Synthesis Configurable Virtual Channel Router

Varun Saxena
vsaxena36@gatech.edu
Georgia Institute of Technology

Sandilya Balemarthy
vbalemarthy3@gatech.edu
Georgia Institute of Technology

ABSTRACT

In the current day of Chip Multi-Processors (CMPs), the need for effective communication between the different nodes has become very important. If not designed efficiently, this could become the system bottleneck. Various advances have been made in the Network-on-Chip (NoCs) design to handle this. The router micro-architecture is one of the main parts of the NoC. Virtual Channel based routers overcome the limitations of various other approaches like circuit switching, store and forward, virtual cut-through, wormhole etc. In this project, we develop this router to understand the internal components of such an approach and use various techniques to optimize the RTL implementation. We develop different arbiter and allocator approaches to handle fairness, speed, and area optimizations. We develop switches to allow context-based usage of these components in the Virtual Channel (VC) and Switch allocator (SW) stages. Additionally, we develop a credit-based system for backpressure signalling. Finally, we synthesize the design to check for viability and integrate the router into a 5x5 Torus topology for full network testing. We also evaluate these approaches for functionality, performance, and synthesis compatibility.

KEYWORDS

Network-on-Chip, Virtual Channel, Arbiter, Allocator

1 INTRODUCTION

Our objective is to design a configurable virtual channel router. We adopt the standard router microarchitecture comprising of buffer management, route compute, VC allocation, Switch allocation and Crossbar switch modules. The arbiter and allocators used within the VC and Switch allocation stages are controlled via a one stop configuration file to encapsulate all design choices. The available arbiter configurations are Round-robin and Matrix arbiter. The available allocator configurations are: Separable and Wavefront allocator. In addition to the configurations and router architecture, we have also accommodated credit signaling to upstream and downstream routers.

2 DESIGN LIBRARY

One of the main components that we provide is an option for is the allocator stages. The allocator is one of the main components which decide the fairness of allocation to the traffic available in the input buffers. We develop 2 types of arbiters[4] and integrate them into 2 types of allocators:

Spring'22, April 2022, Atlanta, GA, USA

2.1 Acyclic Round-robin arbiter

The round-robin arbiter (shown in Fig. 1 provides grants to the input requests in a cyclic fashion. It contains a round-robin logic-cell (R) which decides its grant signal using the carry from the previous logic-cell, priority signal and input request. The carry signal denotes whether any of the previous logic-cells in the cyclic search have their grant signals asserts. This ensures that, if one grant is asserted, then any of the downstream grants cannot be asserts. The priority signal is rotating signal generated from the (P) block, which changes based on the output granted in the previous cycle. This ensures that each request is fairly considered. Further on, to ensure synthesizability, we remove the combinatorial loop by breaking the priority signal wraparound and using an additional set of fixed-priority cells. Then the final grant is decided based on the decision made from the 2 set of logic-cells.

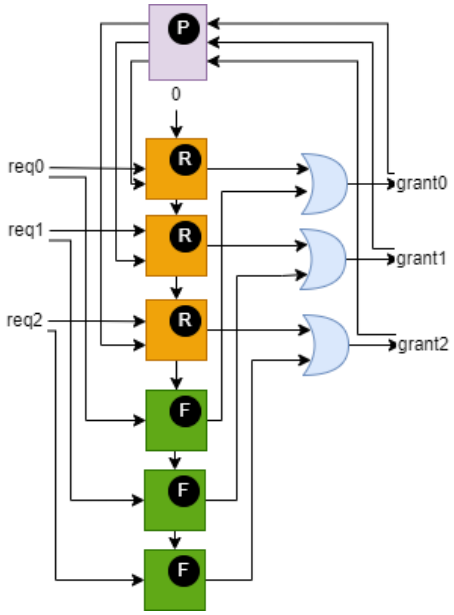

Figure 1: Acyclic Round Robin Arbiter

2.2 Matrix arbiter

The Matrix arbiter (shown in Fig. 2) is an intelligent round-robin arbiter that continuously tracks pairwise precedence between all input requests and updates them in response to each grant, therefore, implementing a least recently served policy. At the next request grant, the arbiter refers to the pairwise precedence and assigns the grant to the least recently served input request. The design of the matrix arbiter consists of an array of requests and an array of grants, along with the clock and reset signal. Internally, the design

contains a 2-D register array that records the pairwise precedence from the last input request grant. The updated priority matrix is fed to a combinational circuit that drives disable signals corresponding to each input request port. The final grant is based on a combination of input requests and the corresponding disable signals. Once an input request 'm' is granted, the priority matrix is updated with all elements in row 'm' being set to 0 and all elements in column 'm' being set to 1. This updates the priority matrix as per the recent grant decision.

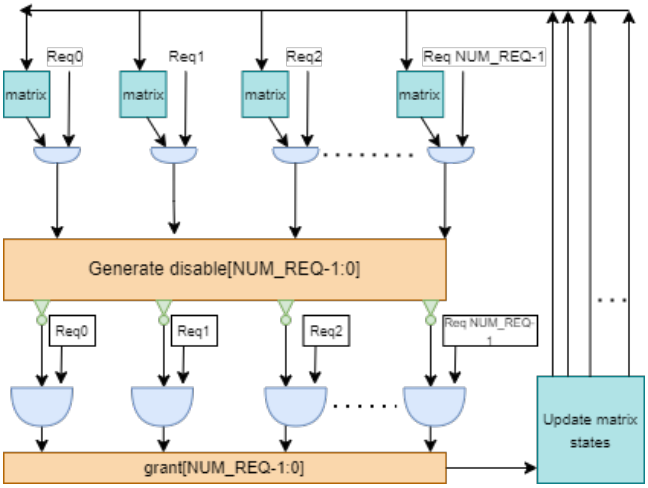

Figure 2: Matrix Arbiter

2.3 Separable allocator

An allocator is used to allocate 'm' requests to 'n' resources and the number of generated grants represents a metric for matching quality. We have implemented an Input-first Separable allocator (shown in Fig. 3) that generates a matching by splitting the allocation process into two stages: The first stage uses an arbiter to select a resource for each requestor with a valid request. The second stage arbitrates for a single requestor for each resource. We use a top-level arbiter wrapper module that selects one of the available arbiters (matrix or wavefront) based on the configuration set within a configuration file. The results of the first stage arbiter are stored in an intermediate array that is re-arranged and later fed through another stage of arbitration to select one requestor per resource.

2.4 Acyclic Wavefront allocator

This allocator (shown in Fig. 4) works on the concept that the requests along the diagonal of the 2D matrix of requests and resources will lead to maximum fairness. This diagonal is then cyclically moved across the complete matrix to choose the diagonal which provides the requires grants. The wavefront allocator itself is a matrix of logic-cells with each logic-cell using the priority signal, input request and carry from the x and y direction to decide the further carry signals and grants. The priority signal is connected in the diagonal fashion by extending the priority signal used in the round-robin arbiter. Further on, to handle the cyclic priority check, we implement a rotation mechanism on the input and output so

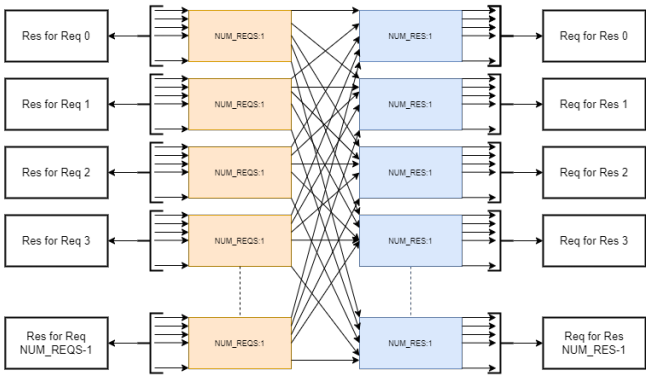

Figure 3: Seprable Allocator

that the diagonal being checked can be moved without with respect to the input and output vectors, without the need of either changing the connections or having duplicate hardware. The priority signal is also updated using a round-robin arbiter. This reduces the fairness of the circuit but reduces the area consumption as duplication scales the area by 2^N, where N is the number of input requests.

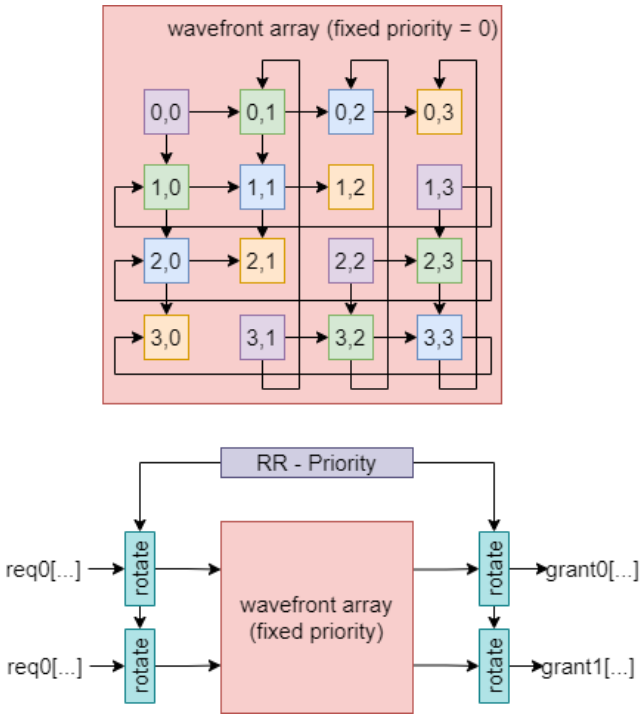

Figure 4: Acyclic Wavefront Allocator

2.5 Additional components

Various other generic and/or custom components were designed to support the system integration such as the arbiter top and allocator top which auto-switches between the required arbiter and allocator based on values set in a top level definition file. Other implemented

blocks are parameterized encoder/decoder blocks, pipe line registers, priority encoder, etc. It also contains the FIFO module which is a very important module to implement the buffers for the VCs.

3 CURRENT DESIGN

We have designed a 5 stage router with buffer divided in terms of Virtual Channels (VCs) and different ports. We have parameterized each component so that the design can be scalable and each parameter can be modified during synthesis such as number of VCs, number of ports, etc. The complete design is pipelined to ensure synchronization and get higher throughput. The Fig. 5 shows the design.

Flit Format The flit is of size FLIT_WIDTH (configured in the top define file). It is divided into 3 parts: VC ID, Dest ID and Data. The VC ID denotes the immediate downstream routers VC the flit will enter (based on VC allocation). The Dest ID is the router ID of the destination router. The "data" is the flit data content. The sizes of the VC ID and Dest ID are set to clog2(VC per port) and clog2(routers in network).

3.1 Arbiter top

We develop a wrapper module B which acts as an interface to the selected arbiter. It allows selecting between the round-robin and matrix arbiter based on configurations set in the top design file. This module allows standarized interface to both types of arbiters for ease of use.

3.2 Allocator top

A similar allocator top A has been developed as an interface to choose between the Separable allocator and Acyclic Wavefront allocator.

3.3 VC/Buffer Implementation

Buffer are designed using a set of FIFOs. The number of FIFOs are configured based on the number of ports and virtual channels which are set in the top configuration file. The buffers 1 are categorized based on the ports and Virtual Channels. Each buffer contains a valid flag and empty signal to denote the status of the head of the FIFO. The signals are used to drive the buffer read, buffer write, etc. stages.

3.4 Buffer write

In the buffer write 2 stage uses the VC ID for each flit to decide a write index. Then it uses this index to write the incoming flit data into the VC and update the required status signals.

3.5 Route compute

This stage 3 computes the route for the flit in each of the VCs in each port. Currently, in the design we assume 1 flit packet, so each flit contains the destination information. Using this information, it uses XY routing to decide the output port. Additionally, this stage uses blocks like a one-hot encoder to convert the port mapping into direction information. Throughout the design we use port 0 to denote "local". For the current project, we are using X-Y routing in a Mesh which considers 4 ports.

3.6 VC Availability

The VC Availability block 4 is responsible for computing the available output VCs and the upstream router increment signals for the 4 input ports. It obtains the destination ports for each input VC from the route compute block, downstream credit signals from the downstream routers and the last cycle's output ports allocated from the switch allocation stage. These are used to updated the credits for each output VC and compute the new output VC availability mask list that is used by the VC Allocation stage to allocate a valid available output VC. Apart from the new VC availability vector, this block also updates the upstream credit signals once switch allocation occurs for a flit. This increment signal is used by the upstream routers to understand the updated VC availability of current router.

3.7 VC Allocation

The VC allocator 5 takes in the destination ports from the route compute module. The destination port for each input VC is represented in the form of a 2-D unpacked array. The VC allocator also takes in the VC availability computed by a separate module. The VC availability is a packed array that denotes whether an output VC is available or is currently being used. The VC allocator then performs a bitwise AND operation, thereby narrowing down the requests to only the possible requests based on the current VC availability. This result is then fed to the allocator which is the configurable top level module that can instantiate either a separable allocator or a wavefront allocator based on the user configurations set from a configuration file. The outputs of the allocator give us the allocated output VCs.

3.8 Switch Allocation

The SW allocation 7 first converts the vc-grant matrix it receives from the VC allocation to convert it into a switch request matrix. For this conversion it again uses the top configuration of port and VC count of its computation, ensuring scalability for higher number of VCs and ports. This converts the $[(no.ofVC)x(no.ofport)]^2$ matrix into a $[no.ofport]^2$ matrix in the module 6. Then it uses the A module directly to convert the request matrix into a grant matrix such that if multiple input ports request for the same output port, then it chooses one of them.

3.9 Buffer read

This stage 8 then uses the processed information from VC and SW allocation stages to compute the access index for each port and VC. The computed indices are then used to pop the data from the buffers. During the read operation it orders the data based on the output port direction so that the port mapping can be done using indices rather than hard-coded directions. Additionally, it also embeds the VC ID into the flit.

3.10 Switch Traversal

This uses a crossbar 9 built using MUX operations to connect the ordered data from the buffer read stage and connect it to the required output port based on the direction the data needs to be dispatched in. This stage requires remapping of the switch allocation

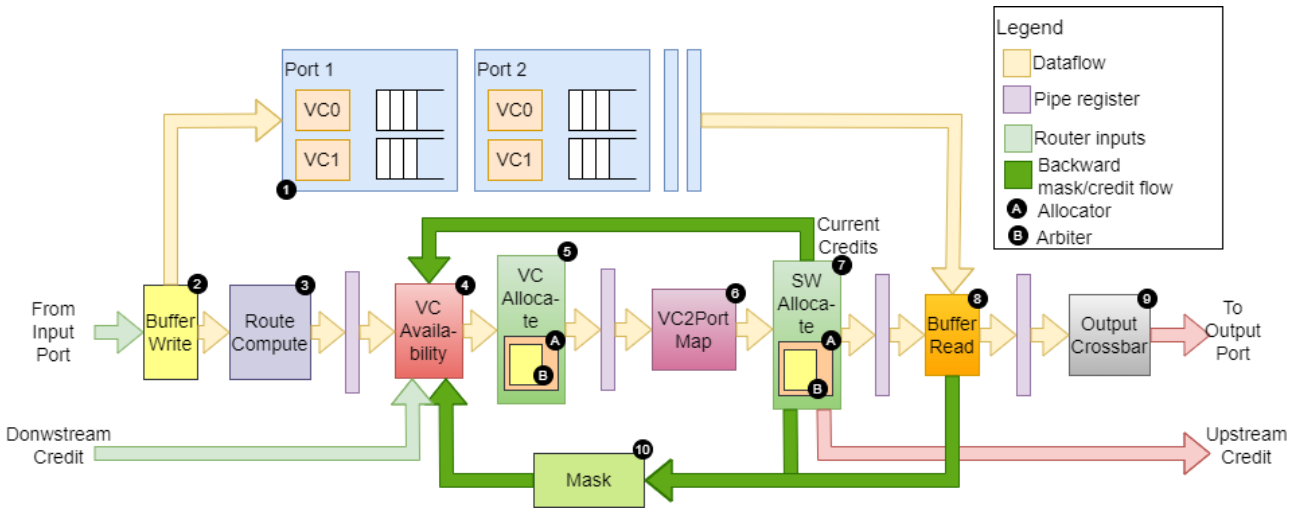

Figure 5: Router Microarchitecture

Figure 6: Flit Format

matrix to ensure that the crossbar structure can use the input port to output direction and decide the required output port connection.

4 DESIGN DECISIONS

Various techniques were used to ensure scalability and add additional features.

4.1 Credit System

The credit system is a part of the VC availability block. It maintains the credits corresponding to each output VC. A credit denotes the available buffer slots at the output VC. The VC availability block sends an upstream router signal to the input VC whose flit has passed through switch allocation. It also decrements the credits corresponding to the output VC upon switch allocation. Once the credits of any output VC reach the 'ROUND_TRIP threshold set with the configuration file, the VC availability vector is updated to prevent the re-allocation of the output VC.

4.2 Configuration file

A top level file contains all control parameters and defines to control aspects of the router like the number of ports, number of VCs, flit data width, direction encoding, arbiter selection, allocator selection, etc.

4.3 Adding Pipelines

To increase the clock frequency and make the design more immune to scaling issues when the number of ports and VCs are increased, we add pipelines to divide the router into sub-stages. The makes the clock frequency dependent on the max latency of the sub-stages as opposed to the end-to-end latency from the input to switch traversal.

4.4 Mask operation

One issue due to adding pipelines is synchronizations issues. The happens as the data is popped from the VC buffer in the buffer read stage, so it takes 3 cycles from the first time the data in sent into the pipelines to the input changing. Due to this, the same data is pushed into the pipeline again and again for 3 cycles which leads to garbage data which causes extensive pop operation due to wrong data propagation. One solution for this is to stall the pipeline for 3 cycles. But, this causes cycle wastage even for ports which would not have been used by the data in the pipeline. So we create mask signals from the downstream stages to mask the "in-use" ports and block these ports from being used while allowing other ports to be used.

4.5 Plug & Play Configurable Control

The Arbiter and Allocator selection throughout the router top module can be controlled via a *VR_define.vh* file. This configuration file consists of parameters for Arbiter, Allocator, Flit data width, Encoding for each port direction, the port assigned to the Network Interface Card *NIC* and the Round trip latency used within the credit system. This router system can be used with any topology by setting the right configurations in the *VR_define.vh* file and updating the *Route Compute Unit* (3) to match the routing algorithm with the topology. One can then instantiate the router into the topology and send and receive flits via the cores of different router nodes. A sample configuration file is shown in Fig. 7

4.6 Scalable Topology

We have constructed a Torus topology module that is scalable in terms of number of rows, columns and number of VCs per router.

The construction of the router connections is shown in Fig. 8. The wiring is auto-configured to connect the North, South, East and West port of any router in the topology to the correct neighbouring router. This is completely automated using System Verilog "generate" blocks to ensure that the Topology file contains no hard-coding.

```
`ifndef VR_define
`define VR_define

// Arbiter identifiers
`define MATRIX_ARBITER        1
`define ROUND_ROBIN_ARBITER   2

// Allocator identifiers
`define SEPARABLE_ALLOCATOR   1
`define WAVEFRONT_ALLOCATOR   2

// Arbiter selection switch
`define ARBITER_TYPE          1

// Allocator select switch
`define ALLOCATOR_TYPE        1

// Switch to indicate whether to arbiter in select_vc or encoder
`define SELECT_VC_ARBITRATE   0

// Flit data width
`define FLIT_DATA_WIDTH       32

// Direction encoding
`define EJECT   3'b000
`define NORTH   3'b001
`define SOUTH   3'b010
`define EAST    3'b011
`define WEST    3'b100

`define NIC_PORT    0

`define ROUND_TRIP 4

`endif
```

Figure 7: VR Define configuration file

```
// Create connections between routers
for (genvar i = 0; i < ROW_COUNT; ++i) begin
    for (genvar j = 0; j < COL_COUNT; ++j) begin
        // Connect north port
        assign rt_input_valid[i][j][`NORTH] = rt_output_valid[(i + (ROW_COUNT-1)) % ROW_COUNT][j][`SOUTH];
        assign rt_input_data [i][j][`NORTH] = rt_output_data [(i + (ROW_COUNT-1)) % ROW_COUNT][j][`SOUTH];
        assign rt_dwnstr_credit_increment [i][j][`NORTH-1] = rt_upstr_credit_increment [(i + (ROW_COUNT-1)) % ROW_COUNT][j][`SOUTH-1];
        // Connect south port
        assign rt_input_valid[i][j][`SOUTH] = rt_output_valid[(i + 1) % ROW_COUNT][j][`NORTH];
        assign rt_input_data [i][j][`SOUTH] = rt_output_data [(i + 1) % ROW_COUNT][j][`NORTH];
        assign rt_dwnstr_credit_increment [i][j][`SOUTH-1] = rt_upstr_credit_increment [(i + 1) % ROW_COUNT][j][`NORTH-1];
        // Connect east port
        assign rt_input_valid[i][j][`EAST] = rt_output_valid[i][(j + 1) % COL_COUNT][`WEST];
        assign rt_input_data [i][j][`EAST] = rt_output_data [i][(j + 1) % COL_COUNT][`WEST];
        assign rt_dwnstr_credit_increment [i][j][`EAST-1] = rt_upstr_credit_increment [i][(j + 1) % COL_COUNT][`WEST-1];
        // Connect west port
        assign rt_input_valid[i][j][`WEST] = rt_output_valid[i][(j + (COL_COUNT-1)) % COL_COUNT][`EAST];
        assign rt_input_data [i][j][`WEST] = rt_output_data [i][(j + (COL_COUNT-1)) % COL_COUNT][`EAST];
        assign rt_dwnstr_credit_increment [i][j][`WEST-1] = rt_upstr_credit_increment [i][(j + (COL_COUNT-1)) % COL_COUNT][`EAST-1];
    end
end
```

Figure 8: Scalable topology connections

5 EVALUATIONS

The designs and testbenches were developed in System Verilog and simulated using Synopsys VCS Functional Verification solution Version R-2020.12-SP2 on the Georgia Institute of Technology ECE server ece-linlabsrv01.ece.gatech.edu. Each buffer in the VC has a depth of 8.

5.1 Library modules Testing

The testbenches were developed to allow easier command line debug for more accurate data and flit tracking as compared to visual methods like graphs. So, they generated command line logs in addition to graphs. The evaluations along with testbench results used for verification are as follows:

```
Time = 0, Request = xxxx, Grants = xxxx
Time = 30, Request = 1111, Grants = 0001
Time = 40, Request = 1111, Grants = 0010
Time = 60, Request = 1111, Grants = 0100
Time = 80, Request = 1111, Grants = 1000
Time = 100, Request = 1111, Grants = 0001
Time = 120, Request = 1111, Grants = 0010
Time = 140, Request = 1111, Grants = 0100
Time = 160, Request = 1111, Grants = 1000
Time = 180, Request = 1111, Grants = 0001
Time = 200, Request = 1111, Grants = 0010
Time = 220, Request = 1111, Grants = 0100
```

Figure 9: Round-robin arbiter test log

```
grants[0]:0, grants[1]:0, grants[2]:1, time:600
requests[0]:1, requests[1]:0, requests[2]:1, time:600
dable[0]:1, dable[1]:1, dable[2]:0, time:600
weight_mat[0][0]:1,weight_mat[0][1]:1,weight_mat[0][2]:0,
weight_mat[1][0]:0,weight_mat[1][1]:1,weight_mat[1][2]:0,
weight_mat[2][0]:1,weight_mat[2][1]:1,weight_mat[2][2]:1,time:600
```

Figure 10: Matrix arbiter test log

```
grants[0]:0001 grants[1]:0010 grants[2]:0000 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111

grants[0]:0010 grants[1]:0100 grants[2]:0000 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111
```

Figure 11: Separable allocator+Round-robin arbiter test log

(1) PASS: Round-robin arbiter test shown in Fig. 9.
(2) PASS: Matrix arbiter test shown in Fig. 10.
(3) PASS: Separable allocator+Round-robin arbiter test shown in Fig. 11.

```
grants[0]:0100 grants[1]:0010 grants[2]:0001 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111

grants[0]:0000 grants[1]:0001 grants[2]:0010 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111

grants[0]:0010 grants[1]:0000 grants[2]:0000 grants[3]:0100
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111
```

Figure 12: Separable allocator+Matrix arbiter combo test log

```
Time=0, Requests = xxxx xxxx xxxx xxxx, Grants = xxxx xxxx xxxx xxxx
Time=30, Requests = 1111 0111 0111 1111, Grants = 0001 0000 0100 0010
Time=40, Requests = 1111 0111 0111 1111, Grants = 0010 0001 0000 0100
Time=60, Requests = 1111 0111 0111 1111, Grants = 0100 0010 0001 1000
Time=80, Requests = 1111 0111 0111 1111, Grants = 1000 0100 0010 0001
Time=100, Requests = 1111 0111 0111 1111, Grants = 0001 0000 0100 0010
Time=120, Requests = 1111 0111 0111 1111, Grants = 0010 0001 0000 0100
```

Figure 13: Wavefront allocator+Round-robin arbiter test log

```
grants[0]:0100 grants[1]:0010 grants[2]:0001 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111

grants[0]:0000 grants[1]:0001 grants[2]:0010 grants[3]:0000
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111

grants[0]:0010 grants[1]:0000 grants[2]:0000 grants[3]:0100
requests[0]:0111,requests[1]:0111,requests[2]:0111,requests[3]:0111
```

Figure 14: Wavefront allocator+Matrix arbiter test log

```
// Direction encoding
`define NORTH   3'b000
`define SOUTH   3'b001
`define EAST    3'b010
`define WEST    3'b011
`define EJECT   3'b100
```

Figure 15: Direction encoding

```
Time = 10, current_router=6, dest_router=12, direction=11
Time = 20, current_router=13, dest_router=2, direction=10
Time = 30, current_router=11, dest_router=15, direction=00
Time = 40, current_router=0, dest_router=7, direction=10
Time = 50, current_router=6, dest_router=11, direction=10
```

Figure 16: Route compute unit test log

(4) PASS: Separable allocator+Matrix arbiter test shown in Fig. 12.
(5) PASS: Wavefront allocator+Round-robin arbiter test shown in Fig. 13.
(6) PASS: Wavefront allocator+Matrix arbiter test shown in Fig. 14.
(7) PASS: Route compute unit test shown in Fig. 15 with direction encoding shown in Fig. 16.
(8) PASS: VC Allocation unit test shown in Fig. 17.

```
Time = 45, Port=0, VC=0, available=1111000000000000000, allocated=1000000000000000
Time = 45, Port=0, VC=1, available=1111000000000000000, allocated=0
Time = 45, Port=0, VC=2, available=11110000, allocated=100000
Time = 45, Port=0, VC=3, available=1111, allocated=10
Time = 45, Port=1, VC=0, available=1111000000000000000, allocated=0
Time = 45, Port=1, VC=1, available=111100000000000, allocated=1000000000000
Time = 45, Port=1, VC=2, available=1111, allocated=0
Time = 45, Port=1, VC=3, available=11110000, allocated=0
Time = 45, Port=2, VC=0, available=11110000, allocated=0
Time = 45, Port=2, VC=1, available=1111000000000000000, allocated=0
Time = 45, Port=2, VC=2, available=111100000000, allocated=1000000000
Time = 45, Port=2, VC=3, available=1111, allocated=0
Time = 45, Port=3, VC=0, available=11110000, allocated=0
Time = 45, Port=3, VC=1, available=11110000, allocated=0
Time = 45, Port=3, VC=2, available=1111, allocated=0
Time = 45, Port=3, VC=3, available=111100000000, allocated=0
Time = 45, Port=4, VC=0, available=11110000, allocated=0
Time = 45, Port=4, VC=1, available=1111, allocated=0
Time = 45, Port=4, VC=2, available=1111000000000000000, allocated=0
Time = 45, Port=4, VC=3, available=111100000000000, allocated=0
```

Figure 17: VC allocator test log

```
********************Time = 80************
**********INPUT SIGNALS******************
Reset = 0
Input:
    input_data[0]=01101100000000000000011011110100
    input_data[1]=00110000000000000000000001101011
    input_data[2]=00010100000000000000001001001000
    input_data[3]=11001000000000000000001111110001
    input_data[4]=00001100000000000000001111111101
    input_valid=11111
```

Figure 18: Input Signals for router top test

Through testing we found that the acyclic Wavefront allocator does not provide the required level of fairness for a sparse request matrix. In those situations, a separable allocator is more preferable. Thus, in the VC allocation stage, with the matrix size being 20x20 for 5 ports and 4 VCs, the separable allocator will work better.

5.2 Router Top Testing

For the full system (router top) testing, we send inputs to the router to simulate the 4 input ports and 1 NIC, with each port contains 4 VCs. Then study its flow through the different router pipeline stages and eventually check if the flit gets ejected from the router. The output directions are shown in Fig. 15. The flits are encoded with 6 MSB bits containing the VC ID and destination router ID (2 bits + 4 bits), along with 11 LSB bits encoded with a unique ID for tracking the flit across the router stages. The VC ID and destination router ID are generated randomly. The stage wise progression of the input flit (highlighted in each image) is as follows:

(1) Fig. 18 shows the inputs that are applied to the top router module at timestamp=80.
(2) Fig. 19 shows the status of the buffers after the buffer write stage.
(3) Fig. 20 shows the route which is computed for the flit being tracked in the route compute stage.
(4) Fig. 21 shows the output VC allocated to the tracked flit in VC allocation.

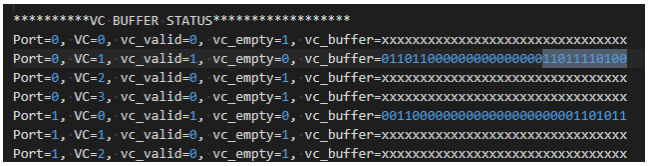

Figure 19: Buffer status for router top test

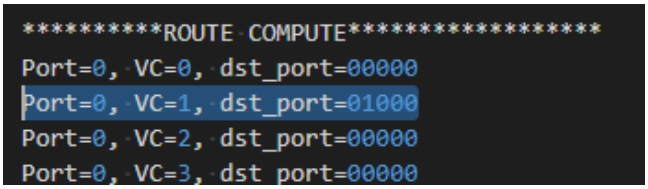

Figure 20: Router compute for router top test

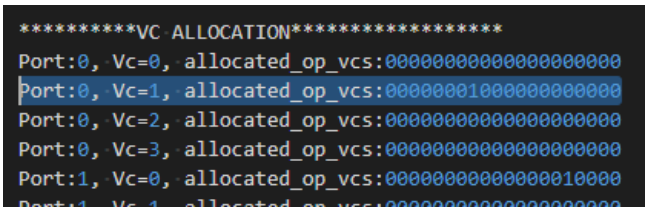

Figure 21: VC Allocation for router top test

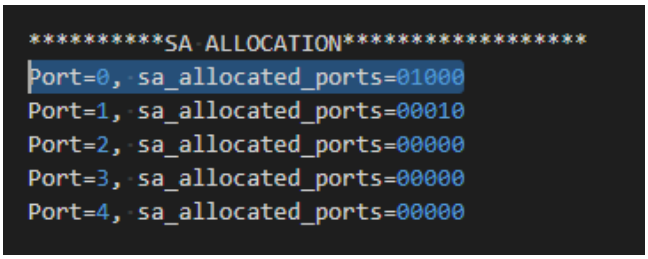

Figure 22: SW Allocation for router top test

(5) Fig. 22 shows the output port allocated to the tracked flit in the SW allocation stage.
(6) Fig. 23 shows the buffer read operation on the port and VC specific to the tracked flit.
(7) Fig. 24 shows the tracked flit traversing the switch to the final output port.

This stage wise tracking of the flits shows that the router module is working as expected.

5.3 Topology Integration Testing

For testing the router operation in a topology, we created a torus topology. The topology module is created in a manner which allows it to scale to any dimensions using the configurable parameters. The wiring to each router in the topology gets auto-configured using the defined generate loops. We instantiate our router to form the 5x5 Torus topology and ran tests on the complete system. We

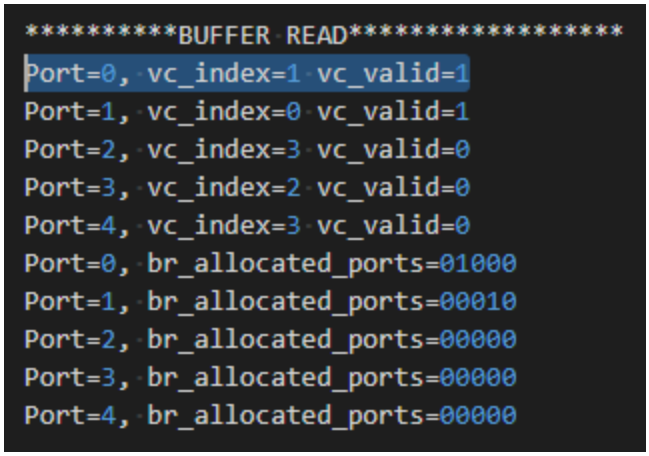

Figure 23: Buffer Read for router top test

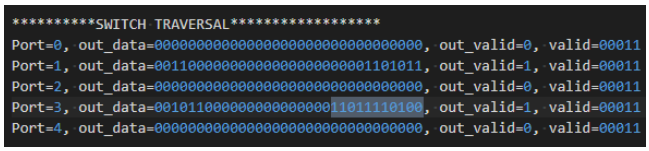

Figure 24: Switch Traversal for router top test

observed that the system works as expected by randomly injecting flits into the network and tracking them to successful ejection at the correct destination routers. The 7 MSB bits contains the VC ID and Destination router ID (2 bits + 5 bits) and 11 LSB bits are encoded with a unique random number between 0, 2047 to enable easier tracking. The test scenarios are as follows:

(1) Scenario 1:
  (a) Input flit is highlighted in Fig. 25 injected into Router 0 at Time=100 with destination as Router 20, unique ID = 1648.
  (b) Output flit is highlighted in Fig. 26 ejected at Time=280 at Router 20, tracked with unique ID = 1648.
(2) Scenario 2:
  (a) Input flit is highlighted in Fig. 27 injected into Router 3 at Time=160 with destination as Router 23, unique ID = 1772.
  (b) Output flit is highlighted in Fig. 28 ejected at Time=340 at Router 23, tracked with unique ID = 1772.
(3) Scenario 3:
  (a) Input flit is highlighted in Fig. 29 injected into Router 17 at Time=440 with destination as Router 2, unique ID = 1197.
  (b) Output flit is highlighted in Fig. 30 ejected at Time=740 at Router 2, tracked with unique ID = 1197.

5.4 RTL Synthesis

We synthesized the router with NCSU FreePDK45 a open-source 45nm standard cell library at 100MHz in Cadence Genus. This involved setting up a TCL script that configured the clock period, external input & output delay values, top level design module and

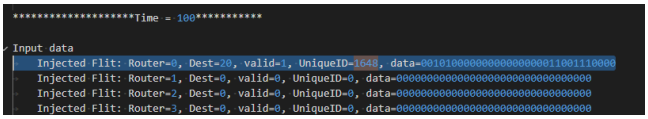

Figure 25: Inputs injected into Router 0

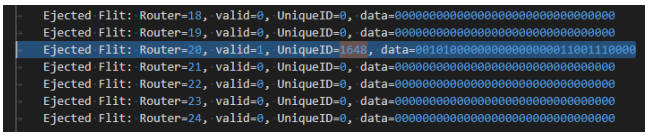

Figure 26: Output ejected from Router 20

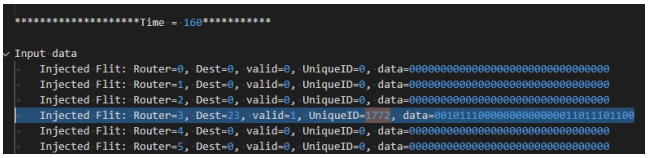

Figure 27: Inputs injected into Router 9

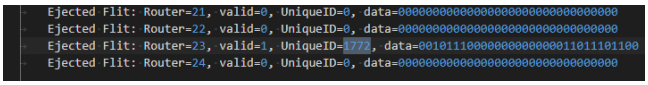

Figure 28: Output ejected from Router 23

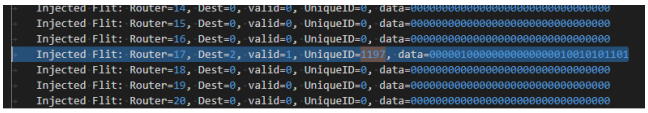

Figure 29: Inputs injected into Router 17

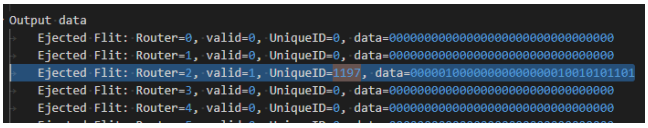

Figure 30: Output ejected from Router 2

including the RTL files present in different levels of hierarchy within the project directory. Procedure to synthesize RTL file using Cadence Genus:

(1) Launch Cadence Genus in legacy UI mode using the following command: /tools/software/cadence/genus/latest/bin/genus -legacy_ui
(2) The above command launches Genus with a terminal to run TCL scripts.
(3) We then create our TCl file with the following details and run into through the terminal:
  (a) Set PDK Location: /tools/designkits/NCSU/FreePDK45
  (b) Set all HDL files to be read.
  (c) Set top level design module to 'router_top'.
  (d) Set Clock period.
  (e) Apply clock constraints and name the clock pin.
  (f) Set fall and rise time to 400ps.
  (g) Run synthesis using: syn_gen, syn_map commands.
  (h) Generate timing, number of gates and power report.
  (i) Launch gui to view the schematic.

6 RESULTS

One of the synthesized schematic is shown in Fig. 31.

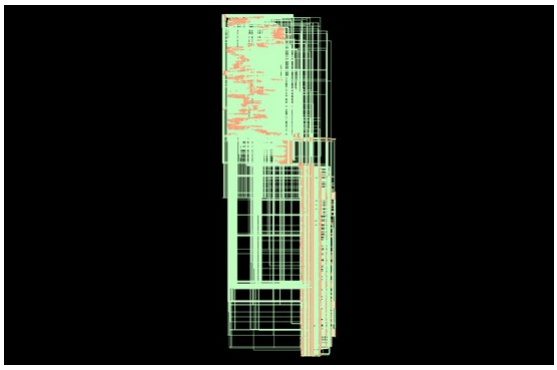

Figure 31: Synthesized Schematic

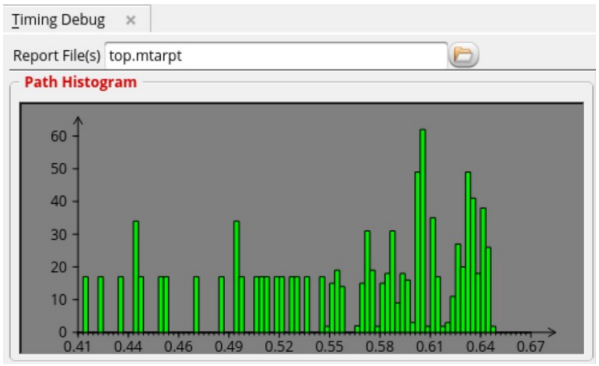

Figure 32: Timing Histogram for all timing paths, green signifies that all paths show a positive slack

Additionally, we generated timing, power and gate count statistics as follows:

(1) **Timing results**: The router shows a positive slack for all timing paths at **100MHz**. Timing histogram is shown in Fig. 32.
(2) **Power results**: The Power Consumption results for the four configurations are shown in Table 1. The percentage distribution of total power across each component is shown in Fig. 34
(3) **Area results**: The Area calculation results for the four configurations are shown in Table 2. The percentage distribution of total area across each component is shown in Fig. 33

The Wavefront allocator takes more area than the separable allocator, but its area does not scale as the arbiter type or sizes

Power in mW	Matrix Arbiter	Round Robin Arbiter
Separable Allocator	5.73	4.87
Wavefront Allocator	5.32	5.31

Table 1: Power Consumption Results

Area in um^2	Matrix Arbiter	Round Robin Arbiter
Separable Allocator	169,637.40	131,979.36
Wavefront Allocator	177,392.58	174,544.87

Table 2: Power Consumption Results

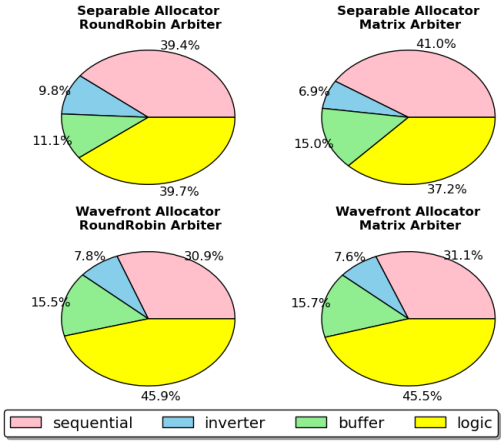

Figure 33: Area contributions of each element towards the total area for all configuration

changes as it uses only 1 arbiter to change the priority whereas the separable allocator is built using two rows of multiple arbiters. So, the wavefront allocator could prove better at large configurations of ports and VCs as the per arbiter size increase could scale the area of the separable to higher values than the logic cell array of the wavefront would scale. The Matrix Arbiter combination occupies most area and consumes most amount of power. We were also able to generate the mapped design and SDC (Synopsis Design Constraints) files that give us an overview of the tool's interpretation of the design.

7 CONCLUSION

We have successfully designed the scalable and configurable Virtual Channel Router. We have successfully achieved all of the design and milestone goals we set for this project. The configurability is supported by a top level configuration file that can be used to control all the design parameters like flit width, number of ports, number of VCs, buffer depth, round trip latency, choose the arbiter and allocator for the router stages. All the router stages and library

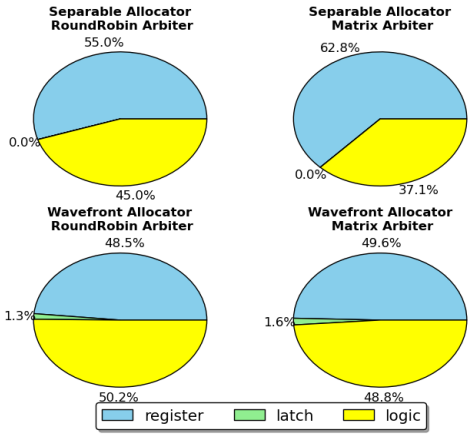

Figure 34: Power contributions of each element towards the total power for all configuration

modules have been fully parameterized to allow scaling the router to any number of ports and VCs. The router can also be used within any topology by only updating the 'Route Compute' unit to match the topology the router is integrated into.

8 FURTHER WORK

The next steps for extending the work are as follows:

(1) Area efficient route compute unit that uses alternatives to modulo and division operation.
(2) Increase clock frequency by splitting the VC allocation stage using pipelines.
(3) Perform Post-Layout testing of the router design.

REFERENCES

[1] Luca Benini and Giovanii de Micheli. Networks on Chip: A New Paradigm for Systems on Chip Design. In *Proceddings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, peages 418-419, 2002.
[2] William J Dally and Brian Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th Conference on Design Automation*, pages 684-689, 2001.
[3] Pierre Guerrier and Alain Greiner. A Generic Architecture for On-Chip Packets-Switched Interconnections. In *Proceeding of the 2000 Design, Automation and Test in Europe Conference and Exhibition*, pages 250-256, 2000.
[4] Gpta Pankaj and Nick Mckeown. Desiging and Implementing a Fost Crossbar Scheduler. *IEEE Micro,* 19(1):20-28, 1999.