VHDL based NoC (Network on Chip) Design
Course Project

COE 718: Hardware Software Codesign of Embedded Systems
1. Introduction

In this NoC design we use a VHDL based methodology to design a NoC system. You will work in a practical FPGA design environment using Quartus-II from Altera. In this project, students will investigate and design a NoC system consisting of the router/switch, IPs (CPU or other hardware module), and interconnection structure (topology) such as mesh, tree, torus, hypercube, etc.

The students are provided the VHDL design of a basic NoC 2x2 mesh structure including the router or switch, IP cores and interconnection as shown in Figure 1. The VHDL design files are available at the course directory /home/courses/coe718/labs/NoC-design-project/. Figure 2 shows a connection between an IP core and the router. Students will learn from the basic NoC architecture and design a more practical NoC design in this project as specified in the last section titled “What to Design and Hand In”.

![Figure 1: 2x2 2D NoC Mesh Architecture](image1.png)

![Figure 2: 2D-router and IP core connection](image2.png)
The timing characteristics and the specification of hardware elements are real constraints to design the NoC architecture. The communication among the NoC elements (e.g. router, source and sink IP cores, interconnection, etc.) is based on asynchronous handshaking that each module can have different clock modules.

2. Asynchronous Communication

Figure 3 shows an asynchronous communication between two modules: sender and receiver. At event #1, at positive edge of sender clock, clk_s a data is transferred out of sender. Then at half clock later, event #2, a credit_out is transferred out of sender too. Then at event #3 or at positive edge of receiver clock module, clk_r, the credit from sender (credit_out) is detected, and then the data is stored in the receiver. The receiver issues a credit signal (credit_in) back to the sender. Then at negative edge of sender clock or event #4, when credit_in is high, it leads to deactivate the credit_out signal. Then at positive edge of receiver clock or event #5, when credit_out is detected as a low signal, the receiver deactivate credit_in signal. After event #5, the sender is free to send new data. If the receiver is not ready to receive data (e.g. it is full), it does not deactivate credit_in signal at event #5. Therefore, the sender does not start sending new data. This process is performed in all modules in our NoC simulators.

![Asynchronous Communication Diagram](image)

Figure 3: Asynchronous Communication.

2.1. VHDL Codes

The VHDL codes given below are implemented in the source IP module and represent the events #1, #2 and #4 as following.

The process OUTPUT creates event #1. In fact, when credit_in and credit_out signals are both low, data goes out and the variable delay is set. The activated delay at process CREDIT causes the credit_out to be activated (i.e. event #2). In the second part of process OUTPUT, credit_in is checked and when it is high, delay becomes low leading to the credit_in becomes low at process CREDIT (i.e. event #4).
OUTPUT:

process (clk) begin
    if (rst = '1') then  -- reset event
        v_data_out <= (others=>'0');
        delay<= '0';
    elsif (rising_edge(clk)) then
        if ((credit_in= '0') and (v_credit_out = '0')) then
            v_data_out <= (data_rom +x"001");
            delay<= '1';  --source_0
        elsif (credit_in = '1') then
            delay<= '0';
        end if;
    end if;
end process;

CREDIT:

process (clk) begin
    if (rst = '1') then
        v_credit_out <= '0';
    elsif (falling_edge(clk)) then
        if (delay = '1') then
            v_credit_out <= '1';
        elsif (delay = '0') then
            v_credit_out <= '0';
        end if;
    end if;
end process;

credit_out of source (credit_in of router) is connected to the write enable (we) signal of a synchronous read/write dual port SRAM inside the FIFO buffer (see Figure 4). Therefore, at the positive edge of router clock cycle, the data is stored in the SRAM (i.e. event#3).

At router (inside the FIFO buffer), the process CREDITOUT checks credit_in (i.e. the source credit_out). When it is activated, the router activated its credit_out (i.e. the source credit_in) i.e. event #3. This process also checks two events, deactivation of credit_in and not being full, then deactivates the credit_out i.e. event#5.
3. Packet Structure

Communication among various NoC components takes place via routing packets. The source module (core) reads packet from a ROM module and inject it to the NoC. A packet is formed by varying number of flits. A flit is the smallest element of data which travels inside the NoC at a clock cycle. In our NoC design, a packet has at least two flits known as header and tail flits. The header flit is needed to route data from the source core (node) to the sink core, and the tail is used to close the packet communication during routing. The flits in-between these two flits are named the payload flits. Each payload flit carries data as well as tail/header bit. In payload flits, tail/header bit is zero. The header and tail flits are illustrated in Figure 5 and described as follows:

- **Sink address bits** are used to identify the receiver nodes.

- **Tail/Header bit** determines the end of a packet and this bit is set high in the tail flit. The payload can be more than a flit.

![Figure 5: Header and Payload Flit](image)

The sink module accepts packets from the router module. It plays the role of a receiver core in the NoC. In our NoC design, it stores the flits in a SRAM. The flits inside the SRAM are connected to the output port to be viewed in the simulation results.

The following statement creates the information of header and tail flits for the arbiter (i.e. sends the last bit of data (header and tail bit) and destination address to the arbiter.) this information (flit) helps the arbiter to perform wormhole routing communication in the NoC.

```vhdl
flit<=data_ram (DATA_WIDTH-1) & data_ram (ADD_SINK-2 DOWNTO 0), -- add last bit of data (header and tail bit) and destination
```
4. Router Module

The VHDL design of a router for the 2D mesh NoC is provided in the accompanying VHDL code. The router has five input and five output ports. The basic router has three main components such as FIFO, Arbiter and Crossbar as shown in Figure 6.

![Figure 6: A 5x5 Wormhole Basic Router](image)

The router module accepts packets from the source core (or other router modules) and passes them to the sink (or other router modules). The router consists of some lower level modules (FIFO, crossbar, arbiter and demux) that are connected by signals together as illustrated in Figure 6. The router used in the 2x2 NoC design is a bit different than the router shown in the Figure 6. All the router tasks like incoming packets, acknowledgments, routing and transferring packets are done by the lower level modules of the router. The router entity code only binds these sub-modules together.

To provide a better understanding of how they work, 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 the first FIFO module. The FIFO module writes the flit into the tail of its buffers. When the flit emerges at the head 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 is requested). The arbiter module performs the required arbitration. When the request is granted, the arbitration result is sent to the configure port of crossbar module (cross_select_0). A grant signal (grant L) is also sent to the grant port of 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 as data_out_1 to. Finally, the flit will leave the router on (Link N). The following sections describe the sub-modules of the router.

4.1. Arbiter Module

The arbiter module handles all the requests for the output ports of a crossbar. When a packet is injected into an input port of 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 rising edge of clock cycle, the arbiter first checks whether any output port is free or not. If it is free, the arbiter enables (change to low) the free_output bit related to that output port. Enabling this bit means that the related 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 whether the output address is free or not. If it is free, then the packet will go through that output port. The arbiter then disables (change to high) a specific bit in the variable, free_output 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 unanswered until the next clock event. In a falling edge of the clock, arbiter takes care of credit_out signals.

4.2. Crossbar Switch Module

The crossbar switch consists of five 5-to-1 multiplexers that switch one of the five inputs to one output.

4.3. FIFO Buffer

A simple schematic of a typical SRAM based FIFO is shown in Figure 7. Two pointers (read and write) point to the address of SRAM (memory) where data is read or written respectively. In the figure, if read event occurs, the cell (P1), which is pointed by read pointer, appears at output, then read pointer is increased one bit (now it points to P2). If write event occurs, the input data will save at the location addressed by write pointer (i.e. P5), then write pointer will be increased one bit. The difference between write pointer and read pointer determines that the FIFO is full or empty. In fact, a full condition occurs when a write causes the difference of two pointers to be equal to the depth of FIFO, and an empty condition occurs when a read causes the difference of two pointers to be equal to zero. Both read pointer and write pointer increase circularly. In other words, after they reach to the tail of FIFO, they start from the head of FIFO.

Figure 7: Shorthand schematic of a typical SRAM-based FIFO
4.3.1. SRAM based FIFO

The SRAM based FIFO has some extra hardwire element as shown in Figure 8. When FIFO is full, no data is allowed to be saved in FIFO. The \textit{status\_cnt} block subtracts the contents of \textit{read pointer} and \textit{write pointer} to create \textit{full} or \textit{empty} conditions. \textit{Full} condition is used in CREDITOUT process to issue acknowledgment signal (\textit{credit out}) in order to signal the upstream router (or source) to stop or send flits. \textit{Empty} condition is used in the REQUEST process to issue request signal to arbiter.

![Figure 8: Architecture of SRAM based FIFO](image)

4.3.2. FIFO VHDL Codes

The following statement is \textit{status\_cnt}.

\[
\text{status\_cnt} = \text{wr\_pointer} - \text{rd\_pointer};
\]

Write pointer process.

\[
\text{WRITE\_POINTER:} \quad \text{process (credit\_in) begin}
\]
\[
\quad \text{if (rst='1') then}
\]
\[
\quad \quad \text{wr\_pointer} <= \text{others=>'0'};
\]
\[
\quad \text{elsif (falling\_edge(credit\_in)) then}
\]
\[
\quad \quad \text{wr\_pointer} <= \text{wr\_pointer}+ 1;
\]
\[
\quad \text{end if;}
\]
\[
\text{end process;}
\]

Read pointer process.

\[
\text{READ\_POINTER:} \quad \text{process (grant) begin}
\]
\[
\quad \text{if (rst='1') then}
\]
\[
\quad \quad \text{rd\_pointer} <= \text{others=>'0'};
\]
\[
\quad \text{elsif (rising\_edge(grant)) then}
\]
\[
\quad \quad \text{rd\_pointer} <= \text{rd\_pointer}+ 1;
\]
\[
\quad \text{end if;}
\]
\[
\text{end process;}
\]

\textit{Output} registers that stores data at positive edge of \textit{grant} signal.

\[
\text{OUTPUT:} \quad \text{process (grant) begin}
\]
\[
\quad \text{if (rst='1') then}
\]
\[
\quad \quad \text{s\_data\_out} <= \text{others=>'0'};
\]
\[
\quad \text{elsif (rising\_edge(grant)) then}
\]
\[
\quad \quad \text{s\_data\_out} <= \text{data\_ram};
\]
\[
\quad \text{end if;}
\]
\[
\text{end process;}
\]
The increment of empty (i.e. FIFO is not empty) goes as a request to the arbiter.

REQUEST:
process (clk) begin
  if (rst='1') then
    s_req <= '0';
  elsif (rising_edge(clk)) then
    if (status_cnt=0) then
      s_req <= '0';
    else
      s_req <= '1';
    end if;
  end if;
end process;

CREDITOUT:
process (clk) begin
  if (rst='1') then
    s_credit_out <= '0';
  elsif (rising_edge(clk)) then
    if (status_cnt=(RAM_DEPTH-1))then -- FIFO is full
      s_credit_out <= '1';
    elsif (credit_in='1')then
      s_credit_out <= '1';
    else
      s_credit_out <= '0';
    end if;
  end if;
end process;

4.3.3. Dual-Port RAM Component

Instantiation and port mapping of a synchronous read and write dual port SRAM. The dual-port ram component is provided in Verilog and students just need to instantiate this component as given below.

COMPONENT ram_dp_sr_sw
  GENERIC ( data_WIDTH : INTEGER := 12; ADDR_WIDTH : INTEGER := 3; RAM_DEPTH : INTEGER :=8 );
  PORT
  ( clk  :  IN STD_LOGIC;
    address_read :  IN STD_LOGIC_VECTOR(ADDR_WIDTH-1 DOWNTO 0);
    address_write :  IN STD_LOGIC_VECTOR(ADDR_WIDTH-1 DOWNTO 0);
    data_in  :  IN STD_LOGIC_VECTOR(data_WIDTH-1 DOWNTO 0);
    data_out  :  OUT STD_LOGIC_VECTOR(data_WIDTH-1 DOWNTO 0);
    we  :  IN STD_LOGIC;
    test  :  OUT STD_LOGIC_VECTOR(data_WIDTH-1 DOWNTO 0) );
END COMPONENT;

DP_RAM: ram_dp_sr_sw port map(clk, rd_pointer,wr_pointer, data_in, data_ram,credit_in,test); -- port mapping

5. Main VHDL Module

The noc_main module is the top-level entity that ties all the NoC modules together. The waveform file, noc_main.vwf, is designed to show the signal of ports of top-level entity, noc_main.vhd.
6. What to Hand In

1. Understanding the code for 2x2 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 core level?
   - Draw the architecture of router. (Figure 6 should be changed)
   - Explain the asynchronous handshaking communication between two routers or a router and a sink.
   - Change the clock time of source modules, \( clk_s \), as same as the router modules, \( clk_r \) and run VHDL simulation. Then explain the simulation waveforms in terms of receiving data by \( sink_1 \).
   - Add a counter in the source modules and sink modules and record data communication in the NoC.
   - The processes in the arbiter manage wormhole flow control communication in the NoC. However, each body flit should have a destination ID (like in header flit) that is not necessary. Change the codes of arbiter in which after receiving the header flit, it does not care of body flit information except the tail bit (the last bit of each flit).

2. Design a 4x4 mesh NoC design 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, VHDL or Verilog code of your choice and final report. The details of the communication patterns are given below.

   - Uniform pattern: Each source sends packets to only one of sinks and no sink receives packets from more than a source (notice: each node of NoC has one source and one sink).
   - Neighbouring pattern: Each source sends packets to one of its neighbouring sinks, and no sink receives packets from more than a source.

3. As a bonus convert your 4x4 mesh topology NoC into a 4x4 torus topology and provide complete NoC design and explanation in your final report. Use uniform and neighbouring traffic patterns employed in 2.