DESIGN OF RECONFIGURABLE AND MODULAR NOC INTERFACE WITH ADVANCED NETWORKING FUNCTIONALITIES

Sujay Gejji & Tripti Kulkarni
Department of E&C, PESIT, Bangalore, Karnataka
E-mail: sujaypesit@gmail.com, triptikulkarni@pes.edu

Abstract - In this paper, the design of reconfigurable and modular NoC Interface (NI) with the advanced networking functionalities has been presented. The basic NoC interface decouples the communication from computation and converts the IP or processor protocols to NoC protocol and vice versa. Apart from these basic NoC functionalities, on top this main architecture the advanced networking functionalities can be implemented, hence it is reconfigurable or scalable. The implemented advanced networking functionalities are: store and forward, power management, end to end protocol interoperability, error management. In heterogeneous multiprocessor System (MPSoC), the designed NI is compliant and compatible to the processor with the little modification in the one of module of the NoC interface, hence this modular approach allows the reusability of the NI modules in MPSoC. The lot of the design time and marketing time can be saved by reusability of the modules. The flexibility is given by the scalability to add or remove the functionalities based on the higher demands so that trade off can be ignored with the intensive advantages. The proposed design is implemented in the FPGA and results are represented in the form of RTL synthesis.

Keywords - Network-on-chip (NoC), Network Interface (NI), Intellectual Property (IP), Multiprocessor System on chip (MPSoC), Network Adaptor (NA).

I. INTRODUCTION

The continuous evolution of the nanometric CMOS technology scaling and the progress in computer-aided design methodologies are fostering the development of complex MPSoC designs[1].

NETWORK-ON-CHIP (NoC) is an emerging design technology for building scalable packet-switched communication infrastructures, connecting hundreds of IP cells, in Multi-Processor System-on-Chip (MPSoC). NoCs provide a methodology for designing an interconnect architecture independently from the connected cores that can be general purpose processors, Application Specific Instruction set Processors (ASIP), Digital Signal Processors (DSP), memories or peripherals. Design flow parallelization, scalability and reusability all benefit from this approach[2].

A key element of a NoC is the Network Interface (NI) which allows IP macrocells to be connected to the on-chip communication backbone in a Plug-and-Play fashion. The NIs are the peripheral building blocks of the NoC, decoupling computation from communication. Basically, the NI is in charge of traffic packetization/depacketization to/from the NoC: it provides protocol abstraction by encoding in the packets header all data to guarantee successful end to end data delivery between IP cores (transport layer) and all Quality of Service (QoS) information needed by the router at network layer.

A NoC packet includes a header and data payloads. The header field is composed of both a Network Layer Header (NLH), whose content is determined by the NI according to the node map network configuration, and a Transport Layer Header (TLH) containing information used by the NIs for end-to-end transaction management. For example, figure 1.1 shows a NoC, based on Spidergon topology, highlighting its hardware building blocks: connected IP cores, NIs, links and Routers (R). The increment in the number of IPs, leads to the network congestion, error, power concern etc. To deal with them some NIs that in addition to the basic IP-NoC interface functionalities some features such as handling of out-of-order, error management, power management, QoS management and NI programmability. But there is trade between the area, power, circuit complexity, design complexity with increased functionalities.

To overcome these limitations this work presents the design and characterization of a NI architecture with a advanced networking features such as: store & forward transmission, error management, power management, interoperability and the implementation in the FPGA. The NI has been conceived as a scalable architecture: the advanced features can be added on top of a basic NI
core implementing data packetization and conversion of protocols, frequency and data size between the connected IP core and the NoC. The NI can be configured to reach the desired trade-off between supported services and circuit complexity.

![Figure 1.1: Spidergon Platform](image)

II. RELATED WORK AND CONTRIBUTIONS

The interface design which supports serial-link packet-based transmission model for network-on-chip application[3]. The transaction-based protocol is used to achieve backward compatibility with existing bus protocols such as AXI, OCP, and DTL[4].

Synchronisation solutions are required, when each core presents a different required bandwidth; when the cores present different clock domains; when the cores present an irregular traffic behaviour, which can change depending on the data being processed[5].

Network Interface Sharing Techniques for NoCArchitectures so that area can be saved, Network interface compatible OCP For packet based NOC, AXI compliant Network Interface (NI) for NoC[6][7][8] have been discussed.

Secure Memory Accesses on Networks-on-Chip so that congestion can be avoided, Low Latency Network Interface Architecture with Gray Code for Networks-on-Chip [9][11].

III. DESIGN OF CORE NOC INTERFACE

A. Design Of Modular (layered ) NI architecture.

The designed NI architecture is a layered architecture with three layers. The modularity of design allows to reuse of the modules. These modules can be inherited in the advanced design. The layers are again split into small modules which represent functionalities of the NI. The layers of the NI are

- Kernel
- Shell
- Interface (buffer).

These small modules of the NI are grouped under the layers based on their properties and similarities. The Shell layer is IP specific, and the Kernel, NoC specific, each one having its own peculiar functionality and interface.

The aim of the Shell/Kernel separation is to abstract IP specific properties (such as bus protocol and data size) from NoC side properties. This way the NoC becomes an IP-protocol agnostic interconnect, that is whichever protocol, bus size, clock frequency the Master or Slave IP is using, all modules in the system may communicate with each other.

The two main domains can be identified (Figure 3.1(a) referring to the top view of an Initiate NI and Figure 3.1(b) referring to the top view of a Target NIs). Figures (a)&(b) also highlight some advanced networking features modules in the layers such as, error and power management. Conversion features must be implemented in the two directions, called request path (from Master to Slave IPs, blue paths in Figures 3.1) and response path (from Slave to Master IPs, red paths in Figure 3.1 respectively. While the Kernel, and the associated NoC interface, is IP protocol independent and its design is common to all possible NIs, the Shell needs to be defined on a per-protocol basis. A specific Shell architectural design is needed for each IP protocol that must be connected to the NoC.

Figure 3.1 highlights the Shell, the Kernel and the NoC interface respectively. Moreover, the top of the figure refers to the request path, while the bottom part refers to the interconnect, and a Downstream (DS) section, receiving packets from the NoC.

![Figure 3.1: Initiator and Target NIs](image)
Fig. 3.1 Layers and Modules of NI design (a) Initiator and (b) Target

The NI Shell part deals directly with the bus protocol, implementing bus specific handshaking rules by means of dedicated Finite State Machines (FSMs). Before passing data on to the Kernel, the Shell also builds the Network and Transport Layer headers, needed by subsequent NoC components (i.e., routers and target NIs) for forwarding the packet and decoding it at destination. So it consists of NLH, TLH, FSM modules. The NI Kernel part manages buffering and other services in an IP-protocol independent way. The kernel part consists of the bysynchronous FIFO, FIFO controller module etc.

B. Implementation of Main block of NI architecture

The Kernel is interfaced to the Shell by means of a FIFO like interface. As shown in figure 5.3, encoded data coming from the Shell are stored in two FIFOs, an header FIFO (holding transport layer and network layer headers) and a payload FIFO (holding bus raw data). Each FIFO has its own read and write managers which update FIFO pointers and status, and provides frequency conversion mechanisms.

The Kernel is connected to the NoC interface stage through two additional FSMs. In the request path, an output FSM (OFSM) reads headers and payloads and converts them into packets according to the NoC protocol. In the response path, an input FSM (IFSM) collects packets and splits header and payload flits into their respective FIFOs. The NI encodes both the TLH and the NLH, while in the decoding action only the TLH is taken into account because the packet has reached its destination and routing data are not needed.

By using bi-synchronous FIFOs in the NI scheme of Figure 3.2 frequency conversion is accomplished between NoC and each connected IP. Each read (write) FIFO manager re-synchronizes in its own clock domain the pointer of the write (read) manager in the other clock domain. Hence, the empty/full status of the FIFO is known by comparison of synchronized pointers, and the header or payload FIFOs can be correctly managed.

As shown in figure 3.2, since read (RD) and write (WR) managers can access a FIFO by different basic storage units, also data size conversion between IP and NoC domain is possible. The conversion is managed by exploiting FIFO rows and columns concepts. A FIFO location, or column, is sized according to the larger data size between data in and data out; a FIFO row is sized according to the smaller data size between data in and data out. Up-size conversion is accomplished by writing by rows and reading by columns; down-size conversion is exactly the opposite. In particular conditions when no size conversion nor Shell/Kernel frequency conversion is needed, nor store &forward support is required, it is possible to remove the bysynchronous FIFOs, by setting their size to zero, thus saving area and power consumption. This feature is known as Zero-FIFO Kernel. As far as the NI crossing latency is concerned, its minimum value depends on the pipeline stages used. At least one retiming is performed due to the presence of the FIFO in the Kernel (unless the Zero-FIFO Kernel feature is enabled).

Fig. 3.2 Main blocks in the NI micro-architecture

The payload FIFO is of size thirty two bits by thirty two. Header FIFO is of size eight by thirty two, which stores the four bits of TLH and four bits NLH information. The block diagram shows modularity approach different layers and sub modules within them.

C. Finite State Machines of the main architecture

The NI design has the four finite state machines which all have their own responsibility. The FSMs are interconnected to the main architecture as shown in figure 3.2 These FSM initiates and helps for the packetization and depacketization of the messages. The upstream process OFSM control and schedule the transaction from NI to the NoC, and upstream IFSM control and direct incoming messages from IP cores. Whereas the downstream process IFSM control, schedule and disassemble the incoming packets from the NoC into data and address and stored into the respective payload FIFO and header FIFO. The downstream OFSM detects the increment in the write pointer and gets message from the FIFO and stored in to the output slave register.

Request OFSM

The Kernel is connected to the NoC interface stage through two additional FSMs. In request path an output FSM (OFSM) reads the headers and payloads or data from header FIFO and payload FIFO respectively and
converts them into packets according to NoC protocol. The Upstream process FSM control and schedule the transaction from NI to the NoC. The master process FSM is responsible of initiating network communication.

If the FSM is at wait state it waits until a request appears. It does this by checking if any of the in-buffers are not empty that is by checking the increment of the write pointer.

**Request IFSM**

The IP core is connected or interfaced with shell of request NI through input FSM. The Master process FSM is responsible for taking care of request from masters at other resources on the NoC. FMS is at wait state it waits until a master IP signal appears. When it detects that a transfer is initiated by a master on the resource, it will write the in-signals to FIFO controller (write manager, read manager). It is responsible for the reception of the signals from the master processor and receives the address and data from the IP cores. The incoming signals data and address are sent to the lower layer of networks network layer and transport layer where the address gets encoded in TLH and NLH block.

**Response Output FSM**

The Downstream process FSM control and schedule the transaction from NI to the NoC. The FSM is at wait state and it waits until a request appears. It does this by checking if any of the in-buffers of the response or target NI are not empty i.e. by checking the increment of the write pointer. The arbitration of the in-buffers is done in a round-robin manner. If one of the in-buffers contains a flits which are together sufficient to become predefined size of packet it reads the incoming message from that in-buffer (header FIFO and payload FIFO) and latches the information to the out port register.

**D. TLH Encoder and Decoder.**

The transport layer is responsible for end to end communication, it is made possible by the adding landmark or specific device address. This particular address is used to find out destination IP core. During positive edge of the cycle, if the valid is high, reset is low then the TLH address is added to the message coming from the upper layer. The included address is stored in to the TLH register. At the target it is decoded to find the destination.

**E. Implementation of Bi synchronous FIFO**

By using bi-synchronous FIFOs in the NI scheme of Figure 5.3 frequency conversion is accomplished between NoC and each connected IP. Each read (write) FIFO manager re-synchronizes in its own clock domain the pointer of the write (read) manager in the other clock domain. Hence, the empty/full status of the FIFO is known by comparison of synchronized pointers, and the header or payload FIFOs can be correctly managed. This bi-synchronous FIFO can handle arbitrary ratio clocks. Since read (RD) and write (WR) managers can access a FIFO by different basic storage units, also data size conversion between IP and NoC domain is possible. The conversion is managed by exploiting FIFO rows and columns concepts. A FIFO location, or column, is sized according to the larger data size between data in and data out; a FIFO row is sized according to the smaller data size between data in and data out. Up-size conversion is accomplished by writing by rows and reading by columns; down-size conversion is exactly the opposite. For example, consider large opcode store operations (i.e., with large amount of payload data) generated by an IP with data bus size of 32 bits and connected to a NoC with flat size of 128 bits (up-size conversion): four 32-bit data write accesses by the IP are necessary to fill a 128-bit payload FIFO location and make it available to the NoC to read it. In particular conditions when no size conversion nor Shell/Kernel frequency conversion is needed, nor Store & Forward support is required, it is possible to remove the bisynchronous FIFOs, by setting their size to zero, thus saving area and power consumption. This feature is known as Zero-FIFO Kernel. As far as the NI crossing latency is concerned, its minimum value depends on the pipeline stages used. Typically, at least one retiming is performed due to the presence of the FIFO in the Kernel (unless the Zero-FIFO Kernel feature is enabled). To increase the maximum operating clock frequency, optional pipeline stages can be added at the IP and NoC interfaces. Thus, a maximum of three retiming stages can be implemented.

**F. Store & Forward (S&F).**

Kernel FIFOs in both Request and Response paths contain flits, either received from the NoC and to be decoded towards the IP bus, or encoded from a bus transaction and to be transmitted over the interconnect. Default NI behavior is that a flit is extracted from the FIFO as soon as it is available. Hence, if the original traffic at an interface (NoC or bus) has an irregular nature, such a shape is reflected also into the other interface (bus or NoC). When Store & Forward is enabled, flits are kept into the internal kernel FIFOs until the whole packet is encoded/received and then they are transmitted(decoded all together. This way, an irregular traffic is changed to a bubble-free traffic thus improving overall system performance. For example, the interconnect can benefit from the S&F mechanism, since the link is engaged only when the entire transaction is available for transmission. After completion of a packet, the FSM controlling the FIFOs reading is in a state where only the header FIFO is checked, to extract the beginning of a new packet. The idea to handle per-packet S&F, in both directions, is to keep the packet header (i.e. the flit in the header FIFO) hidden to the reading logic by simply not updating the header FIFO write pointer.
packet is stored in the FIFOs (both header and payload), the header is unmasked and made visible by updating the header FIFO pointer, and the reading logic detects the presence of a new packet.

The management of the bus-to-NoC S&F per compound transactions is a bit more complex, since a compound transaction is composed of a number of packets, that is a number of headers. The header write pointer must be updated upon arrival of any new packet in the compound transaction, to avoid overwriting the previous one, therefore the headers become visible to the reading logic.

G. Interoperability

Interoperability across the NoC between IPs using different bus sizes and even different kinds of bus, without the need to add specific bus-to-bus bridges: the NIs are capable to handle the protocol, size and frequency conversion not only at IP-to-NoC level (and vice versa), but also at end-to-end level, obviously only for the supported IP protocols.

With no restriction on opcodes but the guarantee of addresses aligned to the Slave data bus size it is possible to enable a simplified end-to-end size conversion hardware based on a Byte Lane Matrix for reshuffling correctly the 32-bit pieces of payload in the transfer, depending on address. In other cases when the limitations cannot be applied, a specific support may be required for address realignment coupled to payload cells reshuffling through specific Byte Lane Matrix and Keep/Pass logic while the Byte Lane Matrix changes the Byte Lane position within the same transfer (vertical reshuffling), the Keep/Pass logic changes the Byte Lane position between two transfers (horizontal reshuffling), which might be needed for some wrap operations.

H. Error Management

The EMU is an optional stage that can be instantiated between the Kernel and the interface to the NoC. EMU can handle bad address errors or security violations (this second type of errors only if the Security support is enabled. When the address of the Master IP transaction is not in the range of the assigned memory map, or when the transaction is trying to access a protected memory zone without having the rights, the packet is flagged in its header as an error packet.

The EMU then filters the packet directed to NoC US interface to avoid it to enter the network, and builds a response packet re-mapping the request header on a new response header, and if needed adding dummy payload. When the Power Manager is enabled, the EMU is also in charge of properly managing incoming traffic at DS NoC interface during power down mode. All the traffic received in request during power down mode is flushed by the EMU, so that it never reaches the Slave IP.
The EMU checks for the address of the TLH and NLH blocks of the packet, if the size or number of bits within the TLH or NLH are not equal to the four bits then error flag is set. This Error flag can be used in other modules so that error packets can be discarded so that congestion in the traffic can be avoided. In future response system also can adapted so that lost messages can be re sent. So thus ensures the reliability of the system. EMU and Power manager blocks in a Target NI: Error Detector, which flushes all error traffic. In Initiator NIs, the outgoing error traffic is identified by a flag in the header, while in Target NIs all incoming packets are flushed if the connected Slave IP is in power down mode; ii Error Encoder, which assembles a new NoC packet to be channeled in the response path.

IV. SIMULATION RESULTS

A. Simulation Result of the Error Detection

Error detection module detects if addressing flaws in the transport layer TLH block and the networking layer NLH block if the TLH or NLH address is out of range that is if it more than four bits error is detected.

B. Simulation Result of Store and Forward

The simulation result shows that transmitter waits until the all data phits are of packet is arrived and then the whole packet is transmitted at once.

C. Simulation Result of Packetized data output from NI (initiator).

This shows that packetization of the input data arriving from the IP functional block. The encoded address from TLH and NLH are assembled with payload and transmitted to the router.

D. Simulation Result of Depacketized data output from the NI(target) to IP2.

The packet is disassembled into the address and payload. The address is dropped. Only data is forwarded to the IP2.

E. RTL schematic of the Top module of NoC.

The IP_I is connected to the NoCrequest by which the IP transactions are converted to the NoC protocol (packets)
The other end of NoC request is connected to the router which routes the packets to the NoC response of the destination IP_T, where the NoC response converts these packets to the IP compatible and transmitted to the IP_T.

### F. RTL synthesis report of NoC

<table>
<thead>
<tr>
<th>Device Utilization Summary (estimated values)</th>
<th>[J]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization</td>
<td>Used</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
<td>2032</td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>1757</td>
</tr>
<tr>
<td>Number of fully used LUTs</td>
<td>295</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>24</td>
</tr>
<tr>
<td>Number of BLUG/BVI numbers</td>
<td>1</td>
</tr>
</tbody>
</table>

### V. CONCLUSION

Network Interface design for on-chip communication infrastructure has been implemented on FPGA. The proposed design supports a wide set of advanced networking functionalities: store & forward transmission, error management, power management, interoperability.

This is layered and modular design so the NI can be made compatible for any IP in the NoC by little modification in the one of the NI, remaining modules are reused hence the design time can be saved.

This implemented design is reconfigurable or scalable, so on top of the advanced functionalities are implemented, when not required can be removed, so this NI configuration ensures an optimal scalability of the design, to reach the desired trade-off between supported services and circuit complexity.

The implemented NI represents a complete solution for the flow control by adapting the store and forward switching and error detection. Power manager to save the power.

NoC can also improve design productivity by supporting modularity and reuse of complex cores, thus enabling a higher level of abstraction in architectural modeling of future systems.

### VI. FUTURE SCOPE

Since only shell part modules of the NI are to be changed or modified to make it compatible to any IP socket, and remaining modules are same for all NI, so it can be implemented in semicustom chip so that area and power can be saved.

If the same IPs in the SoC have same communication socket then, NI can be shared between them on the priority basis and by using or pseudo parallelism, so that area trade off can be reduced.

### VII. REFERENCES


[9] Leandro Fiorin, Gianluca Palermo, Slobodan Lukovic, Valerio Catalano, and Cristina Silvano, “Secure Memory Accesses on Networks-on-


