Cardinal Network Interface Component Architecture Specification

This document specifies the structure and operation of the Cardinal network interface component, which is to be used as a building block of a 4-core chip multiprocessor for the course project for the Spring 2017 EE577B class.

Overview

The Network Interface Component (NIC) to be implemented to provide a path from the processor to the underlying ring network is a two-register interface, which is simple yet efficient. On the sender side, packets are sent via a single *network output channel buffer* in the NIC, to which the outgoing packets are written. On the receiver side, packets are received via a single *network input channel buffer* in the NIC, from which the incoming packets are read. The network input and output channel buffers as well as their status registers are memory address mapped. Thus, processors can access them using regular store/load instructions. As a result, the NIC provides the processors with an interface very much the same as the memory interface so that the processors need not to deal with the details in the network (e.g., handshaking signaling and polarity, among others). The illustration of the processor-network interface is illustrated in Figure 1.

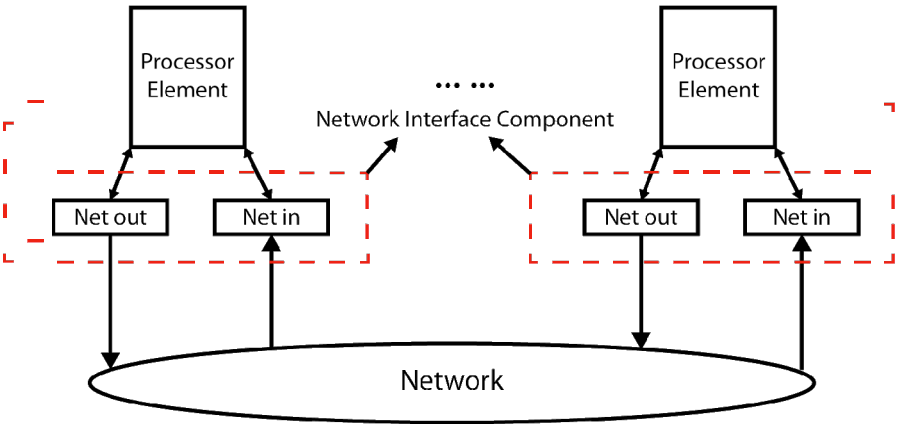

Figure 1: Illustration of the Two-register Network Interface.

ExternalInterfaceDescriptions

Refer to Figure 2 and Table 1 for the external signals for the NIC and their respective description. The NIC is a clocked (synchronous) device. The reset is synchronous and asserted high. There are two channels in the NIC: the *network output channel* and the *network input channel*. On the network output channel, packets go from the processor to the router. The 64-bit packets from the processor (same packet format as depicted in Part 1 Figure 2) to the router are injected into the *d_in* port and delivered to the routers via the *net_do* port. On the network input channel, packets go from the router to the processor. The 64-bit packets from the router are injected into the *net_di* port and delivered to the processor via the *d_out* port.

The interface between the NIC and the router is very similar to the Cardinal router design. Each channel has two control signals (s – sending, r – ready) for handshaking. The polarity signal from the connected router enables the NIC to inject the packets into the correct virtual channel according to the *vc* bit of the packet.

On the other hand, the interface between the processor and the NIC is very similar to the memory interface between the processor and data memory. The 2-bit address specifies one of the internal registers (network input channel buffer, network input channel status register, network output channel buffer and network output channel status register) to be accessed. Table 2 lists the mapping between the address and the registers. Not all the registers can be written by the processors. The network input channel status register, the network output channel status register, and the network input channel buffer are read-only to the processor. The network output channel buffer is write-only by the processor (there is no need for a processor to read from a network output channel). Unsupported (or illegal) operations to these registers will be ignored by the NIC.

The load/store operation to the NIC is synchronized with the clock signal. If *nicEn* and *nicWrEN* are both high and *addr*[0:1] specifies the network output channel buffer, the packet on the *d_in* port is written to the output channel buffer at the next rising clock edge. If *nicEn* is high and *nicWrEN* is low, the content of the register specified by *addr*[0:1] is placed onto the *d_out* port at the next rising clock edge. Because the status registers are 1-bit wide, a valid load from the status register will have the 1-bit register content being placed on the least significant bit of the NIC's data output port (i.e., *d_out*[63] in Figure 2). All the other bits of the data output port (i.e., *d_out*[0:62]) should be 0.

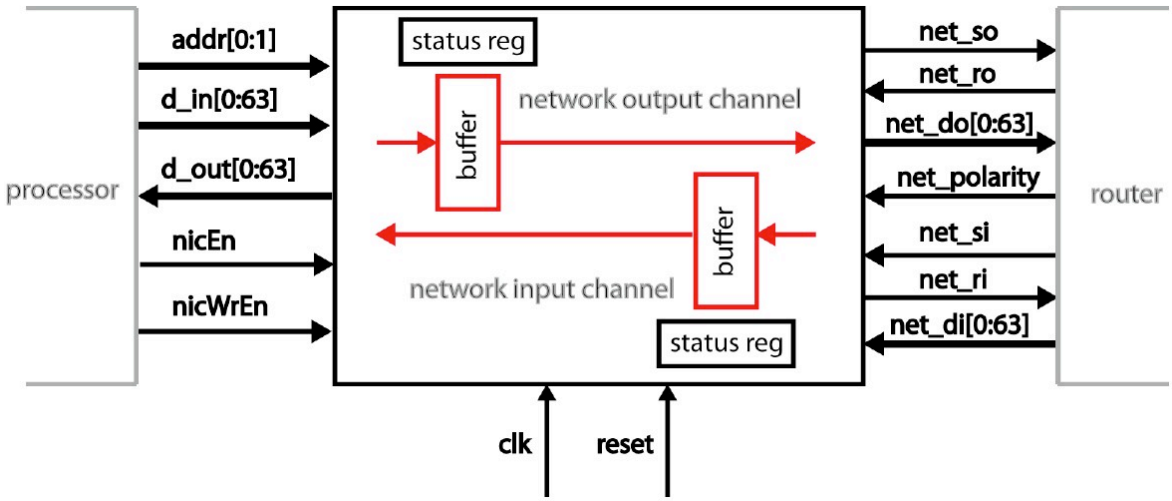

Figure 2: cardinal_nic Module External Interface.

Table 1: Signal Description for Cardinal NIC

Signal Name	Signal Type	Bit Width	Description
addr[0:1]	Input	2	Specify the memory address mapped registers in the NIC.
d_in[0:63]	Input	64	Input packet from the PE to be injected into the network.
d_out[0:63]	Output	64	Content of the register specified by addr[1:0].
nicEn	Input	1	Enable signal to the NIC. If not asserted, d_out port assumes 64'h0000_0000.
nicEnWr	Input	1	Write enable signal to the NIC. If asserted along with nicEn , the data on the d_in port is written into the network output channel
net_si	Input	1	Send handshaking signal for the network input channel. When asserted, indicates channel data is a valid packet that should be latched at next rising clk edge into the internal channel buffer.

net_ri	Output	1	Ready handshaking signal for the network input channel. Asserted when the network input channel buffer is empty.
net_di[0:63]	Input	64	Packet data for the network input channel.
net_so	Output	1	Send handshaking signal for the network output channel. Asserted when the channel buffer has packet to send and net_ro signal is asserted.
net_ro	Input	1	Ready handshaking signal for the network output channel. When asserted, indicates the router has space for a new packet.
net_do[0:63]	Output	64	Packet data for the network output channel.
net_polarity	Input	1	Polarity input from the router connected to the NIC.
clk	Input	1	Clock signal.
reset	Input	1	Reset signal. Reset is synchronous and asserted high.

Table 2: NIC Register Address.

addr[0:1]	**NIC internal registers**
2'b00	Input channel buffer
2'b01	Input channel status register
2'b10	Output channel buffer
2'b11	Output channel status register

InternalRegisters

A typical NIC design implements two First-In-First-Out queues to store packets from processor to the network and from network to processor, respectively. In the Cardinal NIC design, the two FIFO queues are simplified as two 64-bit registers. One register is for the network output channel, thereby being called network output channel buffer. Another register is for the network input channel, thereby being called network input channel buffer. Consequently, the NIC can store and forward one packet at a time for each of the two directions. Besides the channel buffers, each channel has a 1-bit status register to track whether the channel buffer is empty or not. When the channel buffer is full, the corresponding channel status register is set to high. When the channel buffer is empty, the corresponding channel status register is set to low. After a reset, both channel status registers are reset to 0.

HandshakingwiththeRouter

The handshaking between the Cardinal NIC and the Cardinal Router is similar to that between two Cardinal Routers. For the network input channel, the channel control asserts the *net_ri* signal to indicate that this NIC has available buffer space for a new packet from the router that is connected to this NIC. The *net_ri* signal can then simply be regarded as an indication of whether the corresponding buffer is occupied or not. When the router has data that it wishes to forward to the processor and if the *net_ri* signal of the corresponding NIC is asserted, the router asserts its *pe_so* signal along with placing the packet on the data channel. At the NIC side, when the *net_si* input signal is asserted, the data on the channel should be clocked into the channel buffer on the next rising clock edge. On the other hand, for the network output channel, the NIC checks the *net_ro* and *net_polarity* input signals when it has a packet in the channel buffer to be injected into the network. If *net_ro* is asserted and *net_polarity* indicates that the right router virtual channel is accepting packets, the NIC asserts the *net_so* output signal and places the packet data on the network output channel. In the case of blocking that happens when the router is not ready for a new packet, the packet stays in the NIC's output channel buffer.

Furthermore, no more packets from the processor are accepted. Figure 3 gives the handshaking signals on the network output channel. In the actual implementation, the NIC will only inject packets into the network during either even or odd polarity to avoid network deadlock.

After system reset, the *net_so* signal should be negated (reset to 0) and the *net_ri* signals should be asserted (set to 1).

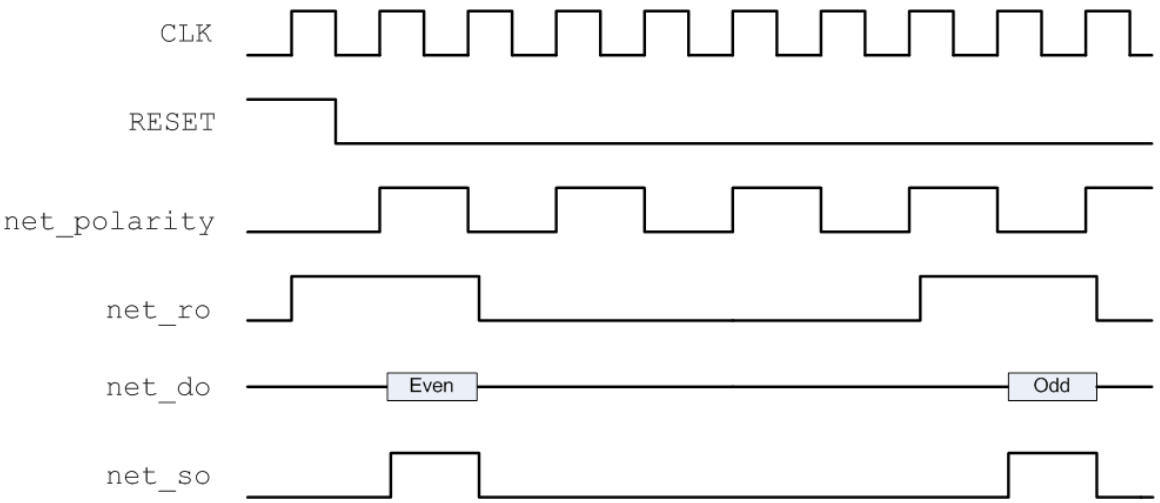

Figure 3: Sample Handshaking Timing Diagram at Network Output Channel.

InterfacingwithProcessor

The processor uses regular load and store instructions to access the internal registers of the NIC. The load and store instructions are defined in the "Cardinal Processor ISA Manual" in part 2 of this project. One memory reference instruction can either access the NIC register or access data memory. But it cannot access both at the same cycle. Processor should be able to distinguish the operation based on the immediate address field of the load/store instruction, as the channel buffers and status registers of the NIC are mapped to the processor's address space. The address mapping adheres to the following rule.

For the 32-bit memory address output from the cpu, *memAddr*[0:31] in a memory reference instruction (load/store instruction),

If *memAddr*[16] and *memAddr* [17] are both 1, the address refers to a NIC register. The least significant 2 bits (*memAddr* [30:31]) specify the NIC register according to Table 2.

If *memAddr* [16] and *memAddr* [17] are not both 1, the address refers to a data block in the data memory, where only the least significant 8 bits are used for our data memory simulation model, just as before. This is a regular memory reference. You will need to alter your Part 2 Cardinal Processor design to ensure that the memEn signal is asserted correctly for this case. You will also need to make alterations to the design to add output signals for nicEn and nicEnWr. Finally, you will also need to alter the processor design for loads to multiplex between the data out from the data memory and NIC appropriately depending on which device is being accessed. The processor design will have two input ports for data from the data memory and for data from the NIC, respectively.

When a processor has a packet in its register file to send to the network, as the first step, it should first load the value of the network output channel status register into its register file. If the network output channel status register is 0 indicating the output channel buffer is ready to accept a new packet, the processor executes a store instruction to store the packet in its register file to the network output channel buffer. If the network output channel status register is 1 indicating the channel buffer is occupied, the processor cannot store a packet to the output channel buffer and must wait in a loop until the buffer becomes available. The looping is implemented in the assembly code and not in hardware.

When a processor attempts to receive an incoming packet, as the first step, it should load the network input channel status register into its register file. If the network input channel status register is 1 indicating the input channel buffer has a new packet, the processor executes another load instruction to load the packet from the channel buffer to its own register file. If the network input channel status register is 0 indicating the input channel buffer has not received a new packet yet, the processor should not load from the channel buffer. In such a case, the processor can poll, waiting in a loop until a packet arrives. The polling and looping is implemented in the assembly code.

Certain exceptional operations must be handled correctly by the NIC. If the processor attempts to store to an occupied channel buffer, the NIC must ignore the store operation to ensure that the existing packet is not corrupted. If the processor attempts to read from an empty channel buffer, the operation is undefined so the NIC can respond with anything, e.g., you may choose to have it simply place the current contents of the network input channel buffer onto the output port to the processor. In this case, the programmer is responsible for avoiding the reading of random and erroneous data into the register file.