Report

A bus-based on-chip message passing network

Author[s]:
Liu, Lisa Ling

Publication Date:
2009-10

Permanent Link:
https://doi.org/10.3929/ethz-a-006819754

Rights / License:
In Copyright - Non-Commercial Use Permitted
A Bus-Based On-Chip Message Passing Network

Computer systems institute, ETH Zürich, Switzerland
ling.liu@inf.ethz.ch
23 March, 2009, rev. 5 October, 2009

Introduction
The design of this on-chip message passing network stems from Chuck Thacker, and modified by Niklaus Wirth in order to reduce the buffers and registers used in the network. In Chuck’s design, 32 * 32 word (32 bit wide) receiving buffer, which is equal to an entire 1k * 32 bit RAM block, is required for each core. 12 cores\(^1\) are divided into 4 groups. Each group owns a bus. Figure 1 shows the message passing network scheme that targets at Virtex-5LX50T FPGA for connecting 12 cores. The idea of this design is to avoid long distance wiring within a column.

\(^1\) In this document, "core" represents the processor that is connected to the on-chip network. The entire chip with 12 cores and the message passing network is called multi-core processor, or processor.
At the center of the columns, there are four horizontal busses, each of which is driven from a single point. This point is the outbound wiring from each column. At the center of the columns are 4 horizontal buses, which are 8-bit wide each. The output of each core is connected to the outbound arbiter, and in the end to the horizontal bus owned by its column. The message packet is driven onto the bus and captured by the inbound arbiter. Figure 2 shows the details of the logic at the center of the columns.

![Diagram of interconnect column](image)

Figure 2: Details of the interconnect column (Column 0 shown)

A message packet is divided into bytes and transmitted from the source, via horizontal bus and to the destination. The first byte of each message packet, which is 32 word long at most, stores the destination address of the packet. The second byte of the packet stores the source address of the packet. The third byte stores the packet length, in terms of number of bytes of the packet. Therefore, this network can transmit variable-length packets from a single source to a single destination.

**Handshaking protocol**

To coordinate the transmission of data between sender and receiver, a handshaking protocol is implemented. Figure 3 shows the steps involved in this handshaking protocol.

---

2 Adding broadcast would be a straightforward extension.
The sender core stores the data into its output buffer, and asserts the outflag, which signals a request to outbound arbiter. When the outbound arbiter receives a request, signal “1Dest” (the most significant bit is 1, the rest 7 bits represent the destination address) is driven onto the horizontal bus, for example, bus H0.

When an inbound arbiter detects a signal “1Dest”, with the destination core in its group and the destination core is not busy, it grants the request by asserting the corresponding grant bus (for example, HG0).

When the outbound arbiter detects the grant bus corresponding to the column is asserted, it issues a grant to the sender core, and keeps the grant until the end of the transmission process.

After acquiring the grant from the outbound arbiter, the sender core shifts bytes to the horizontal bus from its output buffer and the inbound arbiter picks up the data and transfers it to the receiver core.

Once the transmission is finished. The outbound arbiter disgrants the sender core, and the inbound arbiter disasserts the HG bus.

Once the grant signal is disasserted, the sender core disasserts the request signal in the next clock cycle.
Implementation of the interconnect network

The implementation of the interconnect network mainly involves the design of the state machines for the outbound arbiter, the inbound arbiter, the sender and the receiver. Figure 4, 5, 6, 7 show these state machines.

Figure 4: The sender machine

Figure 5: The outbound arbiter machine

Figure 6: The receiver machine
The synchronization among these state machines embodies the handshaking protocol in Figure 3. When the outflag is asserted in the sender core, a request is asserted and the first byte of the message packet is driven to the input of the outbound arbiter. If the outbound arbiter is at state "Idle" and receives a request from the core, a request is put on the horizontal bus. The most significant bit of the request is "1", the rest is the destination of the packet. If the inbound arbiter is at state idle and the receiver core is not busy, a grant signal is driven to the corresponding acknowledgement bus HG. When the outbound arbiter detects that HG is asserted, it issues a grant to the requester core, and the next byte of the sending data is shifted to the output and driven to the horizontal bus in next cycle. In inbound arbiter, the "receive" signal is asserted and the data is stored in the "RecData" the next cycle after the "grant" signal of the outbound arbiter is asserted. Once the "receive" signal is asserted, the data in "RecData" will be stored into the receiving buffer of the receiver in the next clock cycle.

**Arbitration scheme**

A round-robin arbitration scheme is implemented in the outbound and inbound arbiters. A round-robin arbiter allows every request to take a turn in order. A pointer register, which points to the next request is maintained. If that request is active, it gets the grant. If not, the pointer moves to the next request. In this way, the maximum amount of time that a request will wait is limited by the number of requesters. In the outbound arbiter, the pointer register points to 0, 1, 2 in turn. In the inbound arbiter, the pointer register points to H0, H1, H2, H3 in turn.
However, in this design, the amount of the time that a request will wait is not only limited by the number of requests to the outbound arbiter, but also by the number of requests (buses) to the inbound arbiter. Because the outbound arbiter will grant the request when the hg signal (asserted by the inbound arbiter) is asserted. As a result, the outbound arbiter allocates 4 cycles for each request to allow the inbound arbiter to go through the 4 buses. If after 4 cycles, the request is still not granted by the inbound arbiter, then either the receiver is busy or the inbound arbiter is busy. Then the outbound arbiter will put the next request on the bus and wait for 4 cycles again. Thus, ideally in our case, if the inbound arbiter and the destination processor core is not busy, the minimal message latency is 4 cycles. The maximal message latency is $2 \times 3 \times 4 + 4$ (4: request delay; 128: message transmission time; 32: time for processor core to read out message) cycles.

**Tester**

To test this on-chip network, a sequential tester is also built into the logic. Here, "sequential" means that in each core, a sequence of message packets with the destination address varying from 0 to 11 is constructed. Each packet is 125 byte long and contains the sequentially increasing data. That is to say, the packet of each core looks like the following image:

<table>
<thead>
<tr>
<th>byte127</th>
<th>...</th>
<th>byte4</th>
<th>byte3</th>
<th>byte2</th>
<th>byte1</th>
<th>byte0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7c</td>
<td>...</td>
<td>01</td>
<td>00</td>
<td>7d</td>
<td>CoreNo</td>
<td>Dest</td>
</tr>
</tbody>
</table>

The tester in the destination core constructs the expected packet and compares the received packet. If the received packet is not equal to the expected packet, an error signal is asserted and the corresponding led is turned off. The behavior of the tester mainly embodied in the two state machines in Figure 8 and Figure 9.

![Figure 8. The TestSource state machine](image)

![Figure 9. The TestDest state machine](image)

---

3 Here message latency is defined as the number of clock cycles between the request of a transmission by a sender and receipt of the first byte of the transmission by the receiver.


Results

This message-passing on-chip network has been implemented on Xilinx Virtex-5 ML505 board. It uses 17% LUTs, 10% distributed memory (implemented by LUTs) and runs at 116MHz.