SystemC based NoC (Network-on-Chip) Modeling Course Project

COE838/EE8221: Systems on Chip Design

Department of Electrical, Computer and Biomedical Engineering
Toronto Metropolitan University

1. Introduction

In this NoC simulation project, students will model an NoC system using SystemC. They will investigate and model an NoC system consisting of routers (switches) and IPs (CPU or other hardware module). The main interconnection structure (topology) used will be mesh, torus or hypercube.

The students are provided with a SystemC design of a simple mesh NoC of size 1×2 including the routers or switches, IP cores and interconnection as shown in Figure 1. The SystemC code for the 1×2 mesh design is given for downloading by the students from the course directory i.e. /home/courses/coe838/labs/NoC-simulation-project/. Figure 2 shows a connection between an IP core and the router. Students will learn from the basic NoC simulation and then design a more practical NoC system model using SystemC as specified in the last section titled as "What to Design and Hand In".

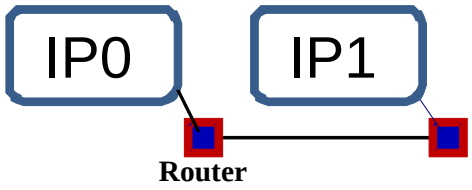


Figure 1: 1×2 NoC Mesh Architecture

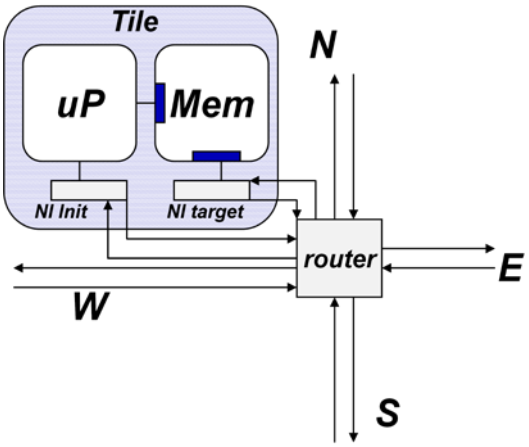


Figure 2: 2D-router and IP Core Connection

2. Modeling and Simulation of NoC

The NoC simulator is divided into a number of modules that represent various components and parts of functionality of an NoC design. These modules are the basic container object* of SystemC. To better understand the structure of a simulator, we start from a small NoC design that is depicted in Figure 3. It consists of a source module, a sink (receiver) module and a router module. These three modules are connected by communication links together.

Each module contains has two basic elements such as port and process. Ports allow communication among the modules. Processes are the main computational elements that execute concurrently. In the following sections, we describe the port and process element for each module.

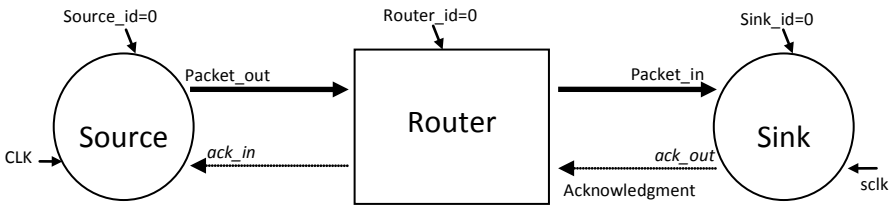


Figure 3: A Small NoC

*A container is a class, a data structure, or an abstract data type.

3. Packet Structure

The source module produces synthetic (or random) packets. The source module uses a particular message structure, which provides the design access to the packet. A message consists of packets where a packet is formed by varying number flits. A flit is the smallest element of data which travels inside the NoC at a clock cycle. In our simulator, a packet has at least two flits of header and payload. The header flits are needed to route data from the source node to the sink node. The header and payload flits are illustrated in Figure 4 and described below. The packet structure in terms of a SystemC code is listed in Figure 5.

- ***Source and sink address* bits** are used to identify the sender and receiver nodes. The size of them is defined by a parameter *FW*, which is determined depending on the number of cores in an NoC. For example, if the number of cores is sixteen meaning *FW* should be more than four. It also determines the size of the FIFO buffer such as 2 x (*FW*+5).
- ***Imaginary clock* bit** flips between 0 and 1 in each new flit and plays the role of a clock in a packet. The NoC simulator is stimulated by events and an event is invoked for a new flit. If two flits have the same contents, then an event will not be created. In order to differentiate between two flits, the clock bit is employed in every flit.
- ***Tail/Header* bit** determines the end of a packet and this bit is set high in the last flit. The payload can be more than a flit. Each payload flit carries data as well as *tail/header* and *imaginary clock* bits.

a) Header flit b) Payload flit

Figure 4: Header and payload flit

4. Source Module

The source module has three input ports *source_id*, *ach_in* and *CLK* and one output port *packet_out* as shown in Figure 6. The output port *packet_out* is connected to the router and the source module uses it to send packets. The input port *source_id* has the identification code that identifies the source module in the NoC. The input port *traffic_id* is connected to a traffic generator and it has a destination address related to the source at each clock. The input port *ach_in* is connected to the router to get an acknowledgement signal for sending a new packet. The input port

clock *CLK* is connected to the clock generator. The source module has a process, which is sensitive to +ve edge transitions for the input port *CLK*. The source process prepares packets according to packet specification, sends packets in the NoC and records the number of packets by using *pkt_snt*.

```
// packet.h file
#ifndef PACKET
#define PACKET
#include "systemc.h"
struct packet {
    sc_uint<11>     data;
    sc_uint<4>      id;       // packet source ID
    sc_uint<4>      dest;     // packet destination ID
    sc_uint<1>      pkt_clk;  // bit for changing the new packet condition
    sc_uint<1>      h_t;      // header or tail flit("1" represents tail)
    inline bool operator == (const packet& rhs) const // arregment of flit elements
    {
        return (rhs.data == data && rhs.id == id && rhs.dest == dest && rhs.pkt_clk == pkt_clk && rhs.h_t == h_t);
    }
};

Inline ostream& operator << ( ostream& os, const packet& a ) // related to SystemC
{
    os << "streaming of struct packet not implemented";
    return os;
}

Inline void         //  this part assign what should be shown in trace graph as a packet
#if defined(SC_API_VERSION_STRING)
    sc_trace( sc_trace_file* tf, const packet& a, const std::string& name )
#else
    sc_trace( sc_trace_file* tf, const packet& a, const sc_string& name )
#endif
{
    sc_trace( tf, a.id, name + ".id" );
    sc_trace( tf, a.dest, name + ".dest" );
    sc_trace( tf, a.flit_clk, name + ".flit_clk" );
    sc_trace( tf, a.h_d, name + ".h_d" );
}
#endif
```

Figure 5: SystemC Packet Structure

```
// source.h
#include "packet.h"
SC_MODULE(source) {
  sc_out<packet> packet_out;
  sc_in<sc_uint<4> > source_id;
  sc_in<bool > ach_in;                  // input acknowledgment
  sc_in_clk CLK;
  int pkt_snt;                          // variable for recording of packet sent
  void func();
  SC_CTOR(source)
    {
      SC_CTHREAD(func, CLK.pos());
            pkt_snt=0;
    }
};

// source.cpp
#include "source.h"
void source:: func()
{
    packet v_packet_out;                                    // a variable for packet
    v_packet_out.data=1000;                                 // e.g.
    v_packet_out.pkt_clk = '0';                             // e.g.
    while(true)
    {
          wait();
          if(!ach_in.read())
          {
              v_packet_out.data = v_packet_out.data +source_id.read()+ 1  ; // made a desired data
              v_packet_out.id = source_id.read();
              v_packet_out.dest= 1;                         // assign a destination
              if(v_packet_out.id == 1) goto exclode;        // prevent from reciving flits by itself
                        v_packet_out.pkt_clk= ~v_packet_out.pkt_clk ; // add an imaginary clock to each flit
              v_packet_out.h_t=false;
              pkt_snt++;
              if((pkt_snt%5)==0)v_packet_out.h_t=true;      // make tail flit (the packet size is 5)
              packet_out.write(v_packet_out);
              cout << "New Pkt Sent:   " << (int)v_packet_out.data
              << "  source:   " << (int)source_id.read()  << "  Destination:   "<< source_id.dest <<endl;
exclode:;
          }
    }
}
```

Figure 6: Source Module Code

5. Sink Module

The sink module accepts packets from the router module and keeps record of the number and time of incoming packets. It plays the role of a receiving core in the NoC. When the sink module successfully receives a packet, it sends an acknowledgment bit back to the router module. The sink module has four ports consisting of three input ports, *packet_in, sink_id* and *sclk*, and one output port, *ack_out* as depicted by the SystemC code of Figure 7. An input port, *packet_in* accepts packets from the router. The clock port *sclk* is connected to the clock generator. The input port *sink_id* has a fixed value that identifies the sink module in the network. The output port *ack_out* is used to send an acknowledgment bit to the router. The sink module contains a process *receive_data* that is invoked whenever a new packet arrives at the *packet_in* port (packet event) and a +ve edge transitions at the clock port (i.e. clock event). In the case of a packet event, the process first stops receiving of new packet from the router. Then it reads packet and keeps the records of time and number of incoming flits. In the case of a clock event, the process lets the router send new packet. The clock adjusts the speed of sink by controlling the acknowledgment to the router. Figure 7 provides the complete SystemC code of a typical sink module.

```
// sink.h
#include "packet.h"
SC_MODULE(sink) {
          sc_in<packet> packet_in;               // input port
          sc_out<bool> ack_out;                  // output port
          sc_in<sc_uint<4> > sink_id;
          sc_in<bool> sclk;
          int pkt_recv;
          void receive_data();
          // Constructor
          SC_CTOR(sink) {
                    SC_METHOD(receive_data);     // Method Process
                    dont_initialize();
                    sensitive << packet_in;
                    sensitive << sclk.pos();
                    pkt_recv = 0;
          }
};

// sink.cpp
#include "sink.h"
void sink::receive_data(){

          packet v_packet;
          if ( sclk.event() ) ack_out.write(false);
          if (packet_in.event() )
          {
                    pkt_recv++ ;
                    ack_out.write(true);
                    v_packet= packet_in.read();
                    cout << " New Pkt Received:   " << (int)v_packet.data<< "  source:  "
                    << (int)v_packet.id  << "  sink:  " << (int)sink_id.read() << endl;

          }
}
```

Figure 7: Sink Module Code

6. Router Module

A simple 2D router has five input ports and five output ports as shown in Figure 8. It is modeled to have a maximum of 10 input/output ports as well as it is used in mesh based topologies. The router module accepts packets from the source (or other router modules) and passes the packets to the sink (or other router modules). The router consists of some lower level modules such as *FIFO, crossbar, arbiter* and *demux* which are connected by signals together as illustrated in Figure 8. The router used in this simulator is some-way different than the router of Figure 8. (You need to identify the difference as given in question 1 of Section 8).

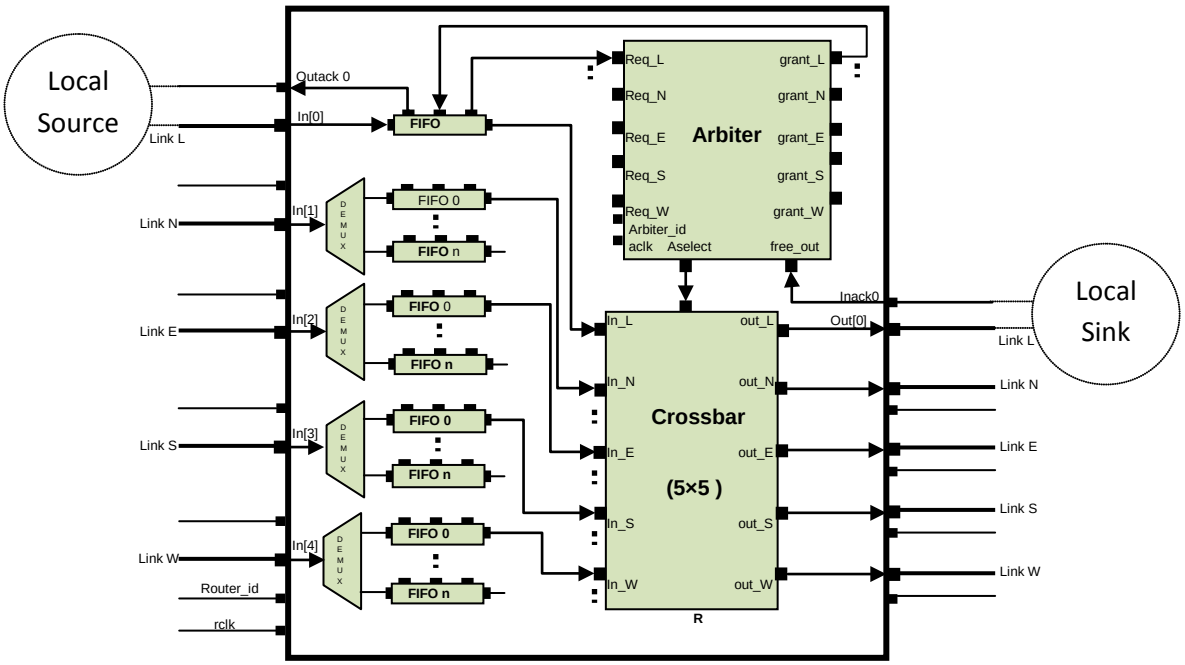


Figure 8: 5×5 Generic Router

To provide a better understanding of how a router works, we describe the journey of a header flit inside the router. Assume a local source module injects a header flit into the input port of first *FIFO* module. The *FIFO* module writes the flit into the tail of its buffers. When the flit emerges at the header of *FIFO* module, a request containing the route information is sent to the request port of arbiter module (*Req_L*) for the desired output port (assume the north output port). The arbiter module performs the required arbitration. When the request is granted, the arbitration result is sent to the configure port of crossbar module. A grant signal (*grant_L*) is also sent to the *grant* port of the *FIFO* module. Then the *FIFO* module activates its read port leading to the injection of flit to the input port of crossbar module. The flit then traverses through the crossbar module from its input port *In_L* to its north output port *out_N*. Finally, the flit will leave the router.

The router SystemC code is illustrated in Figure 9 and it has a total of 22 data and other signal ports including *rclk, router_id,* etc. The first input port, *in0* accepts packets from the source module and port *inack0* accepts acknowledgment bit from the sink module. The output ports *outack0* and *out0* send acknowledgment bit to the source module and data packets to the sink module respectively. The input port *router_id* has the constant value of router ID. The clock input port *rclk* is used to get clock signal from the clock generator. The SystemC code of Figure 9 lists all of these ports. The router process can have a process called *r_func().* This process is sensitive to the events on the four input ports: *in1, in2, in3* and *in4.* When a new packet arrives, the router function *r_func()* is invoked to keep the records of the number of incoming packets. All the router tasks like incoming packets, acknowledgments, routing and transferring packets are done by the lower level modules of the router. The router only binds these modules and executes the router process. The following sections describe these modules and their implementation in detail.

```
// router.h
#include "packet.h"
#include "buf_fifo.h"
#include "crossbar.h"
#include "arbiter.h"
SC_MODULE(router) {
    sc_in<packet>  in0; sc_in<packet>  in1; sc_in<packet>  in2; sc_in<packet>  in3; sc_in<packet>  in4;
    sc_out<packet>  out0; sc_out<packet>  out1;sc_out<packet>  out2; sc_out<packet>  out3; sc_out<packet>  out4;
    sc_in<bool>  inack0; sc_in<bool>  inack1; sc_in<bool>  inack2; sc_in<bool>  inack3; sc_in<bool>  inack4;
    sc_out<bool>  outack0; sc_out<bool>  outack1; sc_out<bool>  outack2; sc_out<bool>  outack3; sc_out<bool>  outack4;
    sc_in<sc_uint<4> > router_id; sc_in<bool> rclk;
    buf_fifo* buf0; // need codes// need codes// need codes
    buf_fifo* buf4;
    arbiter* arbiter0;
    crossbar* crossbar0;
    sc_signal<sc_uint<5>> req_s_0; sc_signal<sc_uint<5>> req_s_1; sc_signal<sc_uint<5>> req_s_2;
    sc_signal<sc_uint<5>>req_s_3; sc_signal<sc_uint<5>  >  req_s_4;
    sc_signal<sc_uint<4>  >  free_s;
    sc_signal<sc_uint<15>  >  select_s;
    sc_signal<sc_uint<1> >  gr_s_0; sc_signal<sc_uint<1> >  gr_s_1; sc_signal<sc_uint<1> >  gr_s_2;
    sc_signal<sc_uint<1> >  gr_s_3; sc_signal<sc_uint<1> >  gr_s_4;
    sc_signal<packet>  re_s_0; sc_signal<packet>  re_s_1; sc_signal<packet>  re_s_2;
    sc_signal<packet>  re_s_3; sc_signal<packet>  re_s_4;
    void func();
    int pkt_sent;
    SC_CTOR(router)
    {
          buf0 = new buf_fifo ("buf0");
                  buf0->wr(in0);
                  buf0->re(re_s_0);
                  buf0->ack(outack0);
                  buf0->req(req_s_0);
                  buf0->grant(gr_s_0);
                  buf0->bclk(rclk);
                  buf4 = new buf_fifo ("buf4");
                  buf4->wr(in4);
                  buf4->re(re_s_4);
                  buf4->ack(outack4);
                  buf4->req(req_s_4);
                  buf4->grant(gr_s_4);
                  buf4->bclk(rclk);
                  arbiter0 = new arbiter ("arbiter0");
                  arbiter0->arbiter_id(router_id);
                  arbiter0->free_out0(inack0);
                  arbiter0->free_out1(inack1);
                  arbiter0->free_out2(inack2);
                  arbiter0->free_out3(inack3);
                  arbiter0->free_out4(inack4);
                  arbiter0->req0(req_s_0);
                  arbiter0->req1(req_s_1);
                  arbiter0->req2(req_s_2);
                  arbiter0->req3(req_s_3);
                  arbiter0->req4(req_s_4);
                  arbiter0->grant0(gr_s_0);
                  arbiter0->grant1(gr_s_1);
                  arbiter0->grant2(gr_s_2);
                  arbiter0->grant3(gr_s_3);
                  arbiter0->grant4(gr_s_4);
                  arbiter0->aselect(select_s);
                  arbiter0->aclk(rclk);
                  crossbar0 = new crossbar ("crossbar0");
                  crossbar0->i0(re_s_0);
                  crossbar0->i1(re_s_1);
                  crossbar0->i2(re_s_2);
                  crossbar0->i3(re_s_3);
                  crossbar0->i4(re_s_4);
                  crossbar0->o0(out0);
                  crossbar0->o1(out1);
                  crossbar0->o2(out2);
                  crossbar0->o3(out3);
                  crossbar0->o4(out4);
                  crossbar0->config(select_s);
                  SC_THREAD(func);
                  sensitive << in0 << in1 << in2 << in3 << in4;
                  pkt_sent = 0;
    }
};
// router.cpp
#include "router.h"
void router :: func()
{
    while(true)     // functionality
    {
       wait();
              if (in0.event()){pkt_sent++;}// only record
                    if (in1.event()){pkt_sent++;}
                    if (in2.event()){pkt_sent++;}
                    if (in3.event()){pkt_sent++;}
                    if (in4.event()){pkt_sent++;}
    }
}
```

Figure 9: Router SystemC Module (only two FIFOs are instantiated)

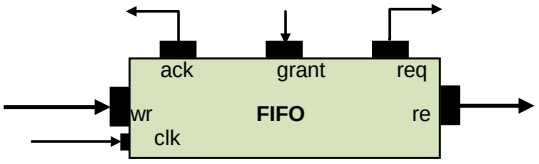
6.1. Arbiter Module

The arbiter module handles all the methods in a router like the routing/switching techniques. The *arbiter* module has eight input ports and six output ports as shown in Figure 8**.** The request and grant ports are connected to FIFO buffers. The SystemC code for the arbiter is provided in Figure 10. In the SystemC code, *aselect* port is connected to the crossbar module and when the arbitration is done, it will hv ve the free requested ootput port. The *free_out* is connected to the *in_ack* port of the router and contains the acknowledgment from the receiver modules. The *arbiter_id* is also connected to the *router_id* so that the arbiter has access to the *id* of the router. The *aclk* is connected to *rclk* leading to the router clock generator. The arbiter module has a process such as *a_func()***.** This process is sensitive to the events at the -ve of *aclk*.

When a packet is injected to a router, it is directed to the FIFO buffer. The *FIFO* module sends the routing address of packet to the arbiter as a request event. At each -ve edge of the clock, the arbiter first checks that whether an output port is free or not. If it is free, the arbiter enables the *free_out* bit related to that output port and the enabling of this bit means that the output port is ready to operate. Then the arbiter checks its request inputs. If any request is activated, it reads the destination address and checks that whether the output address is free. If it is free, then the packet will be sent through that output port. The arbiter then disables a specific bit in the variable *free_out* meaning that no data can be sent through the output port. This bit stays disable until the next clock event. If the output port is not available, the request will stay until the next clock event.

6.2. FIFO Buffer Module

When a flit is directed to the input port of FIFO module, the *FIFO* module writes the flit into the tail of its buffer. The block diagram of a typical FIFO is shown below. The FIFO issues two signals, *empty* and *full* based on the status of FIFO. The *empty* is used in *req* signal and the *full* is used as *ack* signal. When the flit emerges at the head of *FIFO*, a request containing the *empty* and destination ID is sent to the request port of a rbiter module. After the arbiter module performs the required arbitration, it sends a grant signal to the *grant* port of *FIFO* module that leads to the activation of the read port of FIFO. The flit is injected to the input port of crossbar module.



FIFO Module

A detailed description of the above mentioned operation is described here. The *FIFO* module has three input ports: *wr, grant* and *bclk* as well as three output ports: *re, req* and *ack* as illustrated in the SystemC code of Figure 11. It has a process that can be called *f_func()***.** The process is sensitive to the events on the two input ports, *wr* and *bclk*. In the write event *wr.event()*, the packet is stored in the tail of FIFO buffer. In the *bclk* event, the grant is checked and if it is set then the packet is sent to the *crossbar* module. The FIFO *struct* object provides a first-in first-out property to the buffers of *FIFO* module. The module creates this by two functions namely *packet_out()* and *packet_in()***.** The *packet_in* function stores the flit in the tail of FIFO buffer and if the buffer is full, it causes the *FIFO* module to stop receiving new packets, and if the FIFO is not empty, it generates a request to the arbiter. The *packet_out* function shifts the contents of all the registers once toward the head of *FIFO* module, and if the module is not full then it changes the condition so that FIFO starts receiving the new packet.

```
//arbiter.h
#include "systemc.h"
SC_MODULE(arbiter) {
    sc_in<sc_uint<4> > arbiter_id;
    sc_in<sc_uint<5>  >  req0;
    sc_in<sc_uint<5>  >  req1;
    sc_in<sc_uint<5>  >  req2;
    sc_in<sc_uint<5>  >  req3;
    sc_in<sc_uint<5>  >  req4;
    sc_in<bool  >  free_out0;
    sc_in<bool  >  free_out1;
    sc_in<bool  >  free_out2;
    sc_in<bool  >  free_out3;
    sc_in<bool  >  free_out4;
    sc_out<sc_uint<15> >  aselect;
    sc_out<sc_uint<1> >  grant0;
    sc_out<sc_uint<1> >  grant1;
    sc_out<sc_uint<1> >  grant2;
    sc_out<sc_uint<1> >  grant3;
    sc_out<sc_uint<1> >  grant4;
    sc_in<bool> aclk;
    void func();
    SC_CTOR(arbiter) {
      SC_THREAD(func);
      sensitive << aclk.neg();
    }
};
//arbiter.cpp
#undef SC_INCLUDE_FX
#include "packet.h"
#include "arbiter.h"
void arbiter :: func(){
          sc_uint<1> v_connected_input[5]; //set when input is connected to an output
          sc_uint<1> v_reserved_output[6]; //set when output is reserved by a input (one output more for simple coding)
          sc_uint<3> v_req[5];
          sc_uint<5> v_free;  // status of output in term of being free
          sc_uint<4> v_id;
          sc_uint<5> v_arbit;
          sc_uint<15> v_select;
          for(int i=0;i<5;i++){v_connected_input[i]=0;v_reserved_output[i]=0;v_req[i]=0;}
          v_free = 31;  // '11111'
          v_arbit = 0;
          v_select = 0;
          // functionality
          while( true )
          {
                    wait();
                    grant0.write(0);                                  // reset grant
                    grant1.write(0);                                  // reset grant
                    grant2.write(0);                                  // reset grant
                    grant3.write(0);                                  // reset grant
                    grant4.write(0);                                  // reset grant
                    if (!free_out0.read()) {v_free = v_free | 1 ; }  // set the bit 0 showing the output 0 is free
                    if (!free_out1.read()) {v_free = v_free | 2 ; }
                    if (!free_out2.read()) {v_free = v_free | 4 ; }
                    if (!free_out3.read()) {v_free = v_free | 8 ; }
                    if (!free_out4.read()) {v_free = v_free | 16 ;}
                    v_id = arbiter_id.read();
                    if (!req0.read()[4]){//if FIFO buffer is not empty
                        //if(!v_connected_input[0]) // if input is not connected i.e. it is header
                        if(v_id[0] < req0.read()[0]) v_req[0]=3; // go to east
                        else {
                              if(v_id[0] > req0.read()[0])v_req[0]=5; //go to west
                              else{
                                        if(v_id[1] < req0.read()[1])v_req[0]=4; // go to south
                                        else{
                                                  if(v_id[1] > req0.read()[1])v_req[0]=2; //go to north
                                                  else v_req[0]=1; // that is the destination
                                        }
                              }
                        }
                        switch (v_req[0]) {
                              case 1:   v_arbit=v_free &  1; break;
                              case 2: v_arbit=v_free &  2; break;
                              case 3: v_arbit=v_free &  4; break;
                              case 4: v_arbit=v_free &  8; break;
                              case 5: v_arbit=v_free &  16; break;
                              default: break ;
                        }
                        if(!v_connected_input[0]) { // if input is not connected
                            if (v_reserved_output[v_req[0]])v_arbit=0;//if requested output was reserved, goto nxt input
                        }
                        if(v_arbit!=0){
                              grant0.write(1);                                  // set grant
                              v_select.range(2,0) = v_req[0];
                              v_free  = v_free & (~v_arbit);   // inactive the related output
                              v_connected_input[0]=1; // input 0 is connected
                              v_reserved_output[v_req[0]]=1; // output is reserved
                              if(req0.read()[5]){ // if it is tail flit, reset connection and reservation
                                        v_connected_input[0]=0;v_reserved_output[v_req[0]]=0;}
                              }
                        }
                        ..................... (other input codes)
                        aselect.write(v_select);
          }
}
```

Figure 10: Arbiter Module (with one request)

```
// fifo.h
#include "packet.h"
SC_MODULE(buf_fifo) {
          sc_in     <packet>            wr;
          sc_out    <packet>            re;
          sc_in     <sc_uint<1>>        grant;
          sc_out    <sc_uint<5> >       req;
          sc_out    <bool>              ack;
          sc_in     <bool>              bclk;
          void func();
          SC_CTOR(buf_fifo) {
                    SC_THREAD(func);
                    sensitive << wr;
                    sensitive<< bclk.pos();
          }
};
struct fifo {
          public:
          packet registers[4];
          bool full;
          bool empty;
          int regnum;
          fifo(){   // constructor
                    full = false;
                    empty = true;
                    regnum = 0;
          };
          void packet_in(const packet& data_packet); // methods
          packet packet_out();
};
// buf_fifo.cpp
#include "buf_fifo.h"
void fifo::packet_in(const packet& data_packet){
          registers[regnum++] = data_packet;
          empty = false;
          if (regnum == 4) full = true;
}
packet fifo::packet_out()
{
          regnum--;
          packet temp;
          temp = registers[0];
          if (regnum == 0) empty = true;
          else {
                    registers[0] = registers[1];
                    registers[1] = registers[2];
                    registers[2] = registers[3];
          }
          full = false;
          return(temp);
    }
void buf_fifo :: func()
{
          fifo q0;
          packet b_temp;
          q0.regnum = 0;
          q0.full  = false;
          q0.empty = true;
          req.write((q0.empty, q0.registers[0].dest));
          while( true ){
                    wait();
                    if (wr.event()){    //read input packets
                              q0.packet_in(wr.read());
                              ack.write(q0.full);
                              req.write((q0.empty, q0.registers[0].dest));
                    }
                    if (bclk.event()) { //write the packets out
                              if(grant.read() == 1){
                                        b_temp = q0.packet_out();
                                        re.write(b_temp);
                                        ack.write(q0.full);
                                        req.write((q0.empty, q0.registers[0].dest));
                              }
                    }
          }
}
```

Figure 11: FIFO Module

6.3. Crossbar Switch Module

When a flit is injected to the input port of crossbar module, the crossbar module reads the address of output port associated to the packet from the input port *config*, and then sends the packet out of the router via that output port. The *crossbar* module has a process such as *c_func ()*. The process is sensitive to the events on five input ports (except *config* port). The process is invoked when one or more events happen on the input ports. In the event, it reads the configuration address from the *config* port and then sends the packet via its associated output ports. For additional information, the SystemC code of Figure 12 may be consulted.

```
// crossbar.h
#include "packet.h"
SC_MODULE(crossbar) {
          sc_in<packet>  i0;
          sc_in<packet>  i1;
          sc_in<packet>  i2;
          sc_in<packet>  i3;
          sc_in<packet>  i4;
          sc_out<packet>  o0;
          sc_out<packet>  o1;
          sc_out<packet>  o2;
          sc_out<packet>  o3;
          sc_out<packet>  o4;
          sc_in<sc_uint<15> >  config;
          void func();
          SC_CTOR(crossbar) {
                    SC_THREAD(func);
                    sensitive << i0;
                    sensitive << i1;
                    sensitive << i2;
                    sensitive << i3;
                    sensitive << i4;
          }
};
// crossbar.cpp
#include "packet.h"
#include "crossbar.h"
void crossbar :: func()
          {
          packet v_cross0;
          packet v_cross1;
          packet v_cross2;
          packet v_cross3;
          packet v_cross4;
          sc_uint<15>  v_config;
          // functionality
          while( true ){
                    wait();
                    v_config = config.read();
                    if (i0.event()){
                              v_cross0 = i0.read();
                              switch (v_config(2,0)) {
                                        case 2: o1.write(v_cross0); break;
                                        case 3: o2.write(v_cross0); break;
                                        case 4: o3.write(v_cross0); break;
                                        case 5:   o4.write(v_cross0); break;
                                        default: cout << "-------------------------wrong destination   " <<endl ;break ;
                              }
                    }
                    if (i1.event()){
                              v_cross1 = i1.read();
                              switch (v_config(5,3)) {
                                        case 1:   o0.write(v_cross1); break;
                                        case 3: o2.write(v_cross1); break;
                                        case 4: o3.write(v_cross1); break;
                                        case 5: o4.write(v_cross1); break;
                                        default: cout << "-----------------------wrong destination   " <<endl;  break ;
                              }
                    }
                    if (i2.event()){
                              v_cross2 = i2.read();
                              switch (v_config(8,6)) {
                                        case 1:   o0.write(v_cross2); break;
                                        case 2: o1.write(v_cross2); break;
                                        case 4: o3.write(v_cross2); break;
                                        case 5: o4.write(v_cross2); break;
                                        default: cout << "-----------------------wrong destination   " <<endl; break ;
                              }
                    }
                    if (i3.event()){
                              v_cross3 = i3.read();
                              switch (v_config(11,9)) {
                                        case 1:   o0.write(v_cross3); break;
                                        case 2: o1.write(v_cross3); break;
                                        case 3: o2.write(v_cross3); break;
                                        case 5: o4.write(v_cross3); break;
                                        default: cout << "-----------------------wrong destination   " <<endl; break ;
                              }
                    }
                    if (i4.event()){
                              v_cross4 = i4.read();
                              switch (v_config(14,12)) {
                                        case 1:   o0.write(v_cross4); break;
                                        case 2: o1.write(v_cross4); break;
                                        case 3: o2.write(v_cross4); break;
                                        case 4: o3.write(v_cross4); break;
                                        default: cout << "---------------------------wrong destination" <<endl ;break ;
                              }
                    }
          }
}
```

Figure 12: Crossbar Module

7. NoC Simulator Main Module

The *main* function is the top-level entity that ties all the NoC modules together and provides the clock generation and tracing capabilities. The pseudo-code of the *main* SystemC module is shown in Figure 13.

```
// main_noc.cpp
# include "files"
int sc_main(int argc, char *argv[])
{
        Define local signals;
        Define local variables;
        Declare clocks;

        Instantiate the traffic generatore;
        Connect its ports to lacal signals;

        Instantiate the sources;
        Connect its ports to lacal signals;

        Instantiate the sinks;
        Connect its ports to lacal signals;

        Instantiate the routers;
        Connect its ports to lacal signals;

        Trace instractions;

        sc_start();           // start simulation

        Close trace files;    // stop simulaton

        if(REG_TRAFFIC)       // regular traffic
        {
               Calculate the performance, power and area metrics;
        }
        if(IRREG_TRAFFIC)     // irregular traffic
        {
               Calculate the performance, power and area metrics;
        }
}
```

Figure 13: Pseudo-code of the main Function

The main SystemC function includes all the modules related to NoC simulation and modeling. First of all, instantiate each of the lower level modules and connect their ports with the signals to create an NoC model. To instantiate a low-level module, the interface of the module must be visible. The local signals are declared to connect the modules ports together. After declaration of signals,

there are three clock generation declarations: *s_clock* (source clock), *r_clock* (router clock) and *d_clock* (destination or sink clock). The number of clock generator is optional and can be equal to the number of modules in the design. However, we design the simulator to have three clock generators.

The modules in the simulator design are instantiated after the declaration statements. The source, sink and router module are instantiated as well as connected together with the locally declared signals. This completes the implementation of NoC simulator design. The SystemC program can now be built and run. A sample main function (main_noc.cpp) for a 1x2 NoC is provided with the set of files available in the course directory /home/courses/coe838/labs/NoC-simulation-project/. To make it easier to determine if the design works as intended, we create a trace file with the built-in signal tracing methods in SystemC. After simulation is executed, we can examine the results stored in the trace file with a number of visualization tools that generate waveforms and tables of results. After the simulation is completed, the instructions related to the calculation of output results are executed.

8. What to Hand In

1. Understand the given SystemC code for 1x2 NoC and answer the following questions as interim report of the project progress.
    - Explain the architecture of source module. How the source module creates data for different sources? How a packet is made at the source module (core) level?
    - Draw the architecture of router. (Figure 8 should be amended and changed)
    - Set the clock time of source modules *clk_s* equal to the router modules *clk_r* and execute NoC simulation. Then explain the simulation results on the monitor in terms of receiving data by the *sink* module of IP1.
    - Add a variable in each source module and sink module and record the sending time and receiving time of flits and then output on the monitor the average packet delay in the NoC.
    - The processes in the arbiter module manage wormhole (flow control) communication in the NoC. However, each body flit should have a destination ID (similar to header flit) that is not necessary. Change the codes of arbiter in which after receiving the header flit, it does not need any information from the body flit except the tail bit (the last bit of each flit).

2. Design and model a 4x4 mesh NoC and test its functionality by generating various types of communication patterns (uniform and neighbouring pattern) from the source to sink cores. Explain your design with full schematics, documented SystemC code of your choice in the final report. The details of the communication patterns are:
    - Uniform pattern: Each IP core sends packets to only one of the IP cores and no IP core receives packets from more than one IP. Please note that each IP core of NoC has one source and one sink module.
    - Neighbouring pattern: Each source sends packets to one of its neighbouring nodes' sink, and no sink receives packets from more than one source.

3. As a bonus convert your 4x4 mesh topology NoC into a 4x4 torus topology and design a complete NoC model along with simulation. Explain your torus design as part of your final report.