa4b15
#
# Packet Injected    N:         11 P:   1 S: 11 D: 13 T:               510 Value: 78d99bf16c9c4bd9
#
# Packet Injected    N:         12 P:   1 S: 12 D:  2 T:               510 Value: 312307622635fb4c
#
# Packet Injected    N:         13 P:   1 S: 13 D: 14 T:               510 Value: 4fa1559f47b9a18f
#
# Packet Injected    N:         14 P:   1 S: 14 D: 13 T:               510 Value: 7c6da9f8dbcd60b7
#
# Packet Injected    N:         15 P:   1 S: 15 D:  9 T:               510 Value: cfc4569fae7d945c
```

Fig 24. second group of packets injected at 150 ns



```
# Packet recieved    N:          0 P:   0 S:  1 D:  0 T:               510 Value: 8484d609b1f05663
#
# Packet recieved    N:          3 P:   0 S:  7 D:  3 T:               510 Value: 7cfde9f9e33724c6
#
# Packet recieved    N:          5 P:   0 S:  9 D:  5 T:               510 Value: 72aff7e5bbd27277
#
# Packet recieved    N:         13 P:   0 S: 12 D: 13 T:               510 Value: f4007ae8e2ca4ec5
#
# Packet recieved    N:          2 P:   0 S: 10 D:  2 T:               710 Value: 8932d61247ecdb8f
#
# Packet recieved    N:         10 P:   0 S:  2 D: 10 T:               710 Value: 06b97b0d46df998d
#
# Packet recieved    N:         14 P:   0 S: 11 D: 14 T:               710 Value: 793069f2e77696ce
#
# Packet recieved    N:          1 P:   1 S:  5 D:  1 T:               870 Value: 634bf9c6571513ae
#
# Packet recieved    N:          8 P:   1 S:  9 D:  8 T:               870 Value: 9dcc603b1d06333a
#
# Packet recieved    N:         13 P:   1 S: 14 D: 13 T:               870 Value: 7c6da9f8dbcd60b7
#
# Packet recieved    N:          6 P:   0 S:  4 D:  6 T:               890 Value: 00f3e30106d7cd0d
#
# Packet recieved    N:         10 P:   0 S: 15 D: 10 T:               890 Value: b1ef62630573870a
#
# Packet recieved    N:         15 P:   0 S: 13 D: 15 T:               890 Value: 2e58495cde8e28bd
#
# Packet recieved    N:          3 P:   0 S:  0 D:  3 T:               910 Value: 12153524c0895e81
#
# Packet recieved    N:         13 P:   0 S:  6 D: 13 T:              1050 Value: 76d457ed462df78c
#
# Packet recieved    N:         14 P:   1 S: 13 D: 14 T:              1070 Value: 4fa1559f47b9a18f
```

Fig 25. Packets arrived in destinations in different times



```
# Packet recieved    N:          3 P:   0 S:  5 D:  3 T:              1090 Value: 3b23f1761e8dcd3d
#
# Packet recieved    N:          9 P:   1 S:  1 D:  9 T:              1090 Value: 7968bdf2452e618a
#
# Packet recieved    N:         13 P:   1 S: 11 D: 13 T:              1390 Value: 78d99bf16c9c4bd9
#
# Packet recieved    N:          9 P:   1 S: 15 D:  9 T:              1410 Value: cfc4569fae7d945c
#
# Packet recieved    N:          3 P:   0 S: 14 D:  3 T:              1430 Value: 96ab582db2a72665
#
# Packet recieved    N:          7 P:   1 S:  4 D:  7 T:              1470 Value: 75c50deb5b0265b6
#
# Packet recieved    N:          9 P:   1 S:  6 D:  9 T:              1590 Value: de7502bc150fdd2a
#
# Packet recieved    N:          7 P:   1 S: 10 D:  7 T:              1650 Value: bf23327e0aaa4b15
#
# Packet recieved    N:         11 P:   1 S:  0 D: 11 T:              1690 Value: e5730aca9e314c3c
#
# Packet recieved    N:         11 P:   0 S:  8 D: 11 T:              1870 Value: e2f784c5d513d2aa
#
# Packet recieved    N:          2 P:   1 S: 12 D:  2 T:              2010 Value: 312307622635fb4c
#
# Packet recieved    N:         13 P:   0 S:  3 D: 13 T:              2130 Value: b2c2846589375212
#
# Packet recieved    N:         12 P:   1 S:  7 D: 12 T:              2210 Value: 85d79a0bb897be71
#
# Packet recieved    N:          9 P:   1 S:  2 D:  9 T:              2310 Value: 20c4b341ec4b34d8
#
# Packet recieved    N:          2 P:   1 S:  8 D:  2 T:              2390 Value: 42f2418527f2554f
#
# Packet recieved    N:          0 P:   1 S:  3 D:  0 T:              2490 Value: 3c20f378c48a1289
```

Fig 26. Packets arrived in destinations in different times

VII. Uploaded Files Description

The uploaded files include two folders and a report file. If any part of the report contains unclear design pictures or lacks detail, you can refer to the Visio and PDF files located in the "Illustrations" folder, which contain detailed designs. Additionally, all Verilog files are located in the "Verilog Codes" folder, organized into separate subfolders. Finally, the Word version of the report may contains more details than the Pdf version.

IIX. Conclusion

In this computer assignment, we learned about the concept of network on-chip (NOC) and implemented a simple version of NOC using Verilog. Additionally, we verified each module of the design by developing testbenches. Finally, we verified a 4x4 NOC under different scenarios and can confidently claim that the NOC is working correctly.