On-Chip Interconnection Network

CSEE 4340:
Computer Hardware Design
Fall 2012

SECTION 1:

Project Title: On-Chip Interconnection Network
Team Name: Flit Busters

Project Members:	Adil Sadik	(ams2378)
	Ashwin Ramachandran	(ar2648)
	Dechhin Lama	(ddl2126)
	Ayushi Rajeev	(ar3110)

Project Member Duties:

Design Team:

- Adil Sadik (Master)
- Dechhin Lama

Verification Team:

- Ashwin Ramachandran (Master)
- Ayushi Rajeev

Section 2:
Design Overview:

Bus based interconnection systems are not scalable and power efficient for future many core System-on-chips or processors. Network on Chip provides a scalable and efficient solution to this problem. In this project, we will design a synthesizable 4x4 Network on Chip of mesh topology using SystemVerilog for design and validation.

A NoC consists of multiple communicating cores, intermediate router nodes and point-to-point links. Intermediate router nodes are responsible for routing data to its appropriate destination, and thus establish a notion of point-to-point communication between a sender and receiver. Data is represented by messages. A message can consist of multiple packets, where each packet is composed of multiple flits. The very first flit of each packet (header flit) contains the information required to locate the destination node. All of the following flits (body flits) contain actual messages.

The placement of routers and cores is realized with a two-dimensional grid, where each node is identified with a unique (X, Y) co-ordinate. The routing algorithm used for this project is deterministic. Data will always be routed in a predefined order – first along the Y-axis and then along the X-axis. This is referred to as dimension ordered routing. After receiving the first flit of each packet, a router node will locate the co-ordinate of the destination node and route the flit through Y-X dimension-ordered routing scheme mentioned above. The NoC handles the messages in a flit-by-flit basis. This means that the router doesn't wait until it receives all the flits associate with a packet; instead it transmits each flit as soon as it knows that the next router has available space in its input buffer. All the body flits associated with a header flit follow the same routing path established for the header flit. This is referred to as wormhole routing.

To prevent resource starvation and congestion inside the network, the NoC will implement a distributed flow-control mechanism. Specifically, we use credit-based flow-control, which ensures that a router will only transmit data when there are available slots in the next router to accommodate the received flit. Hence, each router keeps tracks of the available free slots in the input buffers of its adjacent routers through the notion of credits. The number of credits indicates the number of free slots in the input data buffer of a router. By implementing credit communication among adjacent routers, we can ensure that the routers are keeping track of the available resources required to make routing decisions for each flit.

Each router node consists of 5 input and 5 output ports - one pair of input-output ports for data in the north, south, east, and west directions, with one extra pair for the local data. Each input port consists of a queue, which can store up to 5 incoming flits per direction. There is a crossbar switch, which routes one direction's input to a different direction's output, and 5 outputs buses, which are capable of sending one flit per cycle. In addition to this routing hardware, there exist credit counters that

keep track of how many free slots exist in the neighboring router nodes' input buffers.

The overall design will follow a top down methodology. The first goal is to implement and validate the routing of a single header flit. Afterwards the body flits will be incorporated with the header flit and it will be ensured that they follow wormhole routing scheme as mentioned before. One important checkpoint will be to design and validate one single router with all the ports and required functionality. Once the router is ready, the whole NoC will be instantiated, and the routers located on the perimeter will be optimized (i.e. eliminate unnecessary ports).

SECTION 3: Unit level interfaces

Router- Top Level

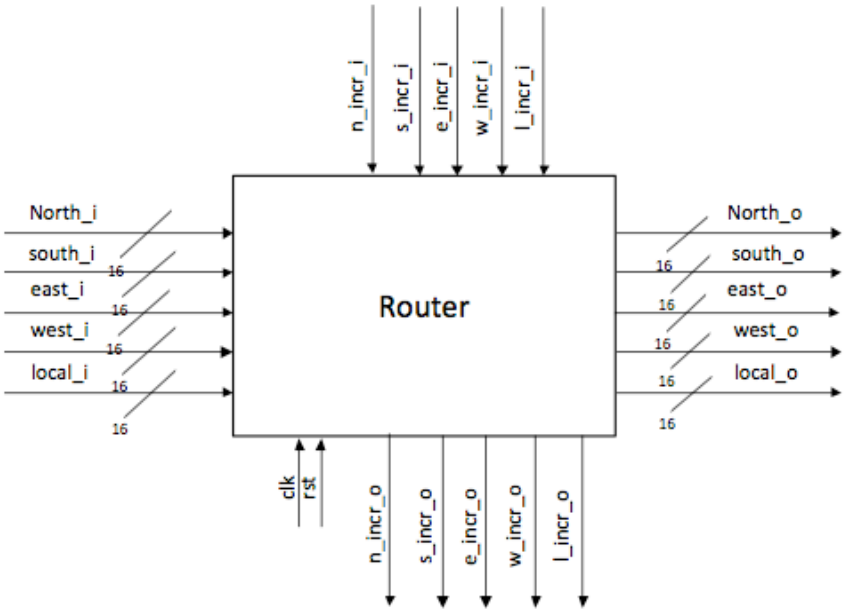

Figure 1: Router Top Level

The above figure represents the general top level interfaces for a router with four neighbors. In case of edge and corner routers, we just need to eliminate the unwanted input and output interfaces.

Signal description:

Signal Name	Type	Width	Description
clk	I	1	Clock
rst	I	1	Reset
n_incr_i	I	1	Increment North counter if empty slot
s_incr_i	I	1	Increment South counter if empty slot

e_incr_i	I	1	Increment East counter if empty slot
w_incr_i	I	1	Increment West counter if empty slot
l_incr_i	I	1	Increment local counter if empty slot
north_i	I	16	Data input from north port
south_i	I	16	Data input from south port
east_i	I	16	Data input from east port
west_i	I	16	Data input from west port
local_i	I	16	Data input from local port
n_incr_o	O	1	Increment north neighboring counter
s_incr_o	O	1	Increment south neighboring counter
e_incr_o	O	1	Increment east neighboring counter
w_incr_o	O	1	Increment west neighboring counter
l_incr_o	O	1	Increment local neighboring counter
north_o	O	16	Data routed to the north output
south_o	O	16	Data routed to the south output
east_o	O	16	Data routed to the east output
west_o	O	16	Data routed to the west output
local_o	O	16	Data routed to the local output

A more detailed view of the router is presented bellow. It includes all the sub-blocks including the connection between them. Detailed description of each of this blocks will be presented in section 4- (sub-unit partitioning and interfaces).

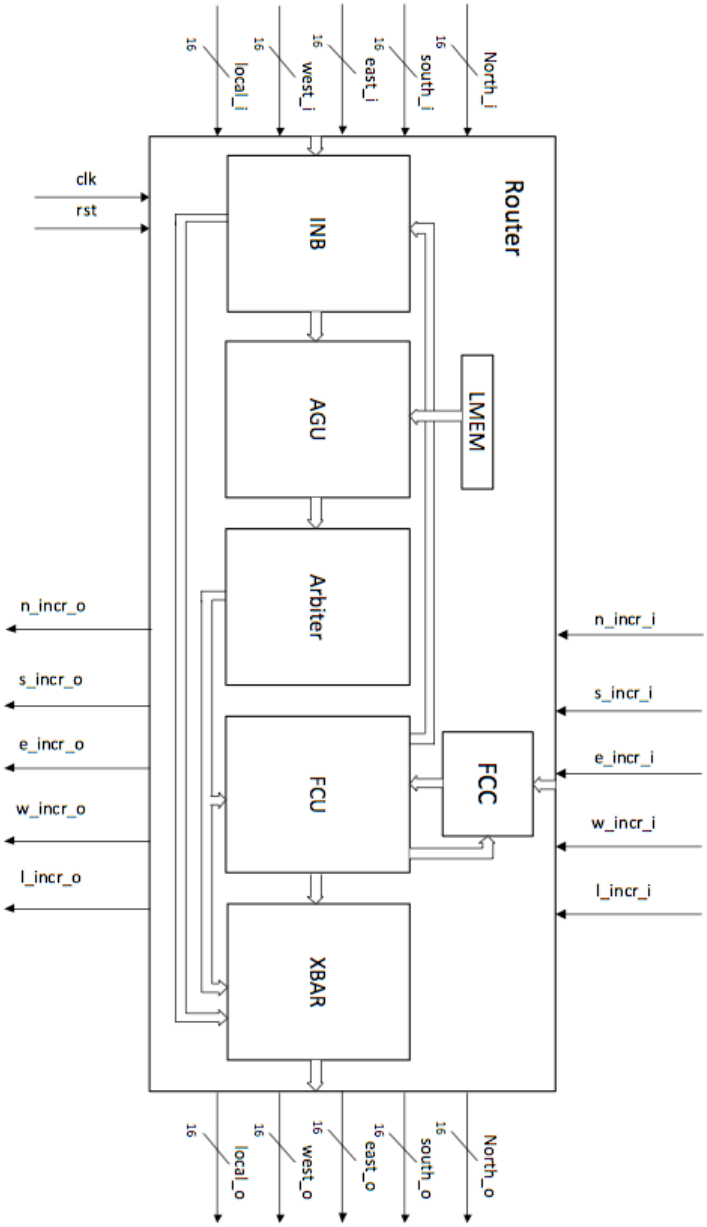

Figure 2: Router Architecture - in block level

SECTION 4: Sub partitioning the interfaces, Test harness structure

Sub-blocks: The router has following sub-blocks-

1. INB - Input Buffer
2. AGU - Address Generation Unit
3. Arbiter
4. FCU - Flow Control Unit
5. FCC - Flow Control Counter
6. XBAR - Cross bar to route flits
7. LMEM- Local Memory

1. Input Buffer

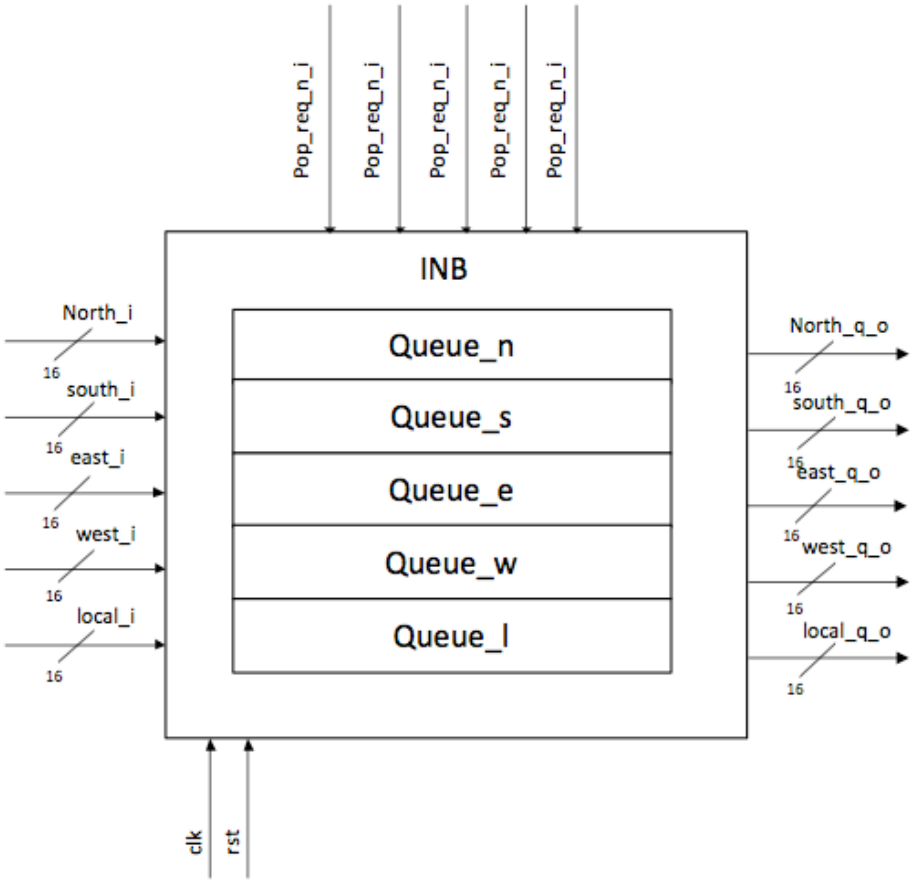

Figure 3 : Input buffer

Input buffer is consist of queues. It consists of five queues (depth- 4, width- 16) - one for each input port. Flits are saved in the queues of input buffer after receiving from the neighbor nodes. Each queue stores the flits and waits for the pop request. Once it receives the pop request, it pop the head flit and send it to the XBAR. XBAR routes this flit to the proper output port.

Signal description:

Signal name	Type	Width	Description
clk	I	1	Clock
rst	I	1	Reset
pop_req_n_i	I	1	Pop request to north queue
pop_req_s_i	I	1	Pop request to south queue
pop_req_e_i	I	1	Pop request to east queue
pop_req_w_i	I	1	Pop request to west queue
pop_req_l_i	I	1	Pop request to local queue
north_i	I	16	Data input from north port
south_i	I	16	Data input from south port
east_i	I	16	Data input from east port
west_i	I	16	Data input from west port
local_i	I	16	Data input from local port
north_q_o	O	16	Data output from north queue
south_q_o	O	16	Data output from south queue
east_q_o	O	16	Data output from east queue
west_q_o	O	16	Data output from west queue
local_q_o	O	16	Data output from local queue

2. Address Generation Unit

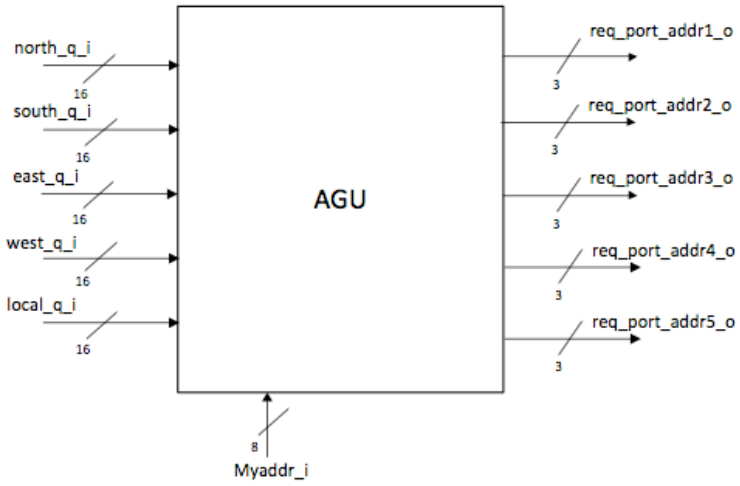

Figure 4: Address Generation Unit

This unit basically takes the header flit form the input buffer queue and finds out in which port the input flit should be routed to. It sends this routing information to the Arbiter.

Signal description:

Signal name	Type	Width	Description
myaddr_i	I	8	Input port address
north_q_i	I	16	Data input from north queue
south_q_i	I	16	Data input from south queue
east_q_i	I	16	Data input from east queue
west_q_i	I	16	Data input from west queue
local_q_i	I	16	Data input from local queue
req_port_addr1_o	O	3	Output address for north queue
req_port_addr2_o	O	3	Output address for south queue
req_port_addr3_o	O	3	Output address for east queue
req_port_addr4_o	O	3	Output address for west queue

req_port_addr5_o	O	3	Output address for local queue

3. Arbiter

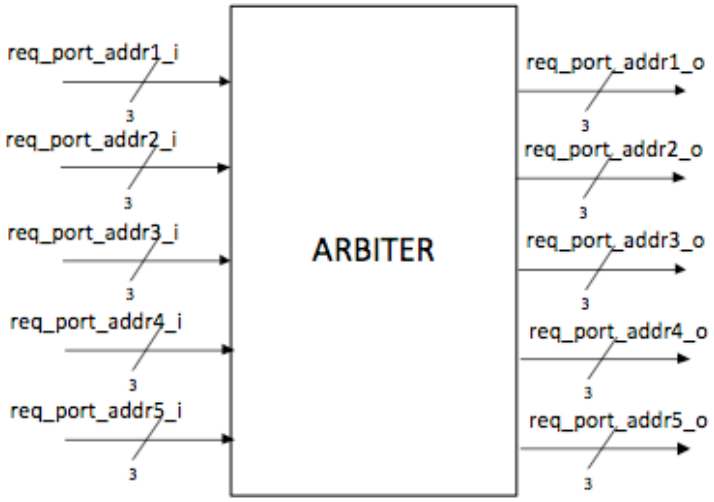

Figure 5: Arbiter block

Arbiter block receives information about the destination node each flit want to access and if there is a scenario where two distinct packets want to access one single port - as an example, flit received from north and south input port want to access east output port - then it handles the issue. It basically implements a *round robin* arbiter.

Signal description:

Signal Name	Type	Width	Description
req_port_addr1_i	I	3	Requested address for north queue
req_port_addr2_i	I	3	Requested address for south queue
req_port_addr3_i	I	3	Requested address for east queue
req_port_addr4_i	I	3	Requested address for west queue
req_port_addr5_i	I	3	Requested address for local queue
req_port_addr1_o	O	3	Output address for north queue

req_port_addr2_o	O	3	Output address for south queue
req_port_addr3_o	O	3	Output address for east queue
req_port_addr4_o	O	3	Output address for west queue
req_port_addr5_o	O	3	Output address for local queue

4. Flow Control Unit

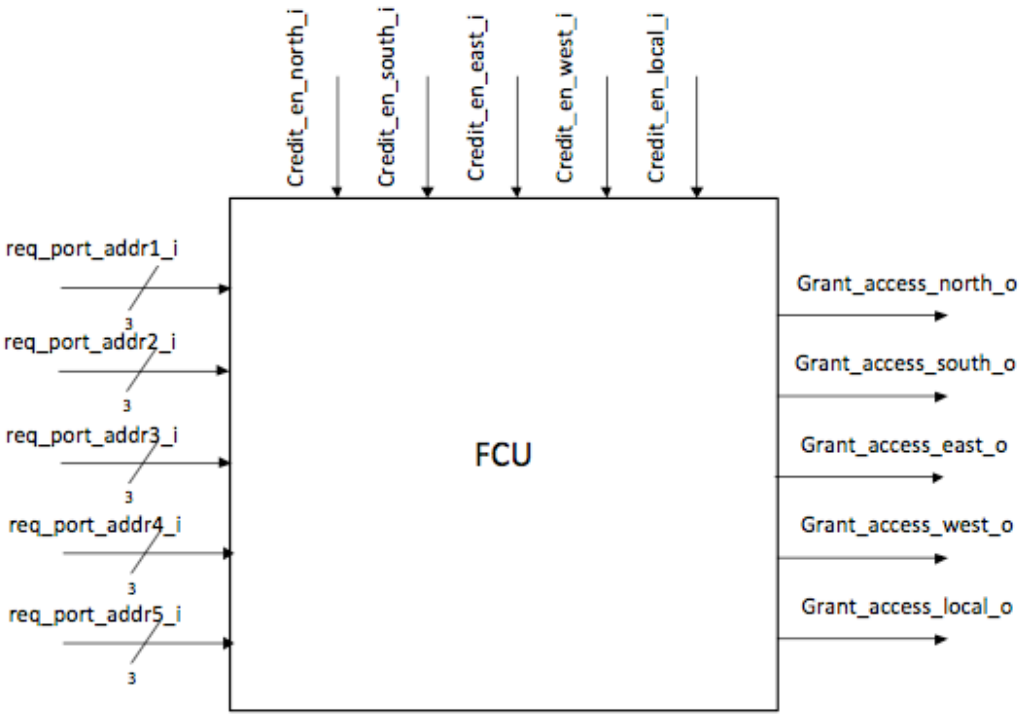

Figure 6 : Flow control unit - block diagram

Flow control unit implements the *credit-based flow control* mechanism. Before data is sent to the XBAR, FCU ensures that the input port of the destination router has enough space in it's input buffer to accommodate the new input. FCC (Flow Control Counter) block tracks the the number of available slots in the neighboring nodes. FCU receives this information from FCC and determines whether the data should be routed to the requested output port or stalled inside the input queue (i.e. INB) of the host router. If the flit is granted access based on credit, then it sends pop request to the respective queue.

Signal Description:

Signal Name	I/O	W	Description
req_port_addr1_i	I	3	Requested address for north queue
req_port_addr2_i	I	3	Requested address for south queue
req_port_addr3_i	I	3	Requested address for east queue
req_port_addr4_i	I	3	Requested address for west queue
req_port_addr5_i	I	3	Requested address for local queue
credit_en_north_i	I	1	Enable signal for North queue empty slot
credit_en_south_i	I	1	Enable signal for South queue empty slot
credit_en_east_i	I	1	Enable signal for East queue empty slot
credit_en_west_i	I	1	Enable signal for West queue empty slot
credit_en_local_i	I	1	Enable signal for Local queue empty slot
grant_access_north_o	O	1	Enable signal for North output
grant_access_south_o	O	1	Enable signal for South output
grant_access_east_o	O	1	Enable signal for East output
grant_access_west_o	O	1	Enable signal for West output
grant_access_local_o	O	1	Enable signal for Local output

5. Flow Control Counter

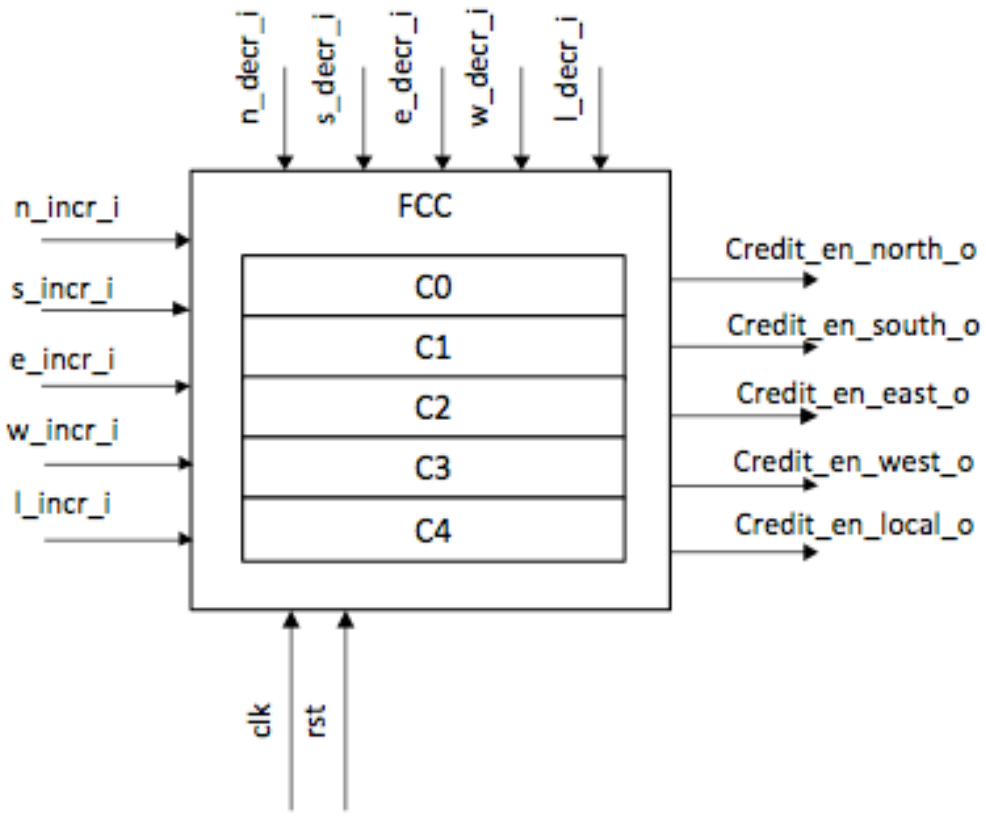

Figure 7 : Flow Control Counter - block diagram

FCC and FCU together ensures the credit based flow control. FCC contains five counters which counts the number of available free slots in the neighbor router nodes and local processor. C0 - C4 above represent these five counters. Initially these counters are initialized with '5'. It decreases the count when FCU determines that a flit has credit to be routed to the output port. On the other hand, it increases the count when it receives information from neighbor router that it has freed its slots.

Signal description:

Signal Name	I/O	W	Description
n_incr_i	I	1	Increment North counter if empty slot
s_incr_i	I	1	Increment South counter if empty slot
e_incr_i	I	1	Increment East counter if empty slot

w_incr_i	I	1	Increment West counter if empty slot
l_incr_i	I	1	Increment local counter if empty slot
n_decr_i	I	1	Decrement North counter if an empty slot taken
s_decr_i	I	1	Decrement South counter if an empty slot taken
e_decr_i	I	1	Decrement East counter if an empty slot taken
w_decr_i	I	1	Decrement West counter if an empty slot taken
l_decr_i	I	1	Decrement Local counter if an empty slot taken
credit_en_north_o	O	1	Enable signal for North queue empty slot
credit_en_south_o	O	1	Enable signal for South queue empty slot
credit_en_east_o	O	1	Enable signal for East queue empty slot
credit_en_west_o	O	1	Enable signal for West queue empty slot
credit_en_local_o	O	1	Enable signal for Local queue empty slot

6. Cross- bar

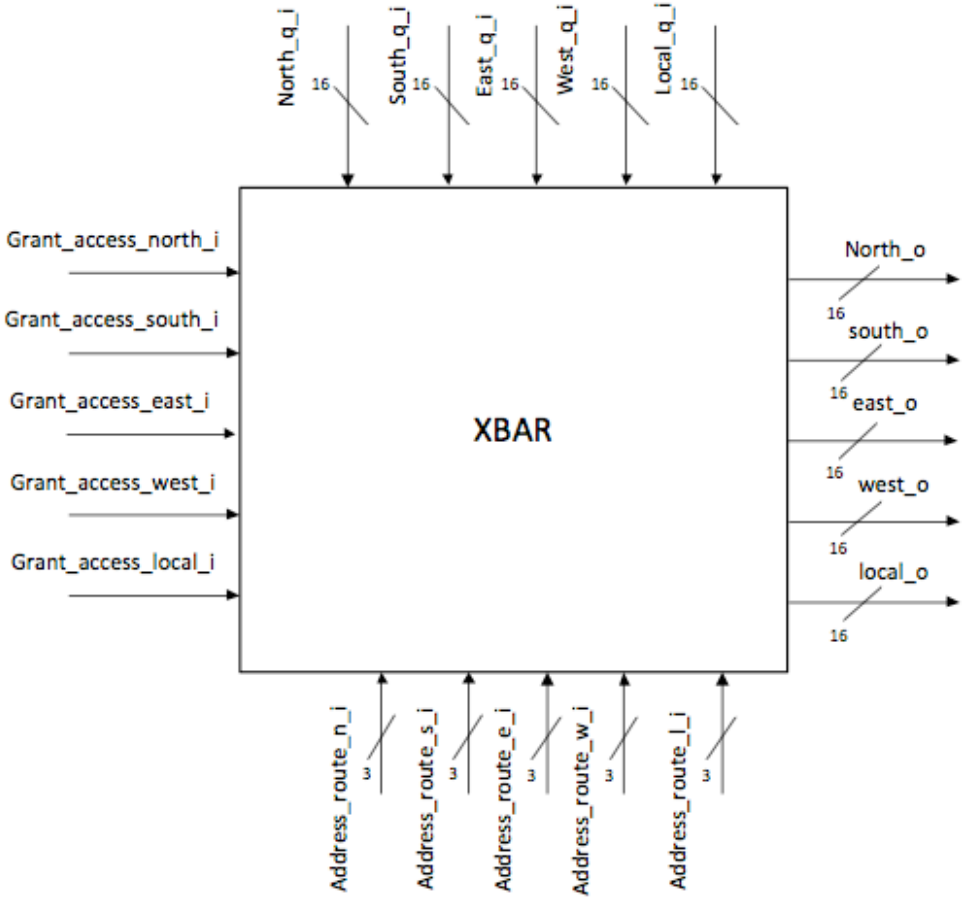

Figure 8 : Cross bar

Crossbar routes the flit to the respective output port. It receives the information regrading which port a flit wants to access from the Arbiter and the respective enables from the FCU which tells if the data should be routed or stalled. If the enable is 1, it routes the data to the output port, otherwise data is stalled in the queue and waits for available credit.

Signal description:

Signal name	I/O	W	Description
grant_access_north_i	I	1	Enable signal for North output
grant_access_south_i	I	1	Enable signal for South output
grant_access_east_i	I	1	Enable signal for East output

grant_access_west_i	I	1	Enable signal for West output
grant_access_local_i	I	1	Enable signal for Local output
north_q_i	I	16	Data input from north queue
south_q_i	I	16	Data input from south queue
east_q_i	I	16	Data input from east queue
west_q_i	I	16	Data input from west queue
local_q_i	I	16	Data input from local queue
address_route_n_i	I	3	Address of incoming port for north output
address_route_s_i	I	3	Address of incoming port for south output
address_route_e_i	I	3	Address of incoming port for east output
address_route_w_i	I	3	Address of incoming port for west output
address_route_l_i	I	3	Address of incoming port for local output
north_o	O	16	Data routed to the north output
south_o	O	16	Data routed to the south output
east_o	O	16	Data routed to the east output
west_o	O	16	Data routed to the west output
local_o	O	16	Data routed to the local output

7. Local Memory

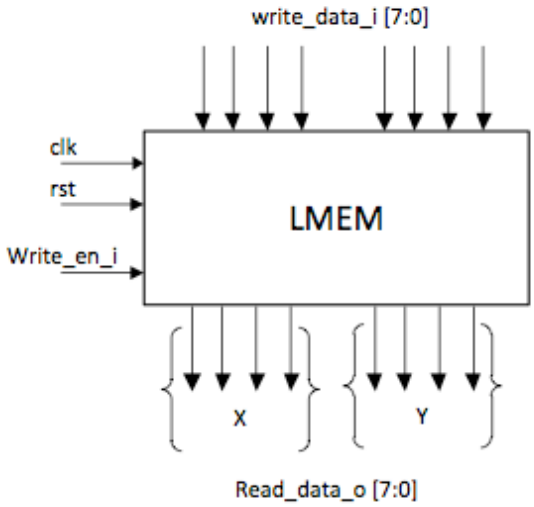

Figure 9 : local memory

This is just a local memory for each router note. It contains the address (co-ordinate) of each router. The co-ordinate are stored in an one-hot notation. Hence each X and Y will have 4 bits as shown above in the figure. Co-ordinates are set up at the beginning when the NoC is initialized.

VALIDATION

This part involves the validation of the sub-partitioned design units for the Network on Chip as well as the top-level validation for the entire router. Each validation skeleton has essentially 4 parts:

1. Interface
   This file has to be modified for each module in accordance with the sub-unit specification for interfaces given by the design team. It defines the basic inputs and outputs for each module as seen by the DUT and the bench. It is also responsible for defining the timing constraints as they relate to the system clock (using the clocking block construct provided by SystemVerilog).

2. Top-level file
   This file assigns the clock, defines the interfaces, and instantiates the appropriate Design Under Test (DUT) and the testbench.

3. Bench file
   This file is the one responsible for running the simulation. It stimulates the DUT with randomly generated inputs, takes the generated output

signals from the DUT, generates its own outputs for the same stimuli, and compares the DUT value to its own value. Simulation continues until either an error is found or the maximum number of simulation cycles have been executed. In order to simplify use of the testbench and make its operation more understandable, the bench file is split into classes. Each class is responsible for doing a single task. The bench file consists of the following classes:

- Environment class
  This class initializes the simulation parameters such as simulation duration, transaction densities for each interface, bit masks, verbosity, etc. The parameters that are initialized are specific to the unit being tested.
- Checker class
  This defines whether the result passes or fails based on comparison of the DUT output and the bench-generated output.
- Tester class
  This contains the golden model function, which duplicates the behavior of the DUT using non-synthesizeable SystemVerilog. The values generated by the golden model function are then checked by the checker class
- Packet class
  This class defines the basic interface of communication with the module. It defines the different input and output fields specific to the DUT. It is also responsible for defining which inputs are random – a crucial component of the simulation.
- do_cycle task
  This task increments the cycle number and is responsible for passing the appropriate data to the DUT and golden model. It makes sure that clocked modules only sample their inputs at the clock edge, and that outputs are ready before the next clock edge.

4. Makefile
   The Makefile is responsible for assembling the correct SystemVerilog source code, compiling it, and running it.

Each of the 4 files mentioned above must be modified appropriately for each testable sub-module.

We will continue to take a bottom-up approach to validating our router. This will be done by first implementing the design of the most basic sub-modules, implementing the corresponding testbench using the already assembled test harness, and

validating their operation. This will ensure that as we assemble our higher level modules, any errors we find at the higher level of abstraction will be due to the higher level module under test, *not* the sub-modules that it instantiates. As it stands, the empty test harnesses for each of the currently-defined modules and sub-modules successfully compile with the top level files and run for 10,000 cycles.

SECTION 5: Microarchitecture design
SECTION 6: Verification Strategy
SECTION 7: Performance estimation
SECTION 8: Area estimate
SECTION 9: Bugs, Coverage
SECTION 10: Document revision history