Cardinal NIC and Chip Multiprocessor Project Part 3

Assigned: April 2, 2018
Due: April 16, 2018 11:59PM
Prof. Tabar

Objective

The goal is for students to complete the design of a 4-node Cardinal Chip Multiprocessor by first designing the Cardinal Network Interface Component (NIC) and then combining instantiations of it with the Cardinal Ring and instantiations of the Cardinal Processor using a RTL style of Verilog. The emphasis is on correct functionality, not timing, for this part of the project. You should not be using delays in the design itself, as we will attempt to synthesize the code in later stages of the project. However, the testbench will need to use delays for simulating the design at a given clock frequency. To establish a point of consistency across all projects, simulate at a clock cycle of 4ns (250MHz).

TeamingPolicies

Everyone should continue to work in their 2-student teams for the project.

VerilogSetup

Continue to use the Verilog environment that you used for prior project parts to complete this project phase. Specifically, use the NC-Sim environment that has been recommended in the class. Codes compiling with other tools but not with the NC-Sim environment will be penalized severely in a similar manner to a non-compiling code.

Whattodo?

1. Download the Cardinal Network Interface Component Architecture Specification from the course website.
2. Write RTL Verilog code to implement all the functionality detailed in the specification. Use a top-level module name of "cardinal_nic" for the NIC. Your NIC module port names should also be consistent with the inputs/outputs shown in Figure 2 of the architecture specification. Not following the naming convention could result in incorrect results in the TA's testbench. Keep in mind that your code needs to be synthesizable to be prepared for Part 4 of the project.
3. Write a testbench to verify your NIC design. Pay close attention to the correct handshaking operation and the NIC's ability to handle blocking. Also, you should ensure that the NIC can send and receive two packets (one from the processor and the other from the network) in one clock. You may also wish to toggle the polarity output to test whether the NIC can inject packets to the correct virtual channel.
4. Simulate your NIC design using the testbench. Make sure the NIC functions exactly as described in the specification before you proceed.
5. If necessary, modify your Cardinal Processor design to correctly handle packet sending/receiving. Follow the descriptions in the specification.
6. Integrate four Cardinal NICs, four Cardinal Processors and the 4-node Cardinal Ring into a 4-core multiprocessor. The connection is given in Figure 1. Name your top-level module cardinal_cmp". Each processor has its private instruction and data memories, which are analogous to L1 caches. Use the same simulation model of data and instruction memories provided in Part2. The external interface of this top-level module is given in Figure 2. Again, follow the inputs/outputs naming convention as shown in Figure 2 to ensure successful execution on any third party testbench.

7. Finally, write a testbench for the four-core chip multiprocessor. In the testbench, each processor sends a packet to each of the other three processors. Figure 3 depicts the communication traffic pattern. When the testbench starts, the packets to be sent by each processor are pre-loaded into its private data memory. Meanwhile, the instruction memory is pre-loaded with instructions which basically read the packets from the data memory to the register file and store the packets from the register file to the NIC when the network output channel buffer is available.

   The receiver processor should store the packets to its data memory and dump the content of data memory before exiting the testbench. Then, you should inspect the received packet to verify that the processor is able to handle the arriving packets as described in the NIC architecture specification.

   Use the same rules from Part 1 of the project to guarantee shortest-path deadlock free routing:
   1) The shortest distance route is determined beforehand when the packets are loaded into the data memory. For packets traversing the maximum distance of two hops, they always traverse in the ccw direction. Packets sent from nodes 0 and 1 should always specify odd virtual channels in their vc fields, while packets from nodes 2 and 3 should always specify even virtual channels in their vc fields. With all this being said, the direction, hop count and vc fields of the packets must be set up accordingly.
   2) The NIC must inject the packet into the proper virtual channel specified by the *vc* field of the packet. You should design only one NIC type to be instantiated four times. The implication is that the NIC design must be capable of detecting the vc field in packets to be sent and wait for the proper polarity on the network side for forwarding on to the network. DO NOT construct two types of NICs, one to inject into even virtual channel only and the other to inject into odd virtual channel only.

   The packet should be constructed properly. The vc, dir and hop count fields guarantee deadlock free routing. The source field represents the id number of the sending processor. Your design should not alter the content of the reserved field in the header as well as the payload.

   To further test the design, snooping logic can be connected to the data path between the NIC and processor and between the NIC and router. The snooping logic logs the packets received by the processor and the packets injected into the network. The functionality is verified by inspecting the log files. It is not required to implement the snooping logic in the testbench for submission. However, it is recommended.

   Whattosubmit?
   Submit the complete Verilog code including the NIC design, the multiprocessor design and testbenches for the NIC and multiprocessor designs. Please include adequate comments not only in your designs but also in your testbenches to justify that your testbenches adequately verify your design. Also, use signal labels that correspond to the signal names in the architecture specification. In addition, include a short report (Word document that is 3 pages or less) that identifies both team members and tells how the work was distributed, and the rationale for the testbench design. Also include any problem in the submitted design in the report. The submission checklist is given in table 1. For the data memory dump file, submit the content of MEM[0] – MEM[31]. Make sure your assembly code stores packets only to MEM[0] – MEM[31].

Table 1 Submission Checklist

	cardinal_nic.v (top-level design file) and any supporting Verilog files for the NIC design.
	cardinal_cmp.v (top-level design file) and any supporting Verilog files for the multiprocessor design, including Cardinal_router, Cardinal_ring, and Cardinal_cpu.
	tb_cmp.v and any supporting Verilog file for the testbench to test the cardinal_cmp module.
	cmp_test.asm and the binary assembly code.
	cmp_test.imem.0.fill, cmp_test.imem.1.fill, cmp_test.imem.2.fill, cmp_test.imem.3.fill
	cmp_test.dmem.0.dump, cmp_test.dmem.1.dump, cmp_test.dmem.2.dump, cmp_test.dmem.3.dump

[Please turn over]

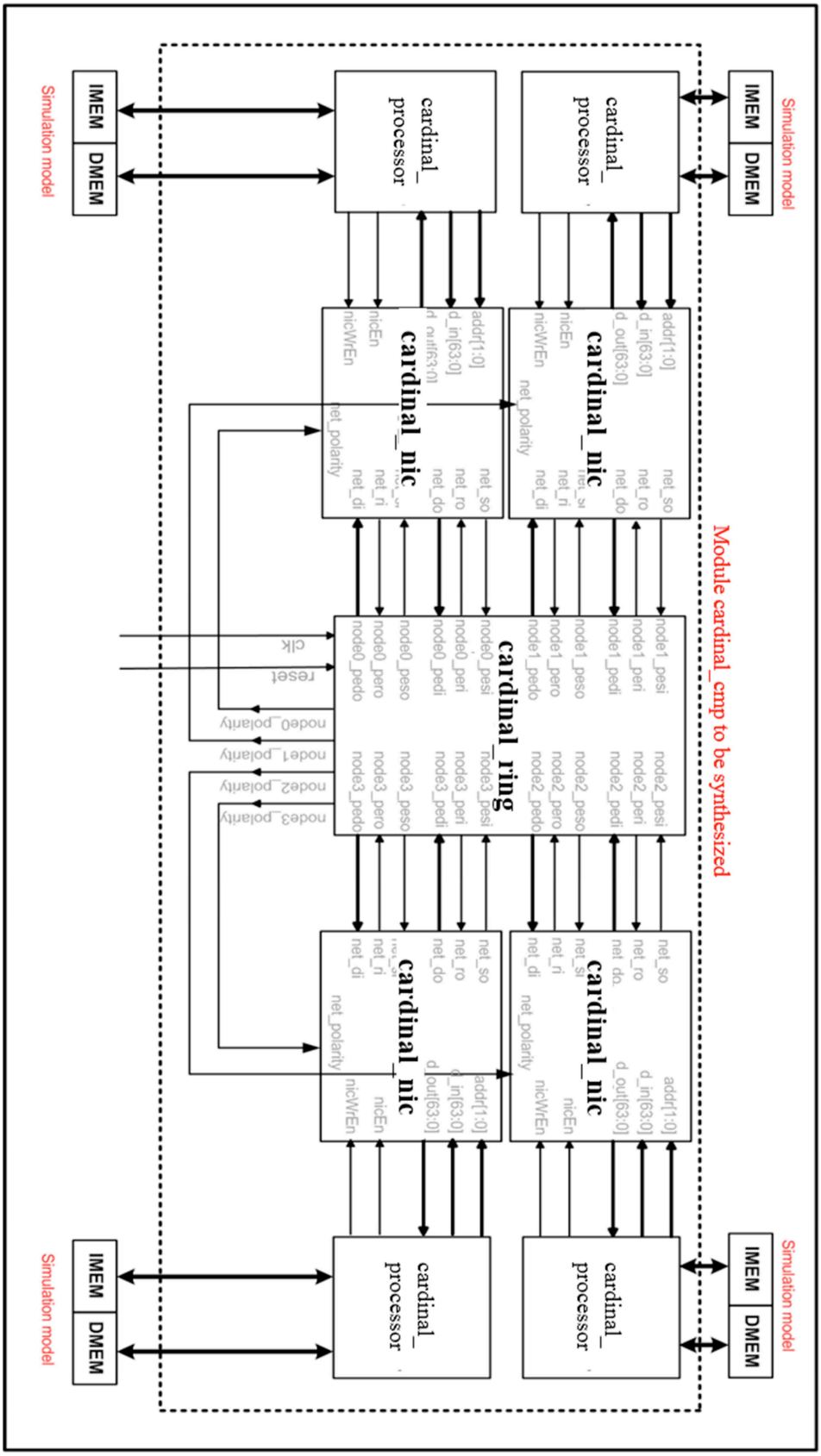
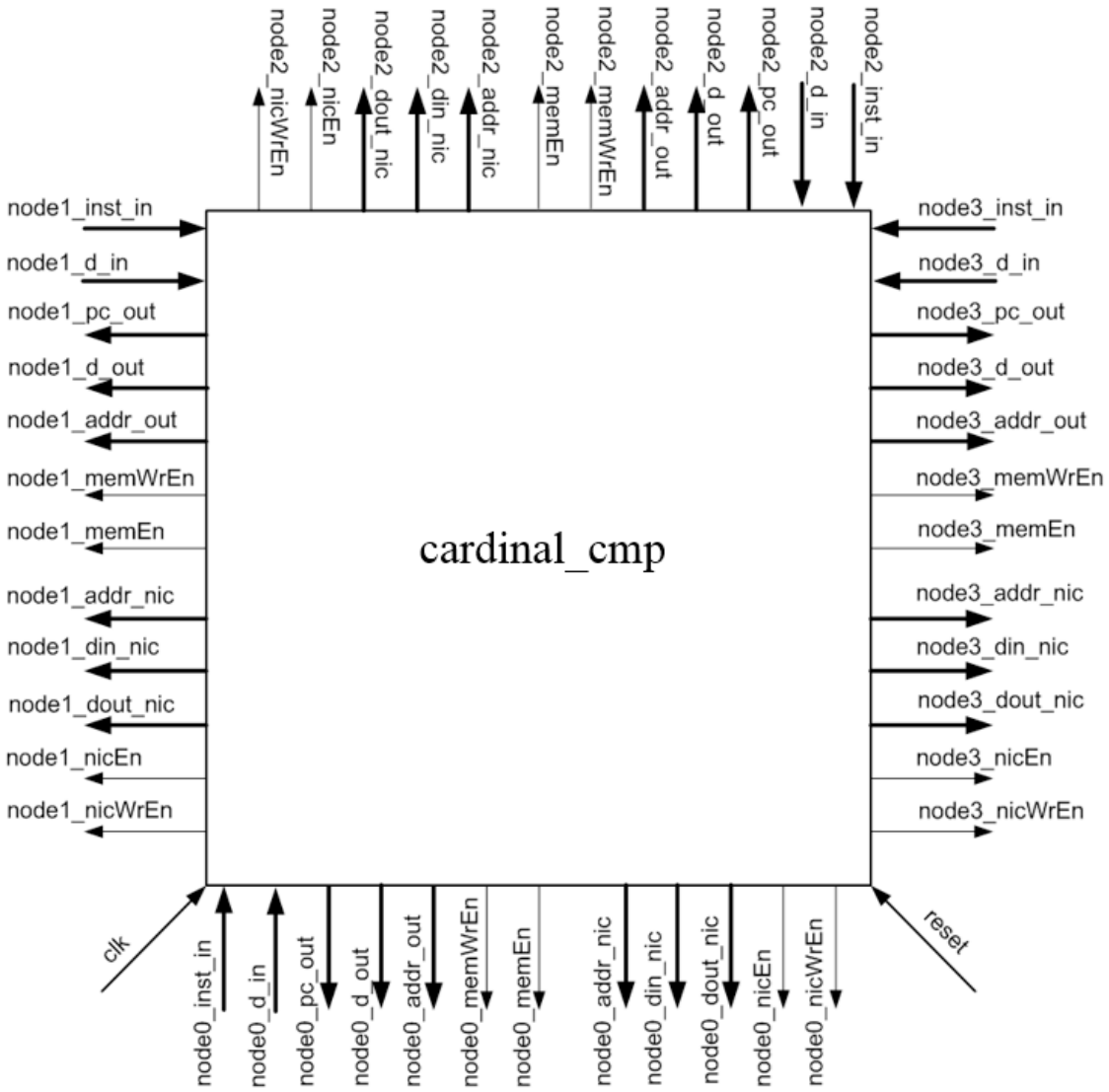


Figure 1: 4-node Chip Multiprocessor Connections.



Signal Name	Signal Type	Bit Width	Description
nodeX_inst_in	Input	32	Instruction from instruction memory to the processor
nodeX_d_in	Input	64	Input data from data memory to the processor.
nodeX_pc_out	Output	32	Program counter to the instruction memory.
nodeX_d_out	Output	64	Output data to the write port of data memory.
nodeX_memWrEn	Output	1	Data memory write enable signal.
nodeX_memEn	Output	1	Data memory enable signal.
nodeX_addr_nic	Output	2	2-bit address to the NIC
nodeX_din_nic	Output	64	Data input port of the NIC
nodeX_dout_nic	Output	64	Date output port of the NIC
nodeX_nicEn	Output	1	NIC enable signal
nodeX_nicWrEn	Output	1	NIC write enable signal
clk	Input	1	External clock in
reset	Input	1	External synchronous reset

Figure 2: cardinal_cmp External Interface and Signal Description.

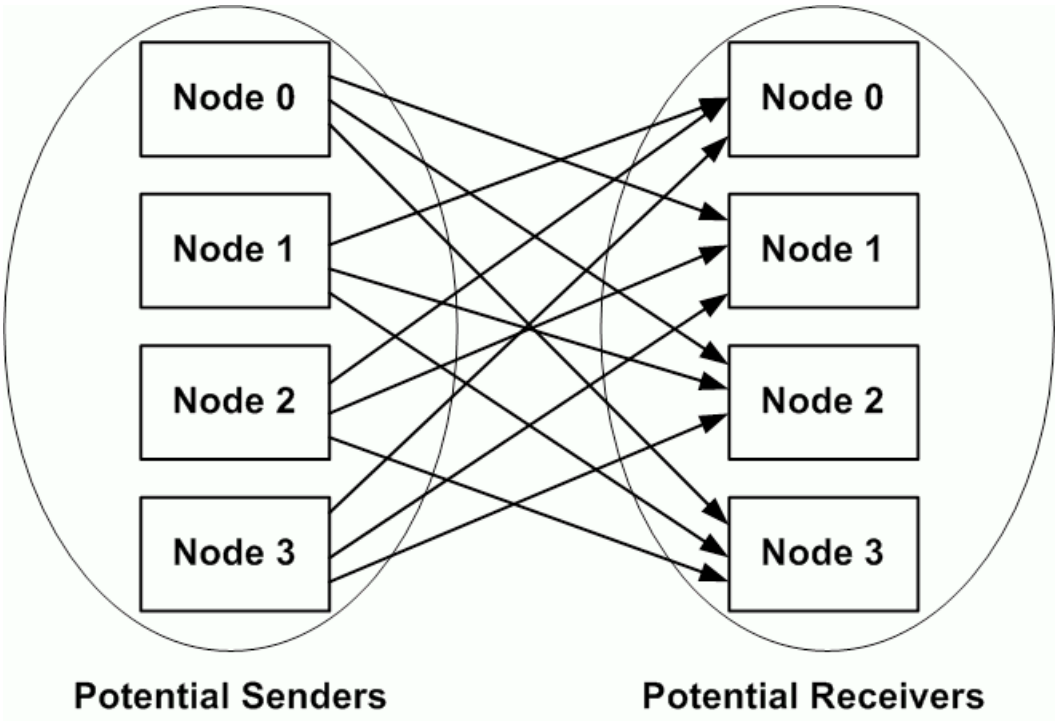


Figure 3: Illustration of Communication Pattern in the Testbench for the Multiprocessor.