On-chip Interconnection Network

Team Name: CTSZ

Design Group: Han Cui hc2737(master) Rishina Tah rt2545
Validation Group: Hui Zou hz2361(master) Yuzhe Shen ys2821

Instructor: Simha Sethumadhavan

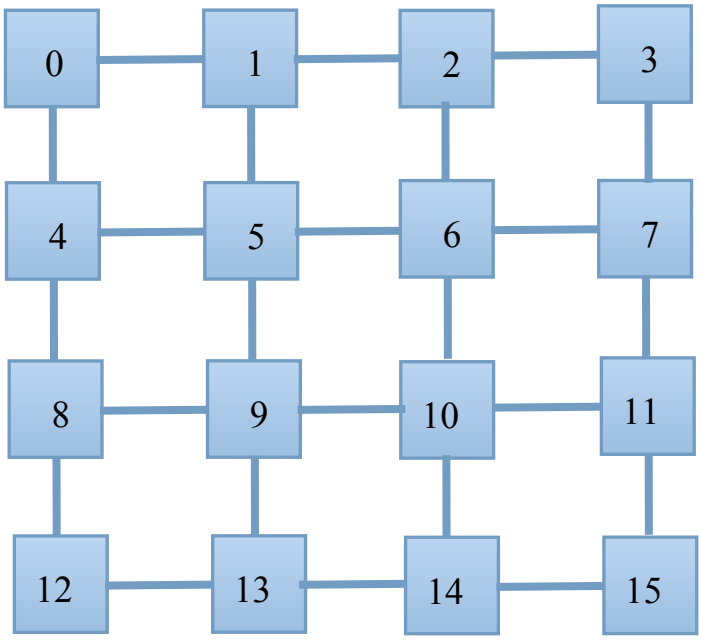
Table of Contents

1. Design Overview

The project's purpose is to build a 4*4 mesh network using completely synthesizable Verilog HDL. The input of the design module are the data packages to some nodes on the network and the output are the same data packages from other nodes on the network after some delays. To finish this, we need to design 16 submodules static structure units each of which represents as a node on the network and the inter-communication methods of them.

1.1 Overall Structure Description:

The basic function of the network is to receive the input 16-bit-data which is called "a flit" at a certain node and transfer the data to the desired node for output.Each node is represented by a router. Thus the basic static units are the "routers". These routers are distributed on Y-X dimension. They are placed in four columns and four rows. The network is based on the intercommunication of such distributed routers. Each router is assigned with a specific number from 0 to 15 as shown in Figure 1.



A router generally consists of a crossbar switch, an arbiter, five input buffers, five credit counters, data input ports (North input, East input, West input, South input, Local input) signal input ports (input credit counts) and data output ports (Local, N, E, W, S), state input ports includes global clock signal, global reset signal as shown in Figure 2.

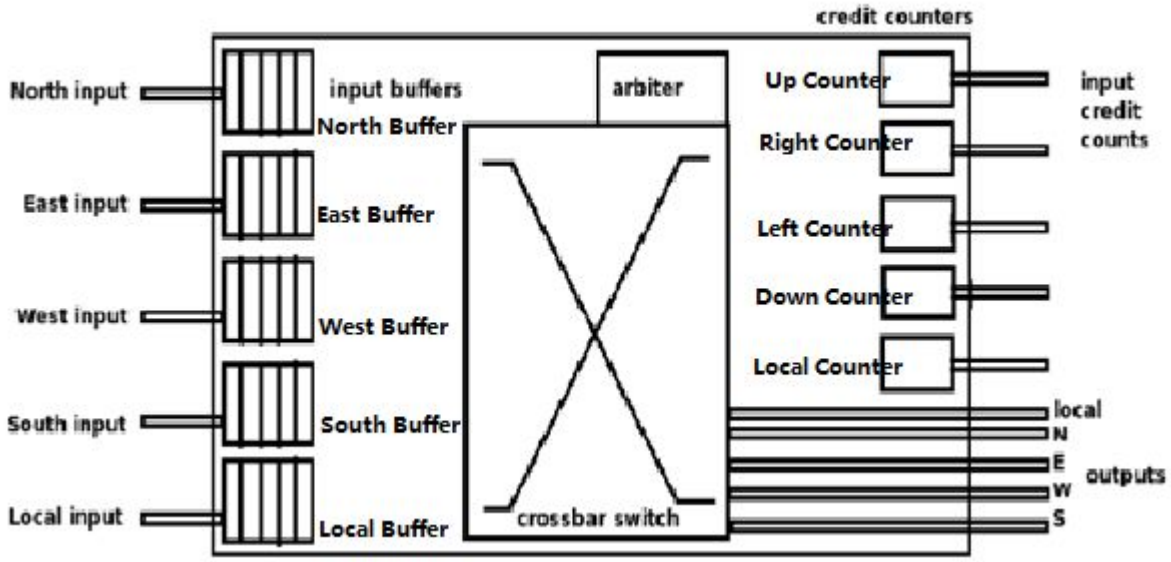


Figure 2, a general router, copied from project description file

1.2 Static Structure Description:

The static structure is the structure of the network that doesn't get involved with inter-router data transmission. This is a result-oriented structure. It consists of arbiter, crossbar switch, local input port, local port, reset port and clock port.

1.2.1 Reset/Clock Port:

Each of the routers contains reset port and clock port, these ports receive global synchronous reset and clock signal. The synchronous reset is applied for the purpose of splitting different tasks and initializing the routers.

1.2.2 Crossbar Switch

If there are flits in any of the input buffers of a router, the router's crossbar switch will analyze the flits and send the data to designated destination according to the transmission priority. In basic level analysis, the crossbar switch will determine the destination of the input flits by analyzing the header flits as well as the router placement; while in the challenge level analysis, the output destination calculated from basic level analysis should be recorded for the following four body flits. Meanwhile, to avoid deadlock, transmission of data within a single clock cycle should be unilateral. For example, flits in South Buffer should not be sent out to the router in the South direction; flits in Local Buffer should not be directed into local port.

1.2.3 Arbiter

The arbiter deals with the situation when two or more flits from different input buffers of a router wish to be transferred to the same output wire simultaneously, with the specific scheduling algorithm. It is to avoid corner cases such that flits in some input buffers wait for a long time before they are sent out. If one input buffer always has new input flits and always requests to send the existing flits out, the arbiter should

not let the flits from it sent out if other flits from other buffers also request the same output location. The scheduling algorithm will see if there is flits waiting in all the five input buffers except for the corresponding buffer of the output direction every time a flit is sent out in the order according to which input buffer the newly sent-out flit comes from. For example, if the just sent-out flit is from North buffer to the E output port, then the arbiter will not search east input buffer since no flits in it have the desired east output direction. Its search order is West buffer, South buffer, Local buffer, and finally North buffer.

1.2.4 Local input/Local port

The local input port receives input data for the network and the local port will output the flits being transferred in the network.

1.3 Dynamic Structure Description

The dynamic structure is the structure of the network that focuses on inter-router data transmission. This is a process-oriented structure. It consists of input ports, input buffers, counters and output ports.

1.3.1 Data Input Ports

Each router has at most five input ports for data transfer. Each input port can transfer at most one flit in a certain time unit. Four of them are from the up, down, left, right neighbor routers and the other one is from the local input, which means that at this certain router point there is input signal from the outside world. Each router also sends output data packages to one of the five different directions similar to input direction. The output signal also transfers at most one flit in a certain time unit. The local output port will output the data being transferred in the network to the outside world while other four output ports send data to the neighbor routers.

1.3.2 Input Buffers

For each data input port the router uses an input buffer to store flits. Each buffer stores at most five flits. The buffers are first-in-first-out. Those input buffers are named North, South, West, East and Local Buffer respectively.

1.3.3 Credit Counters and Signal Input Port

Each counter records the number of credits available (empty flit space) in its neighbor's related input buffers. For example, a router should record the number of credits available of the South input buffer of the router in its North direction. There are four storage units in one router and are named Up, Down, Left, Right Counter respectively. This is implemented through up-down counters whose initial values are all five. The counter value is decremented if one flit is sent to the related buffer. The counter value will be incremented if one flit in that buffer is consumed. The counter knows this through signal input port. For example, when a router sends a flit to its North router, that router's Up Counter will decrease by one. When one flit in a router's North router's South input buffer is consumed, its North router will give a signal to this router's Up signal input port which let its Up Counter decrease by one. Each router also has a counter called Local Counter that records the credits available in its own Local Buffer. That counter increments when a flit in the Local Buffer is

consumed and decrements when a new flit arrives into the Local Buffer.

1.3.4 Data Output Ports

A router's N/E/W/S data output ports are directly connected to the input ports of this router's neighbor routers through interface. The local output port is the data sink the module and send data out to the outside world.

1.4 Communication Methods

Here is the description of the communication protocols and basic principles of data transfer among routers.

1.4.1 Data Package

We use package switching and flit based flow control in communication. Data packages are the basic transferable unit. In the basic part of this project one data package to be transferred consists only one 16-bit header flit that stores its destination information. In the challenge part one data package to be transferred consists of five flits, one header flit as well as four body flits (wormhole routing). The communication purpose is to transfer each data package to the correct destination.

1.4.2 Back pressure Mechanism

To prevent data package lose, back pressure mechanism is implemented. When the W/E/S/N input buffers are full as well as when the value of related counter is zero, no data transfer will happen to that specific buffer. For example, if a router's Up Counter value is zero, it will no longer send flits out to its North router. When Logic Buffer is full, a stall signal will be sent to the uplinker (outside world) and no new data will be created and filled into the Local Buffer.

1.4.3 Clock Synchronization

We use a global clock to synchronize all the data transfer processes. In one clock cycle, each router at most receive one flit from each input signal ports and at most send one flit out to each output direction.

1.4.4 Sequential and Combinational Logic

When some flits in a router are consumed in a certain clock, that router will send the signal to its related neighbor router at the same time to inform this consumption. Then the counter responds immediately so that in the next clock the counter value in its related neighbor router has already decreased. That means counters will change directly due to the input credit counts signals and are combinational logic units. The flits transfer are sequential logic and set synchronous by the global clock.

1.4.5 Edge Routers

Edge routers contain less data input ports, less output ports, less input data buffers, less counters and simpler arbiter logic. For optimization purpose we should make specific design for them.

2. Modules ,Sub-modules, Interfaces

2.1 Module Level

The top level synthesizable module of our design is a module called network. It contains hierarchical different levels of submodules. The bottom level is flip-flop for sequential units and MUX and gate units for combinational units. The top module for the whole project contains this network module and a testbench as well as an interface connecting the network and the testbench. The testbench is based on software language to test our design's functionalities. All the modules are listed in the Table 1 below.

Module Name	Next Level Submodule
top	network testbench
network	router
router	input_buffer counter crossbar_switch arbiter
input_buffer	buffer_storage
crossbar_ switch	valid_tester direction_analyzer output_data_logic header_body_logic header_body_record
output_data_logic	mux
valid_tester	mux
com_dir_logic	mux
direction_analyzer	comparator register com_dir_logic
arbiter	chooser arbiter_logic register
arbiter_logic	mux
chooser	resister
buffer_storage	register
counter	register
header_body_record	register
header_body_logic	mux
register	FF
comparator	no submodules
FF	no submodules
mux	no submodules

(Table 1, modules and submodules in the design)

2.2 Module Description

Then the following part is the introduction to each module and submodule appeared in the Table 1. We consider the functionality of each module as well as the module interface including input signals and output signals.

2.2.1 FF(Flip-flop)

The FF module interface information is shown below in Table 2.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
FF	clk	input	1	synchronous
	rst	input	1	reset
	data_i	input	1	input data
	enable_i	input	1	enable to store input data
	data_o	output	1	output data

(Table 2, FF interface information)

Description: it is a basic flip-flop to store one bit for a cycle synchronized by the global clock signal. It is the basic unit of the sequential logic.

2.2.2 Register

The register module interface information is shown below in Table 3.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
register	clk	input	1	synchronous
	rst	input	1	reset
	enable_i	input	1	enable to store input data
	data_i	input	n	input data
	data_o	output	n	output data

(Table 3, register interface information)

Description: it consists of n flip-flops and can be used to store n-bit data for a cycle. It is the second level basic unit of the sequential logic. Here we use the parameter n to make this module more flexible. And further we will design specific number instead of n for synthesizable consideration.

2.2.3 Comparator

The comparator module interface information is shown below in Table 4.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
Comparator	data1_i	input	n	Input data
	data2_i	input	n	Output data

	Result_o	output	2	Compare result

(Table 4, comparator interface information)

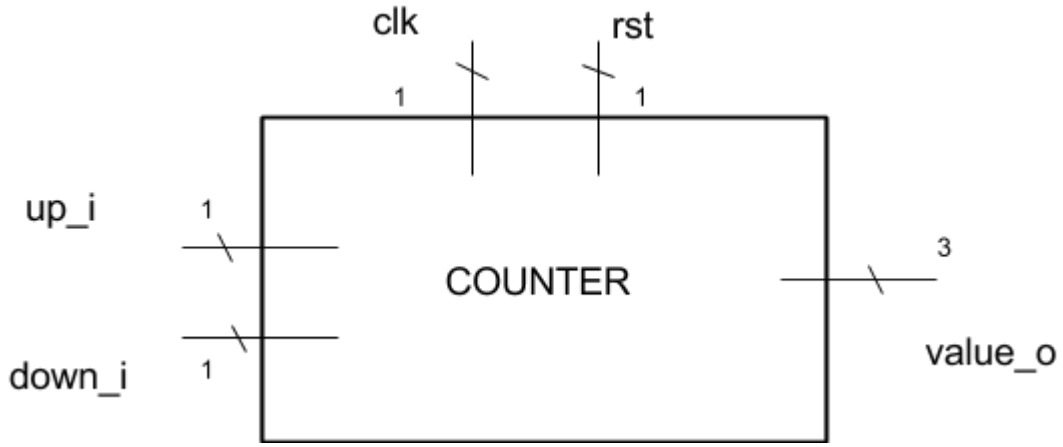
Description: it compares two input data and give a result to indicate which data is large in terms of unsigned notation. Output can have four different kinds and three of them represent data1_i is larger than/equal to/ smaller than data2_i. It is a basic combinational logic unit. We temporarily use a parameter n to indicate input data width.

2.2.4 counter

The counter module interface information is shown below in Table 5 and Figure 3.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
Counter	clk	input	1	synchronous
	rst	input	1	reset
	up_i	input	1	Signal to count up
	down_i	input	1	Signal to count down
	value_o	output	3	Current value stored

(Table 5, counter interface information)



(Figure 3, counter interface)

Description: it is a sequential logic unit and used to hold a value which can be manipulated with increment and decrement. Since the possible maximum value needed to store in our design is five, we only need 3 bits to represent all the possible values including number 0 to 5 plus one more kind to represent error, when attempting to increase the value five or decrease the value zero. When reset is high, the value is set 5.

2.2.5 mux

The mux module interface information is shown below in Table 6.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
mux	control	input	1	To choose one input data
	data1_i	input	n	Input data
	data2_i	input	n	Input data
	data_o	output	n	output data

(Table 6,mux interface information)

Description: it chooses one of the two input data as output due to the value of control signal. We temporarily use a parameter n to indicate input data width.

2.2.6 buffer_storage

The buffer_storage module interface information is shown below in Table 7.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
buffer_storage	receive_i	input	1	Whether buffer should receive a new flit
	data_i	input	17	Input data
	send_i	input	1	Whether buffer should send out a flit
	data_o	output	17	Output data, the oldest flit currently in the buffer
	clk	input	1	synchronous
	rst	input	1	reset

(Table 7, buffer_storage interface information)

Description: it stores at most five flits and is controlled by signals to send/receive flits. It records the order in terms of arrival time of the flits. The special point for this module is that it always send out the oldest data inside it to other modules. When send_i is high, it erases the current oldest flit and choose the second oldest flit as oldest data and begins to send it out on the data_o port. If no flits are stored, it sends out 17-bit 0s. It includes five registers. When reset is high, the five registers inside it are all reset.

2.2.7 chooser

The chooser module interface information is shown below in Table 8.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
chooser	north_hit_i	input	1	Input data

	east_hit_i	input	1	Input data
	west_hit_i	input	1	Input data
	south_hit_i	input	1	Input data
	local_hit_i	input	1	Input data
	last_dir_i	input	5	Last winner flit is from which input buffer
	dir_o	output	5	The winner flit will be from which input buffer
	filter_o	output	5	Show which buffer gets/loses the permission
	arb_o	output	1	Whether arbitration is needed for a specific output direction

(Table 8, chooser interface information)

Description: the chooser will analyze if arbitration to a specific output direction is needed by comparing the desired output direction of at most four input flits from input buffers. When more than one flits are requesting the same output direction, chooser will choose who gets the permission with predefined algorithms. The algorithm needs to know in the last time when a flit is sent out through that chooser direction, which input buffer that flit comes from. The body flits have higher priority than the header flits in a chooser.

2.2.8 arbiter_logic

The arbiter_logic module interface information is shown below in Table 9.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
arbiter_logic	arb1_i	input	1	Input data
	arb2_i	input	1	Input data
	arb3_i	input	1	Input data
	arb4_i	input	1	Input data
	arb5_i	input	1	Input data
	fil1_i	input	5	Input data
	fil2_i	input	5	Input data
	fil3_i	input	5	Input data
	fil4_i	input	5	Input data
	fil5_i	input	5	Input data
	arb_o	output	1	Whether

				arbitration is needed and involved
	filter_o	output	5	Flits from which input buffer are delayed by the arbitration

(Table 9, arbiter_logic interface information)

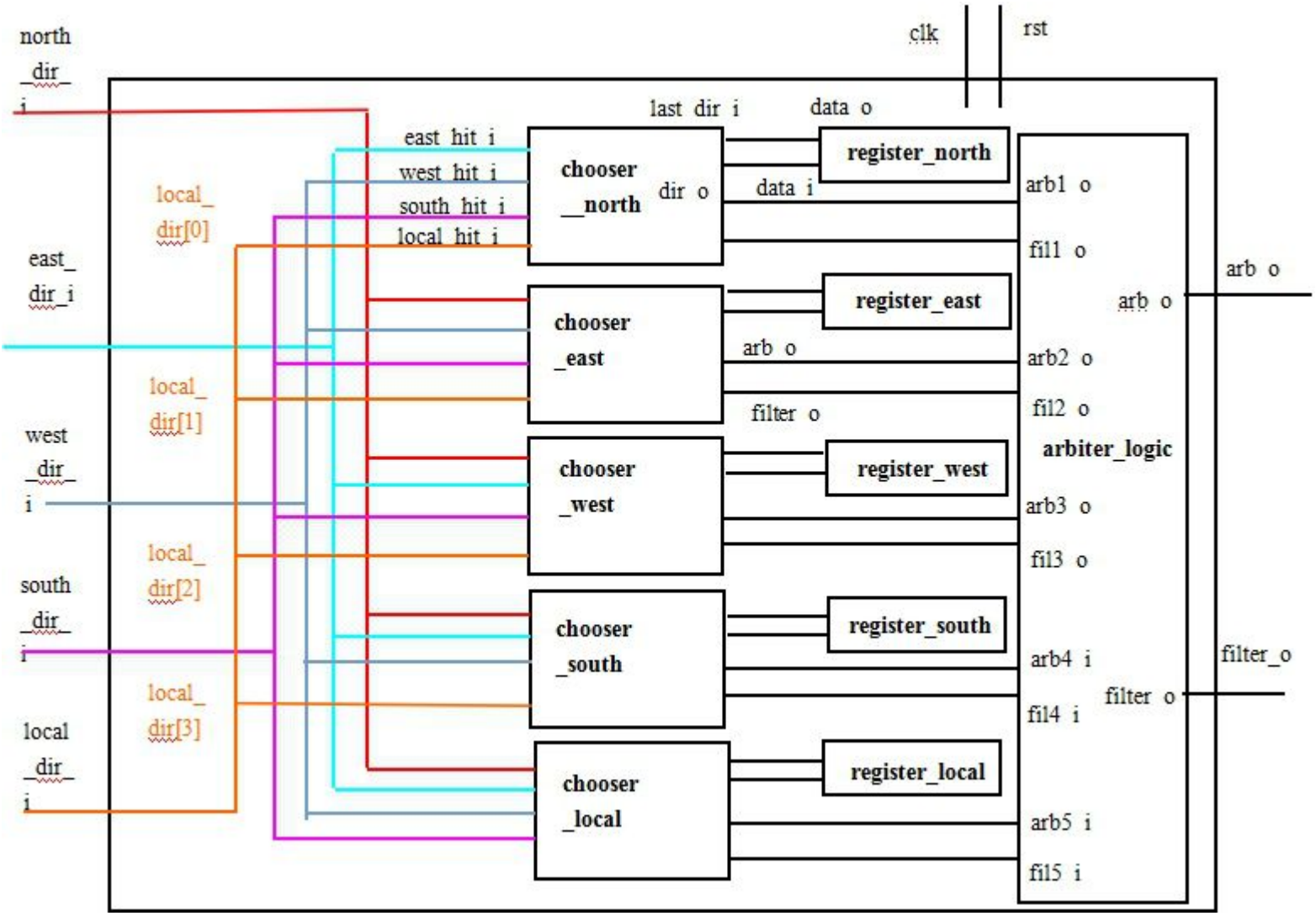
Description: this module is used to analyze and combine the five chooser results to get the final arbiter results and output those results. For output filter_o, one bit represents one input buffer.

2.2.9 arbiter

The arbiter module interface information is shown below in Table 10 and Figure 3.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
arbiter	clk	input	1	synchronous
	rst	input	1	reset
	north_dir_i	input	6	Input buffer desired output direction
	east_dir_i	input	6	Input buffer desired output direction
	west_dir_i	input	6	Input buffer desired output direction
	south_dir_i	input	6	Input buffer desired output direction
	local_dir_i	input	6	Input buffer desired output direction
	arb_o	output	1	Whether arbitration is needed and involved
	filter_o	output	5	Flits from which input buffer are delayed by the arbitration

(Table 10, arbiter interface information)



(Figure 3, arbiter interface)

This figure shows the arbiter input/output and next-level sub-modules inside it. It also shows the interface of its sub-modules. For one wire in the picture, it is the output of one module and input of another module so it can be marked with two different names. Each chooser has four direction_hit inputs since its own direction will not have valid input. Register is used to record the last_dir_i data and updated by dir_o.

Description: the arbiter will use choosers and arbiter_logic to analyze if arbitration is needed. It includes five choosers one of which is targeted to one output direction. It also includes When it is needed, arbiter will judge which output directions need its arbitration, and decide for those directions which input buffer can/cannot send out their flits to them. For output filter_o, one bit represents one input buffer.

2.2.10 direction_analyzer

The direction_analyzer module interface information is shown below in Table 11.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
dicrection_analyzer	data_i	input	17	Input data
	valid_i	input	1	Whether the input data is a valid flit
	direction_o	output	6	Output direction

(Table 11, direction_analyzer interface information)

Description： this module is used to calculated out a data package/flit' s desired output direction by analyzing the last eight bits of the input data and comparing them to the location of router. Since the possible kinds of output direction is five, each output bit represents one direction. Bit value high means that direction is desired. The leftmost bit represents whether this output information is valid in terms of whether the input flit is valid.

2.2.11 com_dir_logic

The com_dir_logic module interface information is shown below in Table 12.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
com_dir_logic	com1_i	input	2	Input compare result
	com2_i	input	2	Input compare result
	valid_i	input	1	Whether the flit is valid
	dir_o	output	6	Output direction

(Table 12, com_dir_logic interface information)

Description: this module is a combinational logic unit to calculate out the desired output direction of a flit by analyzing the X, Y dimension comparator results. Each bit in output dir_o represents one output direction (1 when desired output is that direction) and the leftmost is the extra check bit. It has a local register to record the router's location. That value is different for each router. We temporarily use logic type and assignment sentence to represent it. For synthesizable purpose, it should be further declared separately and differently for each router as a submodule.

2.2.12 valid_tester

The valid_tester module interface information is shown below in Table 13.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
valid_tester	credit_i	input	3	How many flits are already in the neighbor routers
	data_i	input	17	Input data

	data_o	output	17	Output data
	valid_o	output	1	Whether input data is valid to transfer further

(Table 13, valid_tester interface information)

Description: this module will judge whether the data_i is a valid flit. Also, it will also judge if there is extra space in the receiver buffer of a neighbor router by analyzing the credit_i signal from the credit buffer. If no space in the neighbor router is available, the output data will be set 0.

2.2.13 output_data_logic

The output_data_logic module interface information is shown below in Table 14.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
output_data_logic	north_data_i	input	17	Input data
	east_data_i	input	17	Input data
	west_data_i	input	17	Input data
	south_data_i	input	17	Input data
	local_data_i	input	17	Input data
	arb_i	input	1	Whether the arbitration is executed
	filter_i	input	5	Which input buffer is delayed by the arbiter
	north_dir_i	input	6	North input buffer desire output direction
	east_dir_i	input	6	East input buffer desire output direction
	west_dir_i	input	6	West input buffer desire output direction
	south_dir_i	input	6	South input buffer desire output direction
	local_dir_i	input	6	Local input buffer desire output direction
	north_data_o	output	17	Output data
	east_data_o	output	17	Output data
	west_data_o	output	17	Output data

	south_data_o	output	17	Output data
	local_data_o	output	17	Output data
	send_data_o	output	5	Represent which input buffer can send a flit out

(Table 14, output_data_logic interface information)

Description: this module analyzes all the information from arbiter and five direction_analyzers to decide which input flits should be sent out and send them to the correct output direction. The input buffer may not send out a flit if it has no valid flit, or neighbor router has no space, no arbiter does not give it permission.

2.2.14 header_body_logic

The header_body_logic module interface is shown below in the table 15.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
header_body_logic	dir_i	input	6	Input data
	count_i	input	3	Input judgment data
	dir_record_i	input	6	Header flit direction to record
	dir_record_o	output	6	Previous header flit direction
	dir_o	output	6	Output data after judgment between body/header flit
	add_o	input	1	a valid flit has come through

(Table 15, header_body_logic interface information)

Description: this module is combinational and used for tell whether a valid data is a body flit or a header flit and correspondingly gives the header flit desired direction.

2.2.15 header_body_record

The header_body_record module interface is shown below in the table 16.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
header_body_record	clk	input	1	synchronous
	rst	input	1	reset
	dir_i	input	6	Header flit

				direction to record
	add_i	input	1	To count up
	dir_o	output	6	Previous header flit direction
	count_o	output	3	Number of flits in a header/body flit cycle.

(Table 16, header_body_record interface information)

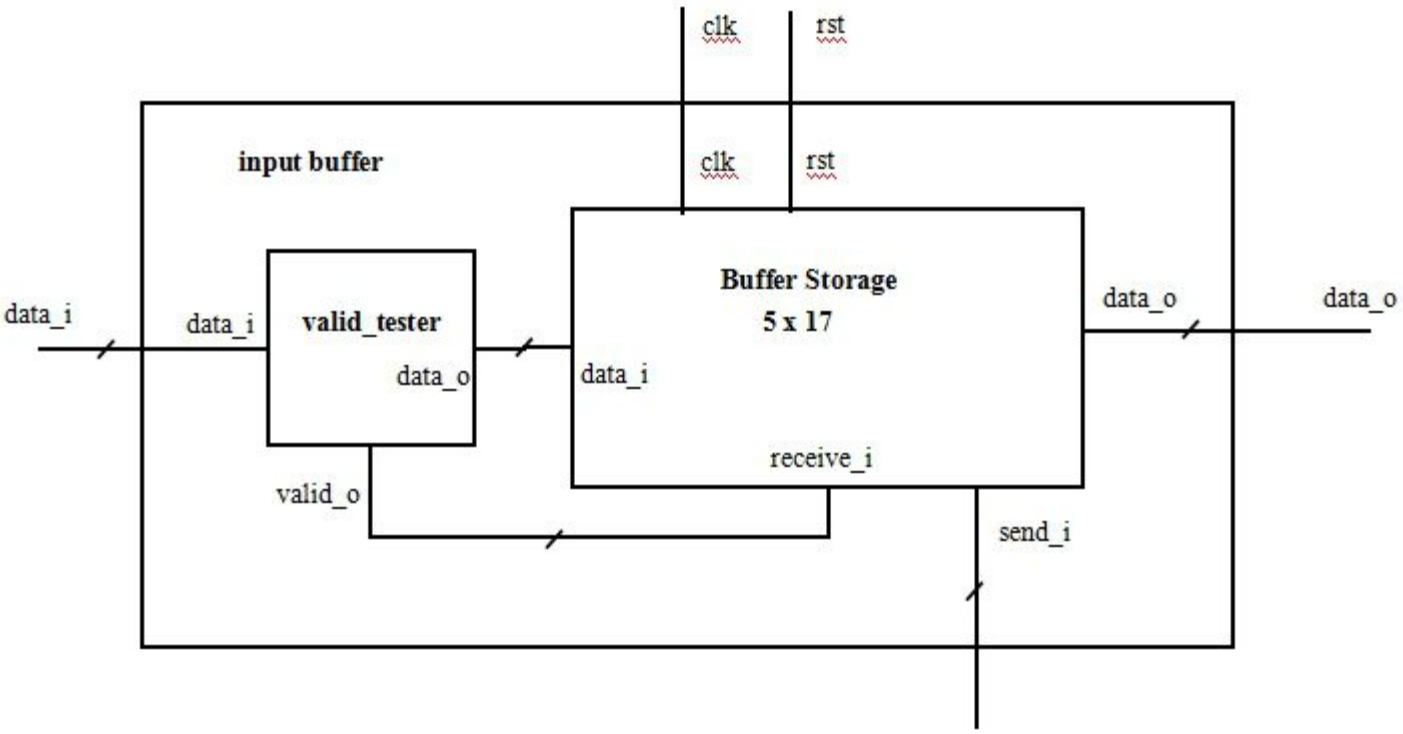
Description: this module is used for record how many flits has already come through the header_logic in order to judge the coming flit is a body/header flit. It also records the desired output direction information for every header flit.

2.2.16 input_buffer

The input_buffer module interface information is shown below in Table 17 and Figure 4.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
input_buffer	data_i	input	17	Input data
	data_send_i	input	1	Whether to erase the recorded oldest data
	data_o	output	17	Output data, always be the currently oldest data
	clk	input	1	synchronous
	rst	input	1	reset

(Table 17, input_buffer interface information)



(Figure 4, input buffer interface)

Description: the input buffer includes a buffer_storage submodule. It stores the flits from corresponding neighbor routers or from the local input port. It sends the oldest data out to the switch_logic sub-module in the router for further analysis.

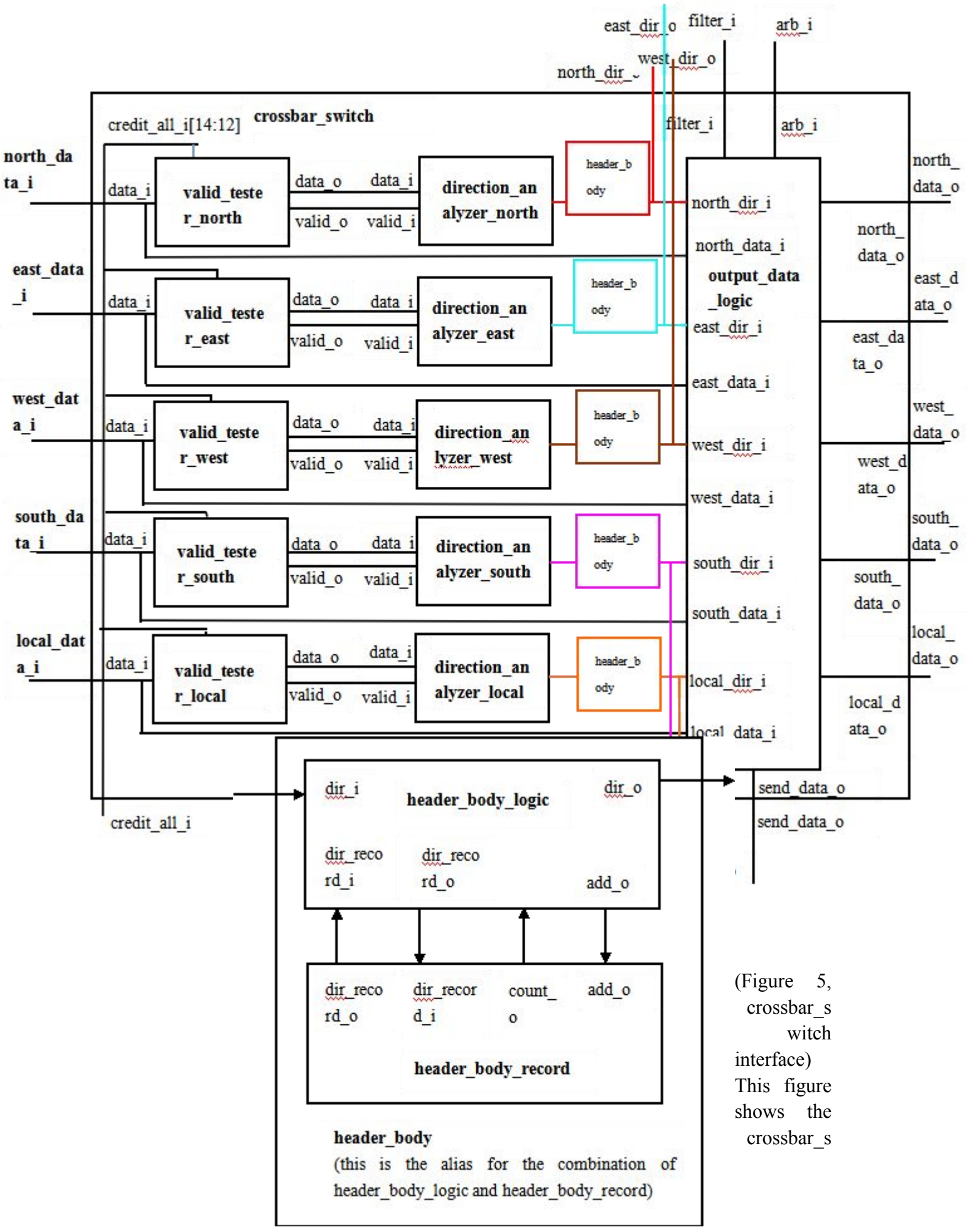
2.2.17 crossbar_switch

The crossbar_switch module interface information is shown below in Table 18 and Figure 5.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
crossbar_switch	clk	input	1	synchronous
	rst	input	1	reset
	north_data_i	input	17	Input data
	east_data_i	input	17	Input data
	west_data_i	input	17	Input data
	south_data_i	input	17	Input data
	local_data_i	input	17	Input data
	arb_i	input	1	Whether the arbitration is needed
	filter_i	input	5	Which input buffer is delayed by the arbiter
	credit_all_i	input	15	Get all the current value of

				five credit buffers of the same router
	north_data_o	output	17	Output data
	east_data_o	output	17	Output data
	west_data_o	output	17	Output data
	south_data_o	output	17	Output data
	local_data_o	output	17	Output data
	north_dir_o	output	6	Output direction information
	east_dir_o	output	6	Output direction information
	west_dir_o	output	6	Output direction information
	south_dir_o	output	6	Output direction information
	local_dir_o	output	6	Output direction information
	send_data_o	output	5	Represent which input buffer can send a flit out

(Table 18, crossbar_switch interface information)



(Figure 5, crossbar_switch interface)

This figure shows the crossbar_s

witch interfaces and its next-level sub-module interfaces.

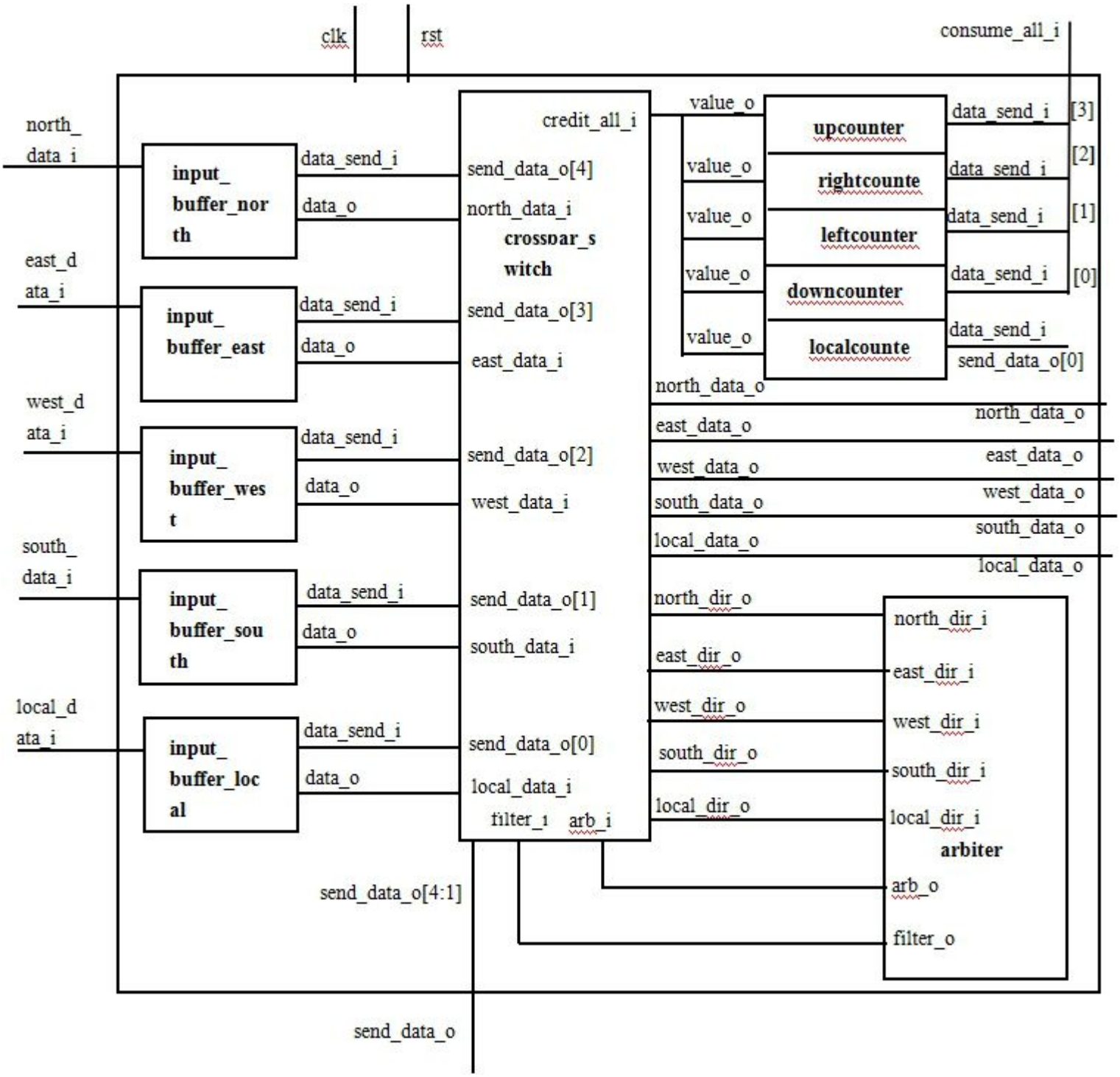
Description: this module can analyze the flits from the input buffer, communicate with the arbiter and finally know which input flits can be sent and send all the permitted flits to their desired output direction and inform other modules which flits are permitted to send at the same time.

2.2.18 router

The router module interface information is shown below in Table 19 and Figure 6.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
router	clk	input	1	synchronous
	rst	input	1	reset
	north_data_i	input	17	Input data
	east_data_i	input	17	Input data
	west_data_i	input	17	Input data
	south_data_i	input	17	Input data
	local_data_i	input	17	Input data
	consume_all_i	input	4	The four consume signals from neighbor routers(local input buffer does not send this signal)
	send_data_o	output	4	The four consume signals to neighbor routers
	north_data_o	output	17	Output data
	east_data_o	output	17	Output data
	west_data_o	output	17	Output data
	south_data_o	output	17	Output data
	local_data_o	output	17	Output data
	local_full_o	output	1	Inform outside world the local input buffer has no space

(Table 19, router interface information)



(Figure 6, router interface information)

This figure shows the router interfaces and its next-level sub-module interfaces. Each router has a unique location and this is recorded in the register inside the crossbar_switch. We temporarily use a logic type to represent it. For synthesizable purpose, we will make it more explicit further. Also, there are eight kinds of edge routers.

We sorted them according to Figure 1.

Type #	Including router	No need input/output	No need input buffer

	number	port of data_i	and credit counter
1	0	North, west	North, west buffer; up, left counter
2	3	North, east	North, east buffer; up, right counter
3	12	South, west	south, west buffer; down, left counter
4	15	South, east	South, east buffer, down, right counter
5	1, 2	north	North buffer, up counter
6	4, 8	west	West buffer, left counter
7	7, 10	east	East buffer, right counter
8	13, 14	south	South buffer, down counter

Description: this module is the basic node on the network to transfer and temporarily store data. The credit counters inside it are implemented by the up/down counters. The consume_all_i and send_data_o are signals to tell the neighboring routers to change their corresponding credit counter values and are pipelined with a extra register between each pair of neighboring two routers. The edge router owns less input/output ports, less input buffers, less credit counters, and simpler logic for purpose of optimization.

2.2.19 network

The network module interface information is shown below in Table 20.

Module Name	Signal Name	Input/Output	Signal Width	Functionality
network	data_0_i	input	17	Input data
	data_1_i	input	17	Input data
	data_2_i	input	17	Input data
	data_3_i	input	17	Input data
	data_4_i	input	17	Input data
	data_5_i	input	17	Input data
	data_6_i	input	17	Input data
	data_7_i	input	17	Input data
	data_8_i	input	17	Input data
	data_9_i	input	17	Input data
	data_10_i	input	17	Input data
	data_11_i	input	17	Input data
	data_12_i	input	17	Input data
	data_13_i	input	17	Input data
	data_14_i	input	17	Input data

	data_15_i	input	17	Input data
	clk	input	1	synchronous
	rst	input	1	reset
	data_0_o	output	17	Output data
	data_1_o	output	17	Output data
	data_2_o	output	17	Output data
	data_3_o	output	17	Output data
	data_4_o	output	17	Output data
	data_5_o	output	17	Output data
	data_6_o	output	17	Output data
	data_7_o	output	17	Output data
	data_8_o	output	17	Output data
	data_9_o	output	17	Output data
	data_10_o	output	17	Output data
	data_11_o	output	17	Output data
	data_12_o	output	17	Output data
	data_13_o	output	17	Output data
	data_14_o	output	17	Output data
	data_15_o	output	17	Output data
	local_full_o	output	16	Tell the outside world which router has no space to store new data

(Table 20, network interface information)

Description: this is the top level of the synthesizable module we decide to build. The input and output are from/to the outside world (from/to the testbench in the top module). It reflects how the routers are interconnected and how data is transferred.

2.2.20 top

The top modules consists three part: network, testbench, and the interface. The interface is designed to connect the testbench and network. The top generates the clock signal to network and testbench, and testbench generates other input signals for the network module. This top module is the highest level and has no input/output itself.

3. Micro-design Architecture

In this part we list some explanation of input/output signals, detailed logic equations and FSM diagrams for some of the modules appeared in part 2 above.

3. 1 Module comparator

In the direction_analyzer module we use two 4-bit comparator to calculate the desired output of a data flit. The meaning of comparator output is shown below in table 21.

State	Comparison Result
State “00”	Equal
State “01”	Data1_i is larger
State “10”	Data1_i is smaller
State “11”	Illegal, not appear

(Table 21, Meaning of comparator output)

Logic equation for result_o (comparator output) :
result_o[0] = data1_i[3]*data2_i[2] + data1_i[3]*data2_i[1] + data1_i[3]*data2_i[0] + data1_i[2]*data2_i[1] + data1_i[2]*data2_i[0] +data1_i[1]*data2_i[0] ;
result_o[1] = data1_i[0]*data2_i[3] + data1_i[0]*data2_i[2] + data1_i[0]*data2_i[1] + data1_i[1]*data2_i[3] + data1_i[1]*data2_i[2] +data1_i[2]*data2_i[3] ;
(data1_i and data2_i are module inputs)

3. 2 Module com_dir_logic

Meaning of dir_o (com_dir_logic output): it is a 6-bit length signal. The first bit is the validation bit. When this module is not needed or some exceptions happen, the first bit will be set high. Normally it is low. The second bit represents the desired output direction is north(up) when sets high. The third bit similarly represents east(right); the fourth bit represents west(left); the fifth bit represents south(down), the last bit represents local.

Relationship between input and output is shown in table 22 below:

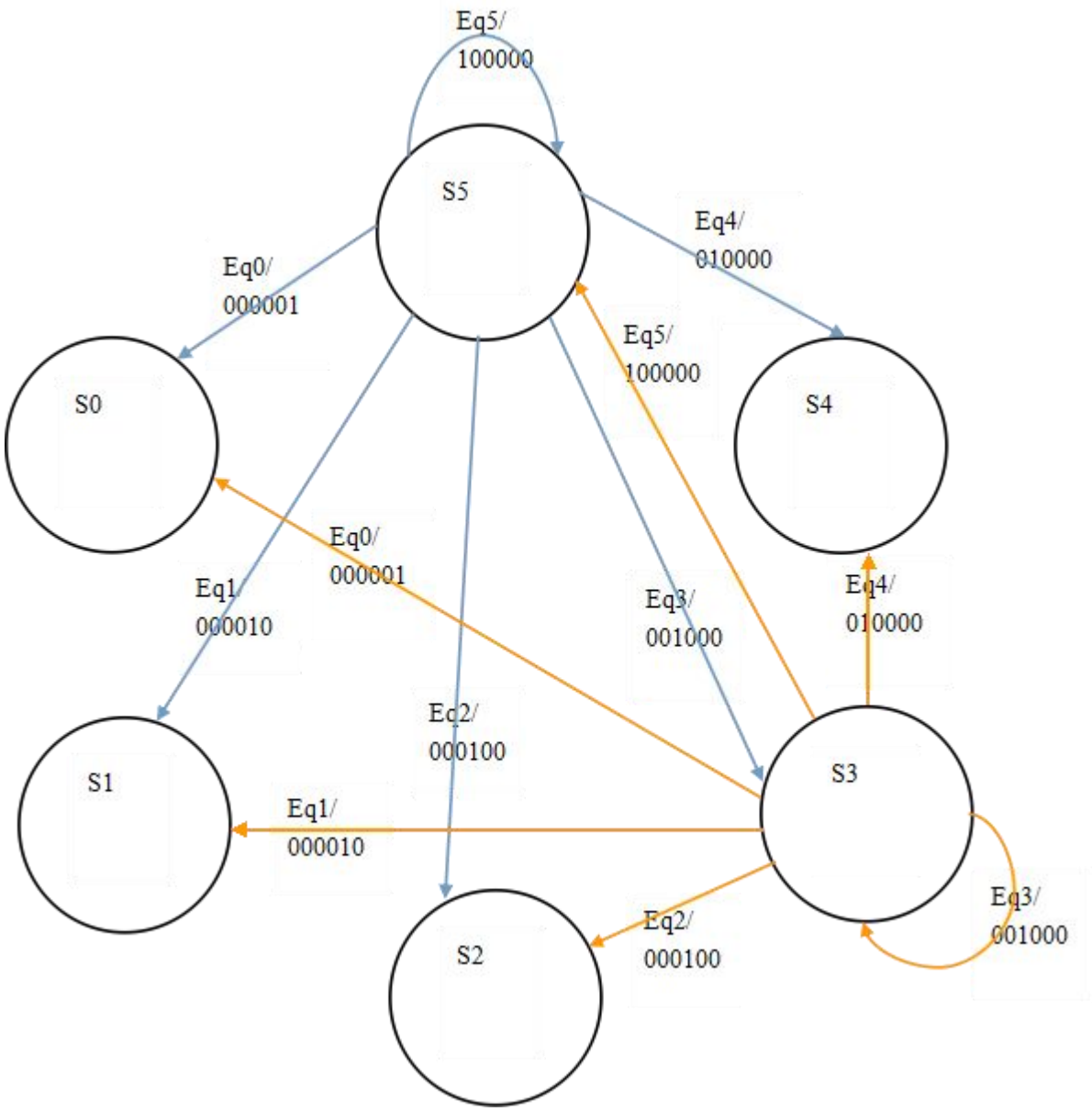
com1_i	00	01	10	00	01	10	00	01	10
com2_i	00	00	00	01	01	01	10	10	10
dirction	L	E	W	S	S	S	N	N	N
dir_o	000001	001000	000100	000010	000010	000010	010000	010000	010000

(Table 22, com_dir_logic input/output relationship)

Logic equation for dir_o:
Eq5:dir_o[5]=valid_i';
Eq4:dir_o[4]=com2_i[1]*com2_i[0]*valid_i;
Eq3:dir_o[3]=com1_i[1]'*com1_i[0]*com2_i[1]'*com2_i[0]'*valid_i;
Eq2:dir_o[2]=com1_i[1]*com1_i[0]'*com2_i[1]'*com2_i[0]'*valid_i;
Eq1:dir_o2]=com2_i[1]'*com2_i[0]*valid_i;

Eq0:dir_o[2]=com1_i[1]'*com1_i[0]'*com2_i[1]'*com2_i[0]'*valid_i;
(com1_i, com2_i and valid are module inputs)

FSM for module com_dir_logic:



(Figure 7, FSM for com_dir_logic)

Each state has all the 6 output arrows, each of which points to another state including itself. And all the arrows pointing to the same state have the same input equation and output value. So there will be totally 36 arrows. Because of the limited space, only 12 of them are shown on this picture.

3.3 Module chooser

For each chooser that targets one output direction, there is a related 5-bit location register to record which input buffer was the source for last passing flit. The detailed explanation is shown in table 23 below.

(Table 23, meaning for chooser-related register value)

Record value	00000	10000	01000	00100	00010	00001
meaning	No data has come before,	north	east	west	south	local

This register communicates with the chooser by providing the chooser input signal last_dir_i and receiving the output signal dir_o.

Logic equation for module chooser:
(left side items of the equations are outputs, and right side items are inputs)

```
arb_o = north_hit_i * east_hit_i + north_hit_i * west_hit_i + north_hit_i * south_hit_i
+ north_hit_i * local_hit_i |
        east_hit_i * west_hit_i + east_hit_i * south_hit_i + east_hit_i * local_hit_i +
        west_hit_i * south_hit_i + west_hit_i * local_hit_i +
        south_hit_i * local_hit_i;
filter_o[4] = last_dir_i[4] * (east_hit_i + west_hit_i + south_hit_i + local_hit_i) +
                last_dir_i[3] * (west_hit_i + south_hit_i + local_hit_i) +
                last_dir_i[2] * (south_hit_i + local_hit_i) +
                last_dir_i[1] * (local_hit_i);
filter_o[3] = (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
              last_dir_i[0]') * north_hit_i +
                last_dir_i[3]' * (west_hit_i + south_hit_i + local_hit_i + north_hit_i) +
                last_dir_i[2]' * (south_hit_i + local_hit_i + north_hit_i) +
                last_dir_i[1]' * (local_hit_i + north_hit_i) +
                last_dir_i[0]' * (north_hit_i);
filter_o[2] = (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
              last_dir_i[0]') * (north_hit_i + east_hit_i) +
                last_dir_i[4] * east_hit_i +
                last_dir_i[2] * (south_hit_i + local_hit_i + north_hit_i + east_hit_i) +
                last_dir_i[1] * (local_hit_i + north_hit_i + east_hit_i) +
                last_dir_i[0] * (north_hit_i + east_hit);
filter_o[1] = (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
              last_dir_i[0]') * (north_hit_i + east_hit_i + west_hit_i) +
                last_dir_i[4] * (east_hit_i + west_hit_i) +
                last_dir[3] * west_hit_i +
                last_dir[1] * (local_hit_i + north_hit_i + east_hit_i + west_hit_i) +
                last_dir[0] * (north_hit_i + east_hit + west_hit_i);
```

```
filter_o[0] = (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
               last_dir_i[0]') * (north_hit_i + east_hit_i + west_hit_i + south_hit_i) +
                  last_dir_i[4] * (east_hit_i + west_hit_i + south_hit_i) +
                  last_dir_i[3] * (west_hit_i + south_hit_i) +
                  last_dir_i[2] * south_hit_i +
                  last_dir_i[0] * (north_hit_i + east_hit + west_hit_i + south_hit_i);
eq0: dir_o[4] = north_hit_i * (
                    (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
                     last_dir_i[0]') +
                     last_dir_i[4] * (east_hit_i' * west_hit_i' * south_hit_i' * local_hit_i')
                     + last_dir_i[3] * (west_hit_i' * south_hit_i' * local_hit_i') +
                    last_dir_i[2] * (south_hit_i' * local_hit_i')
                    last_dir_i[1] * local_hit_i'
                    last_dir_i[0]
               );
eq1: dir_o[3] = east_hit_i * (
                    (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
                     last_dir_i[0]') * north_hit_i' +
                    last_dir_i[4] +
                     last_dir_i[3] * (west_hit_i' * south_hit_i' * local_hit_i' * north_hit_i')
                     + last_dir_i[2] * (south_hit_i' * local_hit_i' * north_hit_i') +
                    last_dir_i[1] * (local_hit_i' * north_hit_i') +
                    last_dir_i[0] * north_hit_i'
               );
eq2: dir_o[2] = west_hit_i * (
                    (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
                     last_dir_i[0]') * (north_hit_i' + east_hit_i') +
                    last_dir_i[4] * east_hit_i' +
                    last_dir_i[3] +
                     last_dir_i[2] * (south_hit_i' * local_hit_i' * north_hit_i' + east_hit_i')
                     + last_dir_i[1] * (local_hit_i' * north_hit_i' + east_hit_i') +
                    last_dir_i[0] * (north_hit_i' * east_hit_i')
               );
eq3: dir_o[1] = south_hit_i * (
                    (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
                     last_dir_i[0]') * (north_hit_i' + east_hit_i' + west_hit_i) +
                    last_dir_i[4] * (east_hit_i' * west_hit_i') +
                    last_dir_i[3] * west_hit_i' +
                    last_dir_i[2] +
                     last_dir_i[1] * (local_hit_i' * north_hit_i' + east_hit_i' + west_hit_i)
                     + last_dir_i[0] * (north_hit_i' * east_hit_i' + west_hit_i)
               );
eq4: dir_o[0] = local_hit_i * (
                    (last_dir_i[4]' * last_dir_i[3]' * last_dir_i[2]' * last_dir_i[1]' *
```

```
last_dir_i[0]') * (north_hit_i' + east_hit_i' + west_hit_i + south_hit_i') +
    last_dir_i[4] * (east_hit_i' * west_hit_i' + south_hit_i') +
    last_dir_i[3] * (west_hit_i' * south_hit_i' ) +
    last_dir_i[2] * south_hit_i' +
    last_dir_i[1] +
    last_dir_i[0] * (north_hit_i' * east_hit_i' + west_hit_i + south_hit_i')
);
```

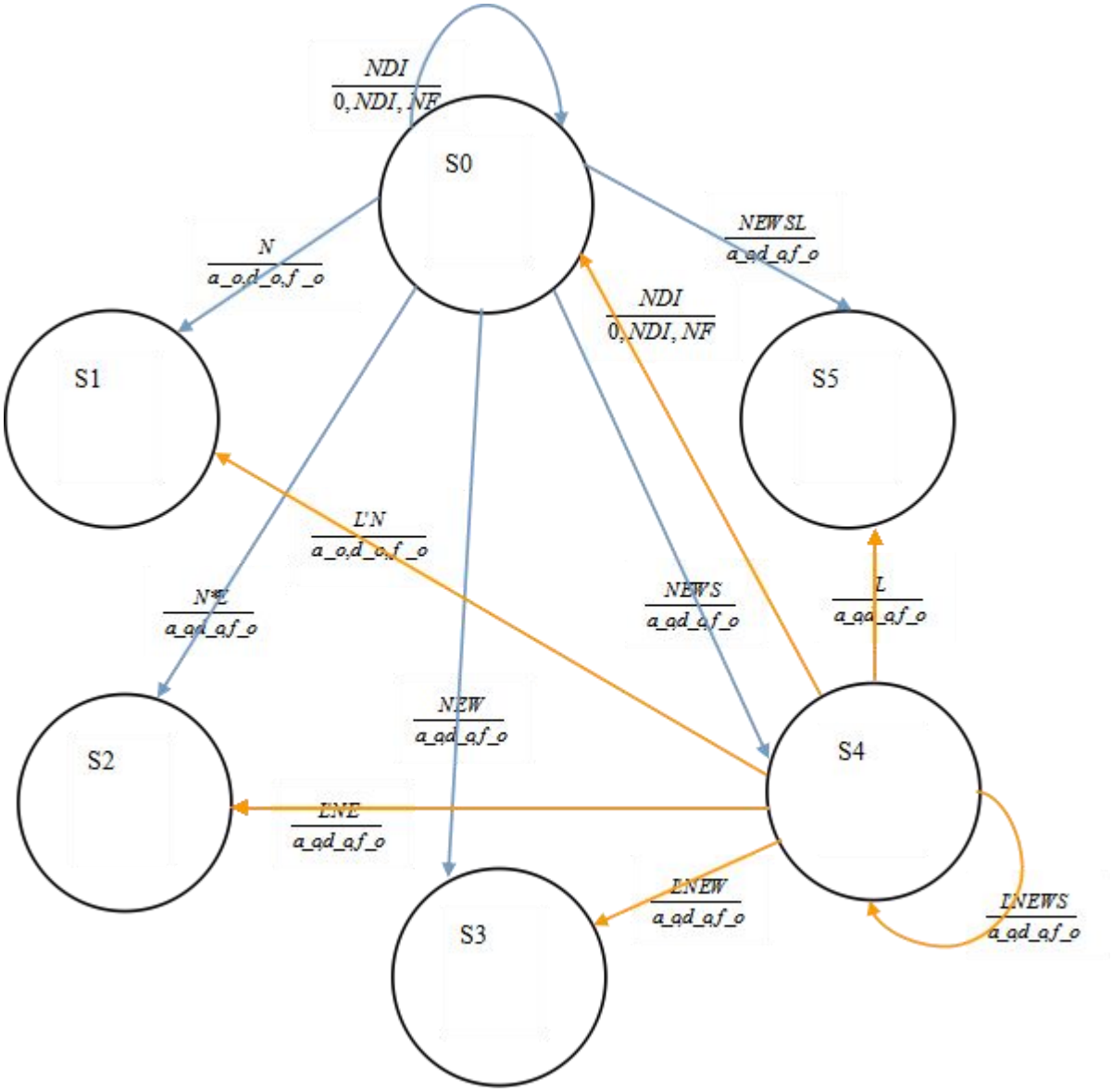
FSM for module chooser:

Last_dir_i	no data/s0(ND)	N/s1	E/s2	W/s3	S/s4	L/s5
state	00000	10000	01000	00100	00010	00001

(Table 24, the state meaning)

The condition form for the FSM:

If dir[4:0] = 00000, we call it no direction(NDI)
If filter[4:0] = 00000, we call it no filter(NF)
a_o is according to arb_o equation
d_o is the combination of the dir equations
f_o is the combination of the filter eauations
north_hit_i(N) east_hit_i(E) west_hit_i(W)
south_hit_i(S) local_hit_i(L)



(Figure 8, FSM for chooser)

Each state has all the 6 output arrows, each of which points to another state including itself. And all the arrows pointing to the same state have the same input equation and output value. So there will be totally 36 arrows. Because of the limited space, only 12 of them are shown on this picture. The states explanation are shown in Table 24.

3.4 Module arbiter_logic

Arbiter_logic module's function is to combine the results from the five chooser. The basic element in it is OR gate.

Logic equation for module arbiter_logic:
arb_o = arb1_i + arb2_i + arb3_i + arb4_i + arb5_i;
filter_o[0] = fil1_i[0] + fil2_i[0] + fil3_i[0] + fil4_i[0] + fil5_i[0];

filter_o[1] = fil1_i[1] + fil2_i[1] + fil3_i[1] + fil4_i[1] + fil5_i[1];
filter_o[2] = fil1_i[2] + fil2_i[2] + fil3_i[2] + fil4_i[2] + fil5_i[2];
filter_o[3] = fil1_i[3] + fil2_i[3] + fil3_i[3] + fil4_i[3] + fil5_i[3];
filter_o[4] = fil1_i[4] + fil2_i[4] + fil3_i[4] + fil4_i[4] + fil5_i[4];
(items at the right side of the equations are inputs)

3.5 Module valid_tester

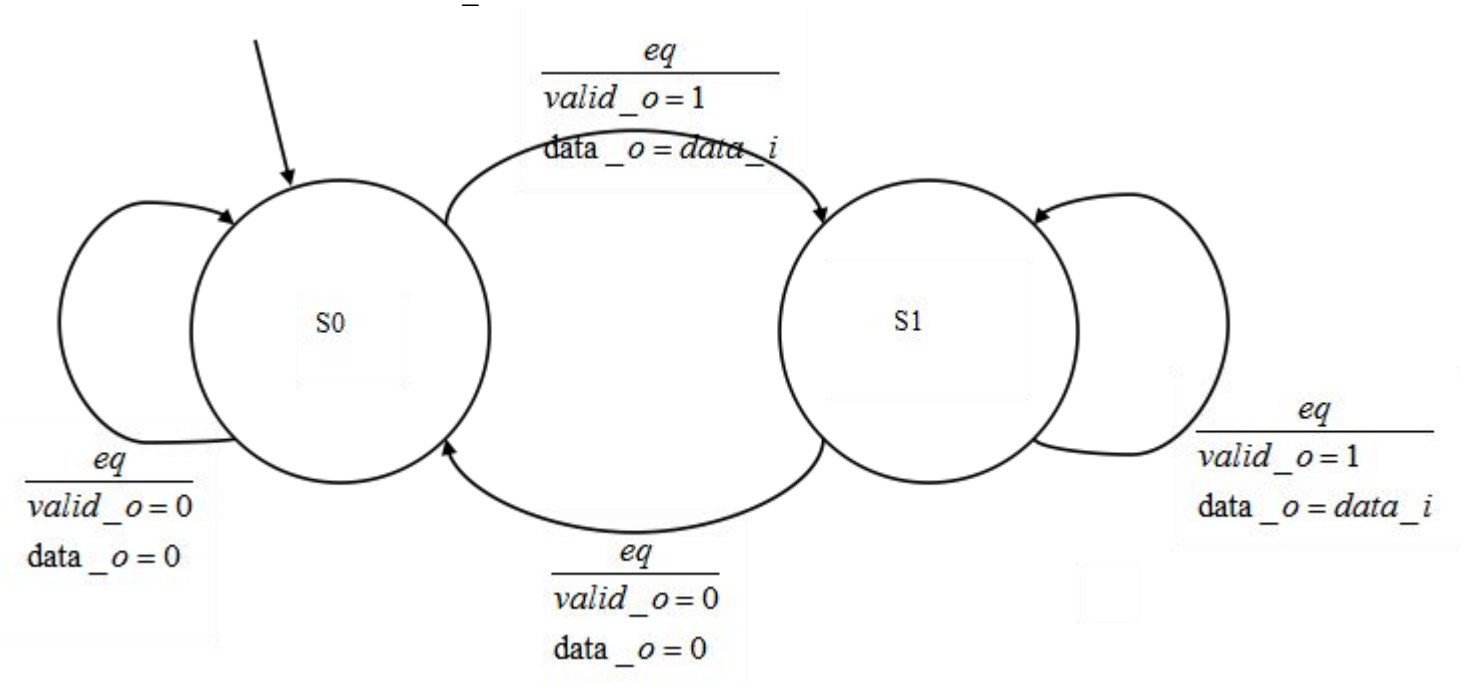
The input credit_i means the count value, it should have seven kinds, six of which representing value from 0 to 5 and the last one represents error as shown in Table 25.

(Table 25, record value explanation)

Record value	000	001	010	011	100	101	110
meaning	0	1	2	3	4	5	error

Logic equation for output valid_o:
eq: valid_o = data_i[16] * (credit_i[0] + credit_i[1]*credit_i[2]' + credit_i[1]' *credit_i[2])

FSM for module valid_tester:



(Figure 9, FSM for valid_tester)

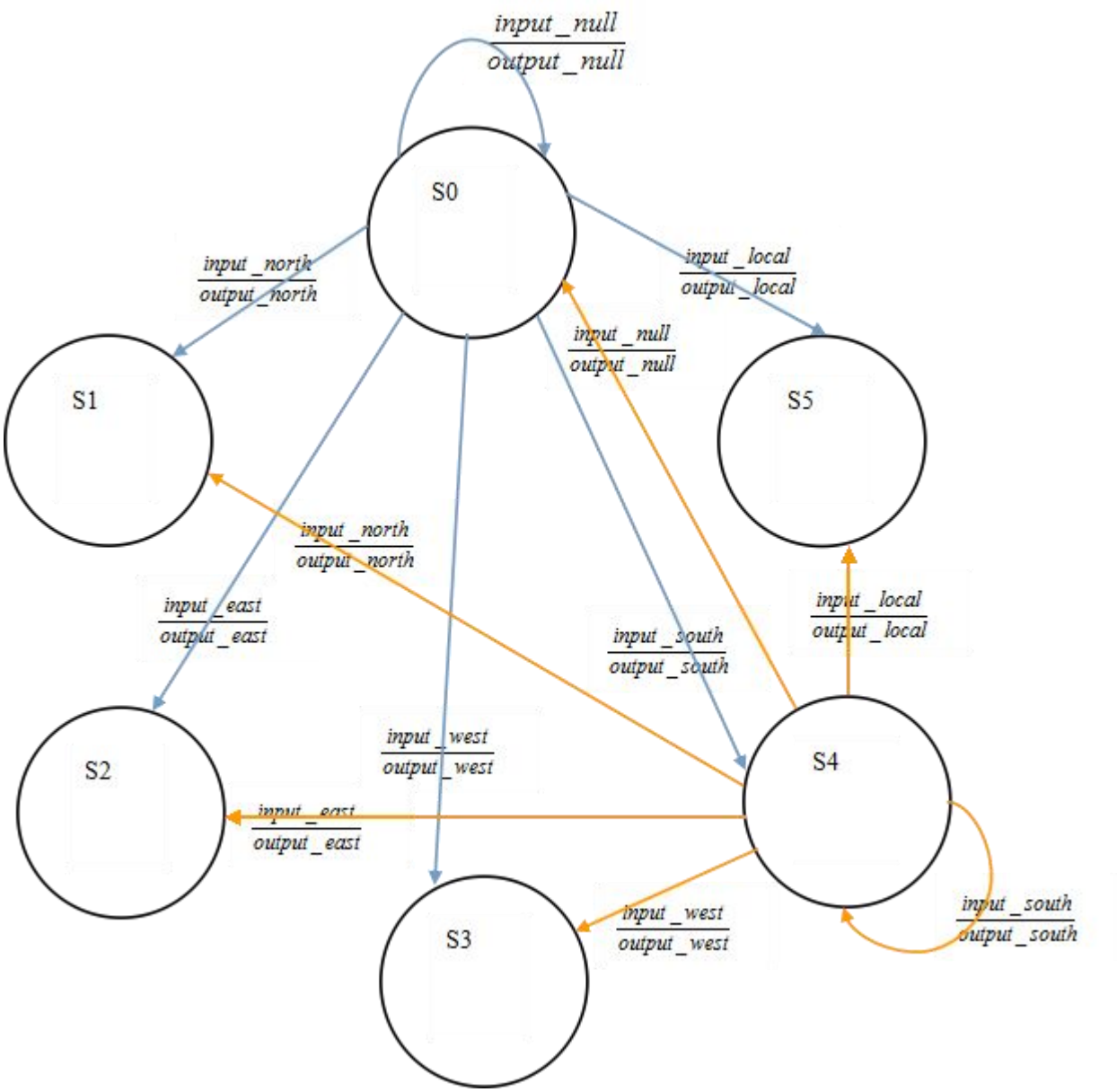
3.6 Module output_data_logic

In this module we use five intermediate logic types, each to record whether for a certain output direction there is a new data flit to come through. They are send_north, send_east, send_west, send_south, and send_local. If they are calculated to be zero,

which means no data flit is requesting the output direction, then for that certain output direction the output data should be reset to zero in case of sending the same data flit twice.

Logic equation for intermediate data send and output send_o:
send_north = north_dir_i[4] + east_dir_i[4] + west_dir_i[4] + south_dir_i[4] + local_dir_i[4]
send_east = north_dir_i[3] + east_dir_i[3] + west_dir_i[3] + south_dir_i[3] + local_dir_i[3]
send_west = north_dir_i[2] + east_dir_i[2] + west_dir_i[2] + south_dir_i[2] + local_dir_i[2]
send_south = north_dir_i[1] + east_dir_i[1] + west_dir_i[1] + south_dir_i[1] + local_dir_i[1]
send_local = north_dir_i[0] + east_dir_i[0] + west_dir_i[0] + south_dir_i[0] + local_dir_i[0]
send_data_o = {send_north, send_east, send_west, send_south, send_local}

FSM for the north_data_i of the module output_data_logic:
(east_data_i, west_data_i, south_data_i, and local_data_i FSMs are similar.)



(Figure 10, FSM for output_data_logic)

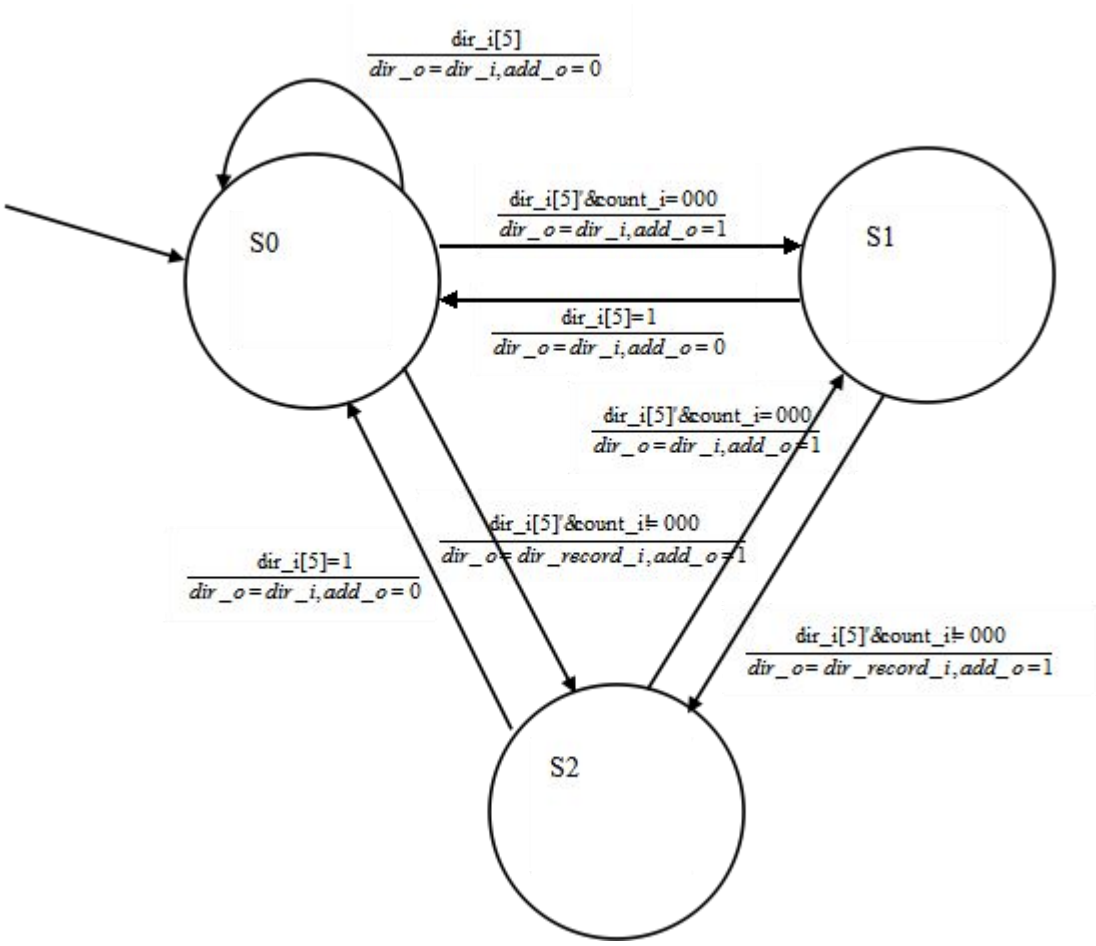
Each state has all the 6 output arrows, each of which points to another state including itself. And all the arrows pointing to the same state have the same input equation and output value. So there will be totally 36 arrows. Because of the limited space, only 12 of them are shown on this picture.

The conditions in the FSM picture is calculated with following equations:
input_null: (arb_i' + arb_i * filter_i[4]')'
output_null: no data is sent out
input_north: north_dir_i[4]*filter_i[4]'
output_north: north_data_o = north_data_i, send_data_o[4] = send_north = 1
input_east: north_dir_i[3]*filter_i[4]'

output_east: east_data_o = north_data_i, send_data_o[3] = send_east = 1
input_west: north_dir_i[2]*filter_i[4]'
output_west: west_data_o = north_data_i, send_data_o[2] = send_west = 1
input_south: north_dir_i[1]*filter_i[4]'
output_south: south_data_o = north_data_i, send_data_o[1] = send_south = 1
input_local: north_dir_i[0]*filter_i[4]'
output_local: local_data_o = north_data_i, send_data_o[0] = send_local = 1

3.7 Module header_body_logic

This module is combinational logic and tells apart body flits from header flits.
FSM for this module:



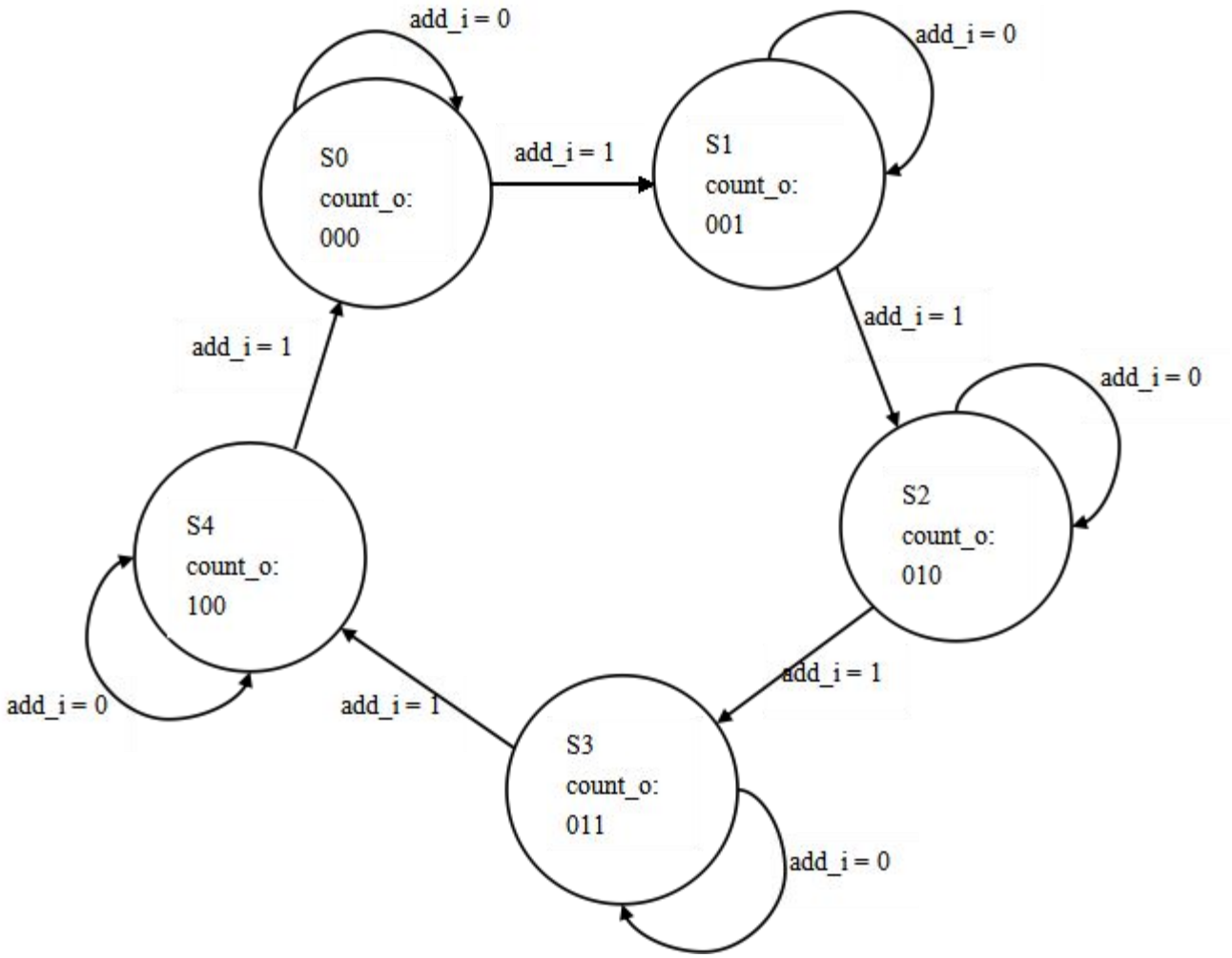
(Figure 11, FSM for header_body_logic)

3.8 Module header_body_record

This is a sequential module with function of recording numbers of flits come through and the header flit's desired direction.

Logic equation for output count_o of next clock cycle is:
count_next[2] = add_i' * count[2] + add_i * count[1] * count[0]
count_next[1] = count[1] * count[0]' + add_i' * count[1] * count[0] + add_i * count[1]' * count[0]
count_next[0] = add_i'*count[0] + add_i * count[2]' * count[0]

FSM for this module:



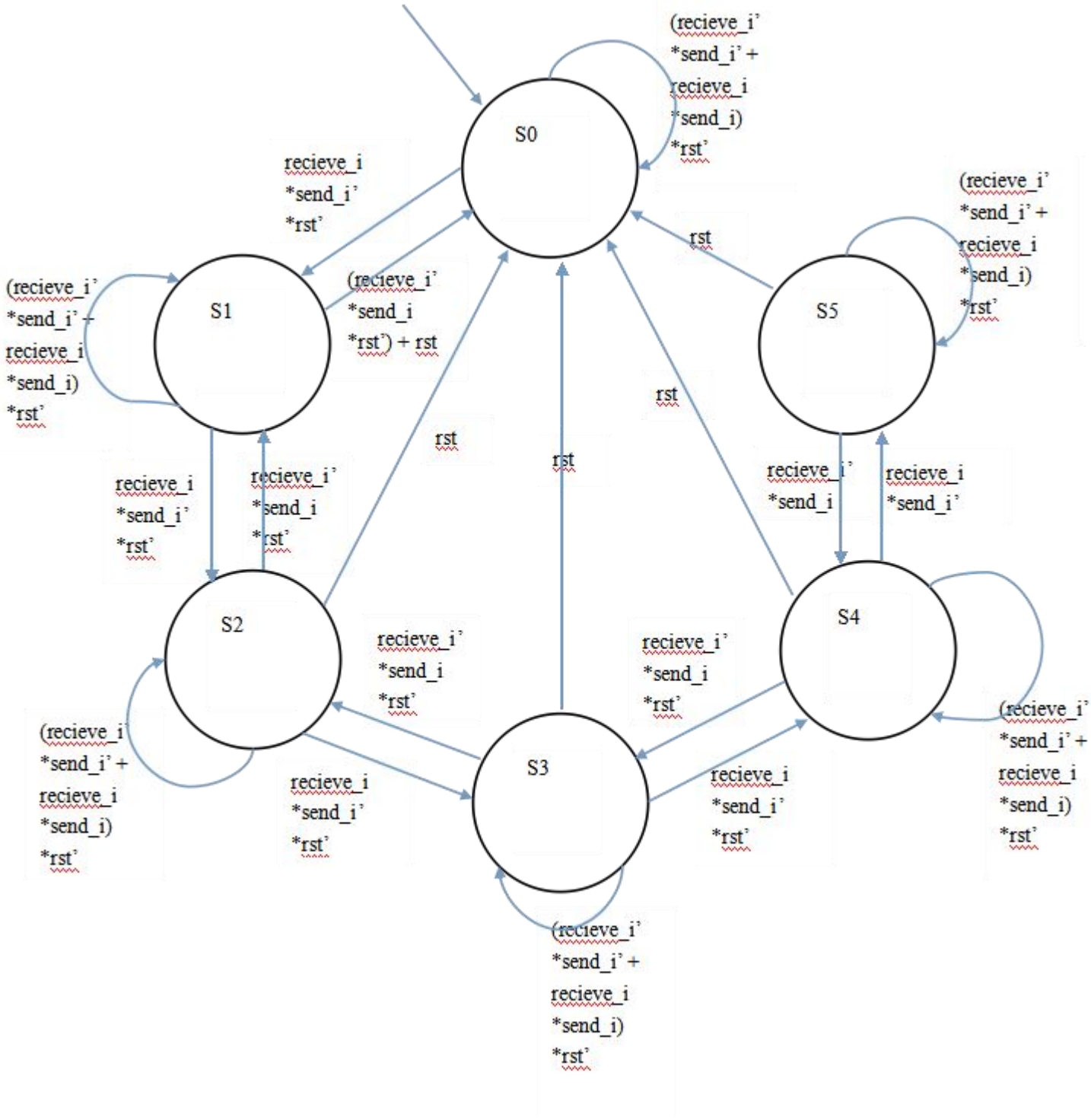
(Figure 12, FSM for header_body_record)

3.9 Module buffer_storage

This module records the valid data flits and is the main part of a input buffer.

FSM for this module:
S0 means this module currently contains no flit
S1 means this module currently contains 1 flit
S2 means this module currently contains 2 flits

S3 means this module currently contains 3 flits
S4 means this module currently contains 4 flits
S5 means this module currently contains 5 flits



(Figure 13, FSM for buffer_storage)

4. Validation strategy

4.1 Verification framework

The module network is implemented with synthesizable system verilog language. When do the verification, we design a testbench with C-like system verilog language. The testbench produces and sends out test stimulus to the network module, receives results from it and at the same time generates a golden result itself. Then the testbench will compare the golden results with the results got from network module to see the difference of them. An interface is designed to connect the network module and the testbench. A top module will generate the global clock signal and it contains each one instance of network module, interface, and the testbench respectively.

4.2 Parameters

For the testing process we use parameters to determine the testing environment and targets to be tested. The parameters are written in the file config.txt and read by the testbench. With the cooperation of the testbench and parameter predefined values, we can generate constrained-random stimulus to test specific parts of the network module.

Each parameter has its special meaning which can be interpreted from its names. Also, each parameter has a valid range. But during the verification process sometimes we will assign parameters out of their valid range to intentionally create errors and exceptions for specific test purposes.

4.3 Module level verification

Assertions: we plan to use immediate assertions to test the correctness of some modules. For example, this will be used to check the one hot state machines. In the module com_dir_logic, the output should only has one high bit, and that feature is tested with assertion. Also, for some control signals, it should always be 0 or 1 such as the valid signal in the direction_analyzer module. We will use assertions to check after the reset signal if there is x or z (unknown) value on those signals.

Special delays: We plan to use special delays to test the functionality of storage of some modules such as the input buffer. After a long period of intended delays, whether the data flits is still stored in it.

Core modules: for some important and complex module in our design such as the crossbar_switch we plan to test them separately with specific designed testbenches before the full design testing. Since if we cannot make sure they function well and directly start testing the full design, it will be a great pain to locate bugs. Also, we can test whether the right destination is calculated out by the direction_analyzer as another example.

4.4 Full design testing

We will set different parameter values to test incrementally the functionalities of

our network module.

4.4.1 Test clock and basic reset

The first step to test the full design is to run an empty clock with reset asserted high for a few cycles. When reset is high check the values for output interfaces. If valid data appears from the output ports, something wrong must happen. Also we can check whether the reset is functioning in some submodules with “display” command such as the value of credit counters.

4.4.2Test input control interfaces

The valid bit inside the data flit is kind of control signal. It determines whether the values on the interface is a valid data flit to be transferred across the network. We can control corresponding testbench input flits by changing the values of parameters data_i_density(i = 0, 1, 2,...15). Then we run the clock for a few cycles with parameters data_i_density assigned all zero. If valid data appears from the output ports, something wrong must happen.

4.4.3 Test input data interfaces

The input data range is set up through the parameters parameters data_i_range (i = 0, 1, 2,...15). Then we run the clock for a few cycles with parameters data_i_density assigned some certain range. If data appears from the output ports with the value outside of that range, something wrong must happen.

4.4.4 Test readable/writable

Randomly give some routers input data and then check whether the router will accept that and output them to their neighboring routers or even directly output them to their local output direction.

4.4.5 Test transfer function

Send only one data flit to the network module through any one of the input ports. Then examine whether that flit will be sent out to the correct/desired output port; Send the same data flit continuously one of the input module, examine whether the correct/desired output port will also send out the same data flit continuously.

4.4.6 Test arbitration function

Continuously send two different data flits with different desired output points to the network module at the same time and make sure they will go across the same router on the path from their input points to output points. Check the output to see if the flits are delayed by the arbitration and how the arbitration really works; then send more kinds of flits continuously and simultaneously to the network module to further test arbitration.

4.4.7 Test corner cases

(a) All the routers form a big chain to transfer data and all the corresponding input

buffers on that long path are full. (critical path is long at this time)
(b) One router continuously send out data.
(c) The input buffer becomes full.
(d) Input flits still come when the local input buffer is full
(e) The credit counter becomes empty.

4.4.8 Incremental development

Set more and more data_i_density and data_i_range parameters valid, increase the testing complexity and coverage of the module network. See the compare results in the testbench to check the correctness of the network module.

4.4.9 Test all randomly

We will try different combination of parameters in the config file, tested all the functions of the network module at the same time in this part. This time run more than 100000 clock cycles for each combiantion type of parameters.

4.4.10 Test error-tolerance

Intentionally give bad/invalid input to the network module. e.x. A data flit has the same input and desired output port; a data flit has no desired output point; a data flit has multiple desired output points and so on.

4.5 Testbench Structure

The testbench generates the network input, receive its output and compare the received ones with the golden ones to judge the real functionality of our network design It includes several files as listed below.

File name	File description
config.txt	Set up the parameter of the testbench data
environment.sv	Read the parameters into the testbench
transaction.sv	Produce random data with some predefined limitations
checker.sv	Compare the actual result got from the network module with the theoretical results got from software calculation
bench.sv	Execute the testbench, do the loops of testing

(Table 26, the testbench files)

5. Performance Estimates

5.1 Latency

For a router in one clock cycle, a data flit in the input buffer can be transferred to the output wire that connects two neighboring routers (assuming the clock period is long enough for all the combinational logic functioning). Then for next clock posedge, another router can receive that data flit into its input buffer for further transferring. In one clock cycle, if a flit from input buffer wants to go outside to the local output direction and is not delayed by the arbiter, it will appear on the wire that connects the router output and outside world. So for a single flit to transfer, on its transfer pass each router will take one clock cycle time. For if a flit needs to go through three routers and no need to wait for other flits, it will be received by the outside world at the destination router wire after three clock cycles. Transfer latency for one router is one clock.

The input buffer can hold at most five flits, and follow first-in first-out mechanism. So a newly-come flit will not be served but wait in the input buffer for its turn. The waiting latency is determined by the flit number currently on that router's input buffer. The worst case is that four more flits have already been waiting in the input buffer.

The arbiter may delay a flit's transfer. It can delay at most three clock cycles on one router since for an output direction at most four flits requests to go through it. So the maximum latency due to the arbiter on a router is three clock cycles.

For a router, if in a clock cycle it sends out a data flit, it will send a signal to inform that to its neighboring router, which will be received by the neighboring router at next clock positive edge. In that way under high density data flits situation, if several routers form a chain to send flits, they cannot send the flits at the same time though that is OK, because the routers on the chain will be informed one by one instead of informed simultaneous. That increases latency but avoids critical path and decreases the complexity of the combintional logic.

For the network, the stall strategy chosen is pipelined stall. In terms of router pattern and data transmission structure, it would be easier and sufficient to program with pipelined stall, since such strategy should totally reduce programming efforts by avoiding complex combinational logic involving several routers and the crossbar_switch as well as the arbitration deals only with the first data in the input buffer and the related credit counter, while global stall needs a lot more programming to specify the stall process.

The actual latency is highly related to the flits density on the network, and that is more obvious if the transfer path is long.

We can assume the flit density is on average, when a new flit comes in, it becomes the third flit waiting in the input buffer. Each flit will be delayed one clock cycle in a router. That certain flit's transfer path goes through four routers (minimum is two,

maximum is seven). Then for each router that flits takes six clock cycles, and the final latency is 4*6 = 24 clock cycles.

5.2 Bandwidth

In terms of router, each router transfer one data flit in one clock cycle and at most transfer five data flits simultaneously. So for a router the maximum bandwidth is five flits/(clock cycle* router).

In terms of the network, there are totally 24 flit transfer bunch of wires connecting between two routers plus 32 (16 inputs, 16 outputs) flit transfer bunch of wires connecting between network module and outside world. So for the network module the maximum bandwidth is 56 flits/(clock cycle* router)

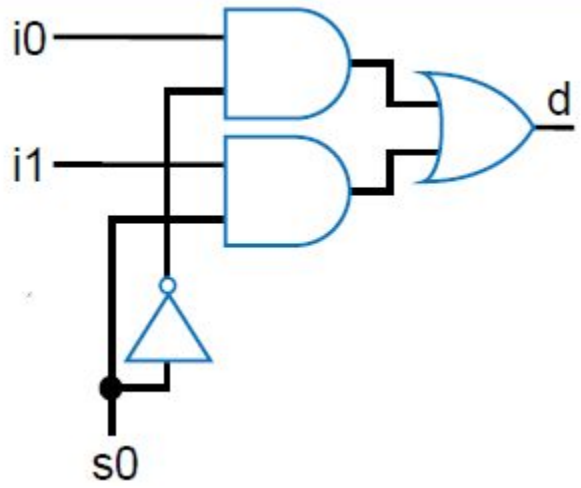
The average bandwidth is highly related to the data flits density currently on the network.

6. Area Estimates

To estimate space resources used by our network module, we use the number of basic gates used to represent. The basic gates include AND, OR, and NOT gate.

6.1 A D flip flop with reset signal will use one NOT gate, one AND gate for reset signal, and five more NAND gate for the flip flop itself. (we know from the website below)
http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/dflipflop.html
One NAND gate uses one NOT gate and one AND gate. We use enabled flip flop, so it needs one more NAND gate. So generally one flip flop we use in our design consumes 5 AND gates and 5 NOT gates

6.2 A 1-bit two-to-one mux uses 2 AND gate, 1 OR gate, and 1 NOT gate.



(Figure 14, mux gate level) (VE270, Gang Zheng, Lec4, SJTU)

Then we can estimate the gates used for each

sub-module in our design with flip flop, mux, and previous calculated logic equations in the micro architecture part.

6.3 Module comparator: we use 4-bit data comparator in our design. The if-else statement is realized with MUX, in this game a 12-1 MUX consists of 3 4-1 MUX and 2 2-1 MUX thus gates needed are 8 NOT gates, 16 AND gates and 5 OR gates.

6.4 Module register: register consists of some logics, EFF and if-else statement, which if-else statement is reliazed with 5-1 MUX, thus the unit needs 8 NOT gates, 13 AND gates and 3 OR gates for 1 bit data.

6.5 Module counter: For counter module it consists some logics, if-else statement and case statement which are realized with a 6-1 MUX and a 5-1 MUX for 3 bits data, thus it consists 36 NOT gates, 54 AND gates and 21 OR gates

6.6 Module buffer_storage: it mainly includes 5 17-bit registers so it has 85 D flip flops. That uses 510 NOT gate and 595 AND gate. We also need a flip flop to record a certain register is being used or not. That needs extra 35 AND gate and 30 NOT gate. So in total we use 630 AND gate and 540 NOT gate.

6.7 Module chooser: since this module is all combinational logic, we can derive the gates used from the logic equations in the micro architecture part. It uses 90 NOT gates, 235 AND gates and 105 OR gates

6.8 Module com_dir_logic: this part consists of a 7-1 MUX which consists of 2 4-1 MUX and the gates used for 6 bits are 24 NOT gates, 48 AND gates and 12 OR gates.

6.9 Module direction_analyzer: it contains two 4-bit comparators, one com_dir_logic, and one 4-bit register (to record the router's location), thus it contains 68 AND gates, 20 OR gates, and 36 NOT gates.

6.10 Module header_body_logic: it contains one 1-bit two-to-one mux for output add_o, one 6-bit two-to-one mux for output dir_o, one 6-bit two-to-one mux for output dir_record_o, and also one OR gate to judge dir_i[5], three OR gates to judge whether count_i = 000; thus it totally uses 26 AND gate, 17 OR gates, and 13 NOT gates.

6.11 Module header_body_record: it contains one 6-bit register to record dir_i, one 3-bit register to record count_o, and also combinational logic to calculate next state's count_o whose logic equation is in the micro architecture part. It totally contains 74 AND gates, 4 OR gates, and 60 NOT gates.

6.12 Module arbiter_logic: from the logic equation we know it contains only 24 OR gates.

6.13 Module arbiter: it contains five choosers, five 5-bit registers, and one arbiter_logic sub-module, thus it totally uses 650 NOT gates, 1500 AND gates and 624 OR gates .

6.14 Module valid_tester: it uses a logic equation to calculate valid_o, and uses a 17-bit two-to-one mux to select data_o. Thus it totally consumes 40 AND gates, 20 OR gates, and 20 NOT gates.

6.15 Module output_data-logic: it uses five logic equations to calculate output send_data_o, five 17-bit six-to-one mux for output data, and related combinational logic to deal with input dir_i (north, east, west, south, local) signal as shown in FSM conditions. It totally uses 856 AND gates, 446 OR gates, and 433 OR gates.

6.16 Module crossbar_switch: it contains five valid_testers, five direction_analyzers, five header_body_logic, five header_body_record, and one output_data_logic. Thus it totally uses 1736 AND gates, 658 OR gates, and 1011NOT gates.

6.17 Module input_buffer: this module contains one buffer_storage and one valid_tester, thus it uses 670 AND gates, 20 OR gates, and 560 NOT gates.

6.18 Module router: typically a router contains five input buffers, five credit counters (made of module counter), one crossbar_switch and one arbiter, thus it totally contains 9856 AND gates, 1487 OR gates, 4641 NOT gates. For edge routers (type one) with number 1, 2, 4, 7, 8, 11, 13, 14 in Figure 1, they can have one less input buffer and one less credit counter compared with the typical router, thus they use 6132 AND gates, 1446 OR gates, and 4045 NOT gates. For edge routers (type two) with number 0, 3, 12, 15 in Figure 1, they can have two less input buffers and two less credit counters compared with the typical router, thus they use 5408 AND gates, 1405 OR gates, and 3449 NOT gates.

6.19 Module network: it contains four typical routers, eight type one edge routers, and four type two edge routers, thus it totally uses 110112 AND gates, 23136 OR gates, and 81416 NOT gates.

The summary Table 27 below shows the gates used for each module.

Module name	AND gate	OR gate	NOT gate
D flip flop	5	0	5
1-bit two-to-one mux	2	1	1
4-bit comparator	16	5	8
counter	54	21	36
Buffer_storage	630	0	540

chooser	235	105	90
Com_dir_logic	48	12	24
Direction_analyzer	68	20	36
Header_body_logic	26	17	13
Header_body_record	74	4	60
Arbiter_logic	0	24	0
arbiter	1500	624	650
Valid_tester	40	20	20
Output_data_logic	856	446	433
Crossbar_switch	1736	658	1011
Input_buffer	670	20	560
router	9856	1487	4641
Edge router (type one)	6132	1446	4045
Edge router (type two)	5408	1405	3449
network	110112	23136	81416

(Table 27, gates used summary table)

7. **Bugs**

(1) we have used judgment sentence such as
logic [4 : 0] tmp;
if (tmp == 00101) begin
…
end
and that branch is not entered correctly. We fixed by changing the if condition into
if (tmp == 5'b00101)
(2) In the module header_body_record whose function is to help judge a flit is a body one or header one, it should count up not only when a valid flit is coming, but also when that flit is a new one instead of a old flit that has been delayed in last clock cycle
(3) For buffer_storage module, when the input signal send_i is high and receive high is low, we forget to let the right register back to empty mode.
(4) In the network module, a stupid error about the typo, write data3 and indeed should write data4.
In the router, local counter is kind of special which we ignored first. (5) When other counters are full, no data transfer to that direction is allowed. But when local counter is full, the data transfer to local output port is still allowed.
(6) The counters' recorded values should be given to the output_data_logic module instead of direct judge before the direction_analyzer module. Since that determines which output direction is currently blocked, not which input buffer cannot currently send out data.
(7) In the register module, since parameter is used,

assign data_o = data & {17{valid}};
should be changed into
assign data_o = data & {DATA_WIDTH{valid}};
(8) The module crossbar switch should give related outputs to indicate which input buffers can send out a data flit in this cycle. But that is not enough, it also should give out the information about for a output direction, if in this cycle a data is sent out through that. (ignored in the design process)
(9) When writing the test vectors, a header flit cannot have the desired output the same as its input.
(10) The reset signal in the FF.sv file is designed to be synchronous, so in the testbench we need to give one more cycle of reset signal at the beginning of the tests.
(11) The back pressure mechanism is implemented by the signal local_full_o that output from the network module to the testbench with no clock delays, but the testbench responds to that signal with one cycle delay. That incurs the embarrassing situation that the input buffer is full and newly-come data will be lost silently.
(12)When the crossbar_switch computes in a certain cycle which input buffers are allowed to send data, it needs to take the counter values into consideration.

8. Module Analysis

8.1 Time Analysis

For the synthesizable version we set up the clock period to be 6ns in the dsyn.tcl file. The timing result is shown in the following Figure.

```
library setup time                                    -0.04       5.96
data required time                                                5.96
--------------------------------------------------------------------------
data required time                                                5.96
data arrival time                                                -5.96
--------------------------------------------------------------------------
slack (MET)                                                       0.00
```

Timing is a very important factor in measuring the performance of a system. The timing result shows that the slack is met when the design is synthesized using dc. Zero time slack is already optimal.

8.2 Area analysis

The area result is shown in the following Figure

```
Number of ports:                          562
Number of nets:                          1429
Number of cells:                           18
Number of combinational cells:              2
Number of sequential cells:                 0
Number of macros:                           0
Number of buf/inv:                          2
Number of references:                      17

Combinational area:       524640.153710
Noncombinational area:    218941.746073
Net Interconnect area:    43616.900066

Total cell area:          743581.899783
Total area:               787198.799849
```

The number of cells here are the next-level modules of the top module instead of cells digging into the very bottom level. The combinational area is the largest one and this is reasonable, since the complexity of this design is mainly located in the crossbar_switch module and arbiter module, both of which consist mostly combinational modules.

8.3 Power analysis

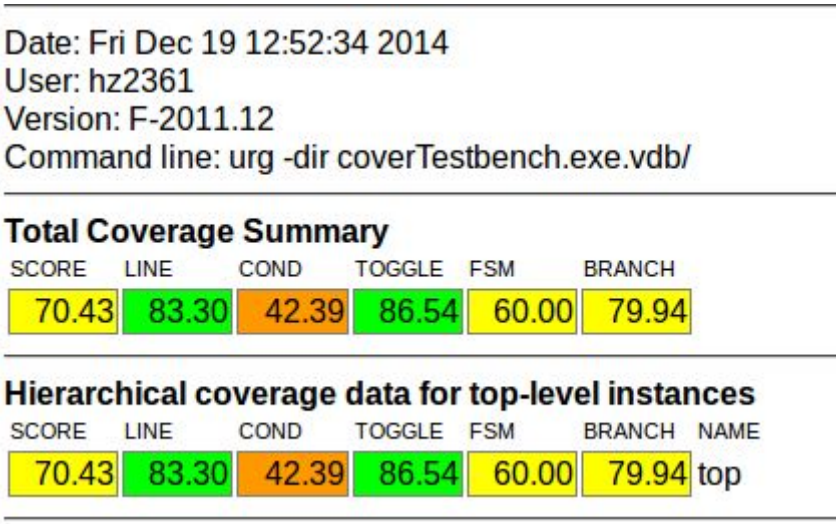
The power result is shown in the following Figure

```
Global Operating Voltage = 1.2
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   2.8638 mW   (70%)
  Net Switching Power  =   1.2344 mW   (30%)
                         ---------
Total Dynamic Power    =   4.0982 mW  (100%)

Cell Leakage Power     =   3.3340 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000           0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000           0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000           0.0000  (   0.00%)
clock_network      0.0000           0.0000            0.0000           0.0000  (   0.00%)
register         246.4010          99.0447        9.0571e+08        1.2512e+03  (  16.83%)
sequential        25.7866           5.6556        1.9408e+08          225.5254  (   3.03%)
combinational  2.5916e+03        1.1296e+03        2.2342e+09        5.9555e+03  (  80.13%)
--------------------------------------------------------------------------------------------------
Total          2.8638e+03 uW     1.2343e+03 uW     3.3340e+09 pW     7.4322e+03 uW
```

The power analysis also shows that most of the power is consumed by the combinational part of the design. The combinational part which constitutes the main logic of the design accounts for around 80% of the power whereas the sequential part and registers account for the rest.

8.4 Coverage analysis

We used the vcs tools to do the coverage analysis. We generated a unified coverage report to show total coverage summary and hierarchical coverage as well. The figure below is of the dashboard of our unified coverage report.


Date: Fri Dec 19 12:52:34 2014
User: hz2361
Version: F-2011.12
Command line: urg -dir coverTestbench.exe.vdb/

Total Coverage Summary

SCORE	LINE	COND	TOGGLE	FSM	BRANCH
70.43	83.30	42.39	86.54	60.00	79.94

Hierarchical coverage data for top-level instances

SCORE	LINE	COND	TOGGLE	FSM	BRANCH	NAME
70.43	83.30	42.39	86.54	60.00	79.94	top

From the report we see that the line coverage and branch coverage results are acceptable, which means majority of the statements we wrote had been executed. The report also gives good coverage for modules in modlist.

9. Document Revision History

Nov 4.
    design overview
Nov. 14
    Update project modules
Nov 18.
    Determined all the modules, sub-modules, interfaces

Nov 26.

Finish FSM of com_dir_logic, arbiter_logic, output_data_logic, gate-level diagram of the project

Dec 2.

micro-design architecture

validation strategy

performance analysis

area analysis

Dec 3.

Finish code for function buffer storage

Start debugging code of buffer storage function

Dec 4.

add header_body flit design modules header_body_logic, header_body_record.

Dec 5.

Buffer storage debugged and sent for final revision with corresponding test bench.

Dec 9

Buffer storage function completed. Finish code for function input buffer, arbiter and crossbar switch

Start debugging code of input buffer,arbiter and crossbar switch functions.

Dec 12.

Input buffer, arbiter and crossbar switch debugged and sent for final revision with test benches.

Dec 14

Update final code of functions and test benches

Dec 17.

Update area estimate of the project

Dec 18

Refine timing analysis and area analysis, collect the total error log.

Reference:

1. EECS4340 computer hardware design lecture slides, Unit 4: Validation. Prof. Simha Sethumadhavan

2. EECS4340 Project Description [1-6]

3. http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/dflipflop.html

4. VE270, Gang Zheng, Lec4, Lec6, UM-SJTU Joint Institute.

5. http://www.ee.usyd.edu.au/tutorials/digital_tutorial/part2/counter07.html

6. http://hyperphysics.phy-astr.gsu.edu/hbase/electronic/dflipflop.html