A Generic Network Interface Architecture for a Networked Processor Array (NePA)

Seung Eun Lee, Jun Ho Bahn, Yoon Seok Yang, and Nader Bagherzadeh

536 Engineering Tower, Henry Samueli School of Engineering
University of California, Irvine, CA 92697-2625, USA
{seunglee,jbahn,ysyang,nader}@uci.edu

Abstract. Recently Network-on-Chip (NoC) technique has been proposed as a promising solution for on-chip interconnection network. However, different interface specification of integrated components raises a considerable difficulty for adopting NoC techniques. In this paper, we present a generic architecture for network interface (NI) and associated wrappers for a networked processor array (NoC based multiprocessor SoC) in order to allow systematic design flow for accelerating the design cycle. Case studies for memory and turbo decoder IPs show the feasibility and efficiency of our approach.

Keywords: Network-on-Chip (NoC), Interconnection Network, Network Interface, Networked Processor Array (NePA), Multiprocessor System-on-Chip (MPSoC).

1 Introduction

In order to meet the design requirements for computation intensive applications and the needs for low-power and high-performance systems, the number of computing resources in a single-chip has been enormously increased. This is mainly because current VLSI technology can support such an extensive integration of transistors and wires on a silicon. As a new SoC design paradigm, the Network-on-Chip (NoC) [1][2][3][4] has been proposed to support the integration of multiple IP cores on a single chip. In NoC, the reuse of IP cores in plug-and-play manner can be achieved by using a generic network interface (NI), reducing the design time of new systems. NI translates packet-based communication into a higher level protocol that is required by the IP cores by packetizing and de-packetizing the requests and responses of the cores. Decoupling of computation from communication is a key ingredient in NoC design. This requires well defined NI that integrates IP cores to on-chip interconnection network to hide the implementation details of an interconnection.

In this paper, we focus on the architecture of NI in order to integrate IP cores into on-chip interconnection networks efficiently. We split the design of a generic NI into master core interface and slave core interface. First, we present an NI architecture for an embedded RISC core. Then, an application specific wrapper for a slave IP core is introduced based on the NI. In order to implement a wrapper, we start by choosing application-specific parameters and writing an allocation table for architecture description. The allocation table is used for the configuration of the modular wrapper and for the software adaptation. The main contributions of this paper are a description of a generic NI architecture which allows to accelerate the design cycle and a proposal of a systematic design flow for an application specific interface.

This paper is organized as follows. Section 2 introduces an example of networked processor array (NePA) platform and related works in NI. The prototype of NI for OpenRISC interface is addressed in Sections 3. Section 4 describes a modular wrapper for a generic NI and presents case studies based on the proposed design flow. Finally, we conclude with Section 5.

2 Background

2.1 Networked Processor Array (NePA)

Since the focus of this paper is on developing a generic NI to support plug and play architecture, a simple mesh based NoC architecture is assumed. As shown in Fig. 1, NePA platform has a 2-dimensional $m \times n$ processor array with mesh topology. Each router communicates with its four neighbors and each core is connected to a router using an NI. The packet forwarding task follows a simple, adaptive routing, that uses a wormhole switching technique with a deadlock- and livelock- free algorithm for 2D-mesh topology [4]. The packet structure, shown in Fig. 2, includes two major fields. One is the destination address $(\Delta x, \Delta y)$ field to indicate the destination node in the head flit. The address of the destination node is represented by the relative distance of horizontal and vertical direction,

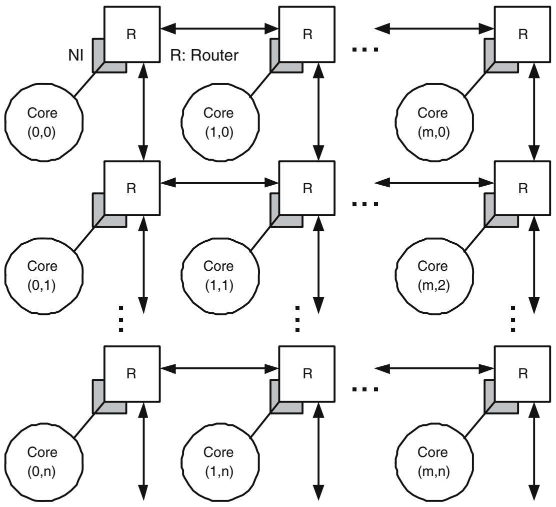

Fig. 1. A NePA architecture with mesh topology

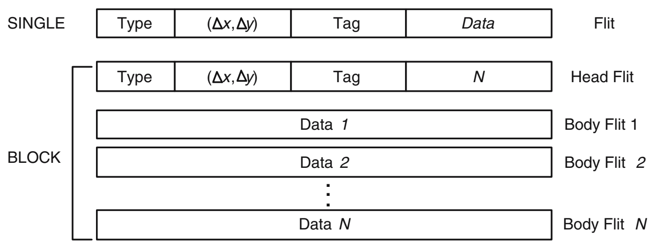

Fig. 2. Message structure

so it is updated after each transition. The second field consists of a tag and the number of data to be exchanged. The body flits deliver data to the destination IP core.

2.2 Related Works

Since most of the published works have focused on the design of novel network architecture, there has been relatively little attention to NI design. Bhojwani and Mahapatra [5] compared three schemes of paketization strategy such as software library, on-core and off-core implementation, and related costs in terms of latency and area are projected, showing trade offs in these schemes. They insisted that a hardware wrapper implementation has the lowest area overhead and latency. Bjerregaard et. al. introduced Open Core Protocol (OCP) compliant NIs for NoC [6][7][8][9] and Radulescu presented an adapter supporting DTL and AXI [10]. While standard interface has the advantage of improving reuse of IP cores, the performance is penalized because of increasing latency [7]. Baghdadi proposed a generic architecture model which is used as a template throughout the design process accelerating design cycle. Lyonnard defined parameters for automatic generation of interface for multiprocessor SoC integration [11]. However, they limited the embedded IP cores to CPUs (ARM7 and MC68000) [12]. The designs of wrapper for application specific cores still lack generic aspects and only tackle restricted IP cores. This paper investigates the actual design of NI for NePA and presents systematic design flow for arbitrary IP cores. The long-term objective is to develop a tool that automatically generates an application specific wrapper accepting as inputs the IP core interface specifications.

3 Network Interface Architecture

In the current prototype of NI, we limit the processing elements (PE) to OpenRISC cores. A tile consists of an adaptive router [4], a network interface, OpenRISC and program/data memory as shown in Fig. 3. Some parameters are needed to build a packet header for sending/receiving data over a network. These parameters are given by the PE (OpenRISC).

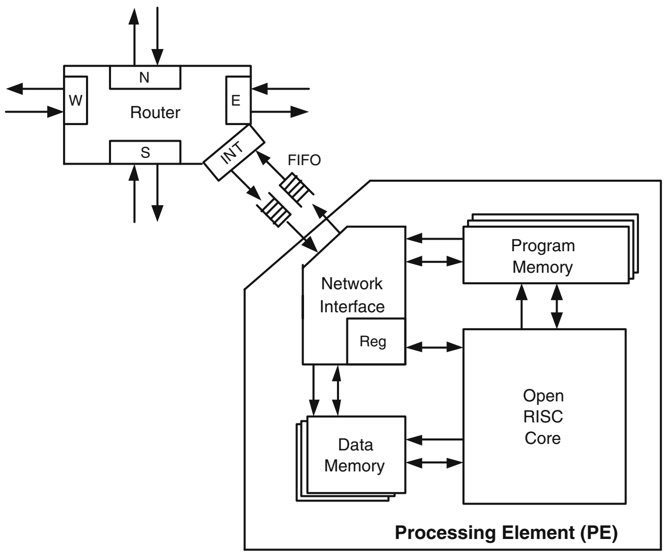

Fig. 3. A NePA tile architecture

3.1 Design of Network Interface

NI consists of a packetization unit (PU), a depacketization unit (DU) and PE interface (see Fig. 4). NI is located between a router and a PE, decoupling the communication and computation. It offers a memory-mapped view on all control registers in NI. That is, the registers in NI can be accessed using conventional bus interface. In this prototype, the parameters required to manage NI are given by OpenRISC. Table 1 shows the registers details. With this interface model, a simple implementation can be accomplished. All of the register accesses are done by bus interface and BLOCK data transfer can be handled by the DMA controller. DMA controller manages BLOCK data transfer from/to the internal memory by controlling *sReadAddrReg*, *rWriteAddrReg* and the given number of transferred data (this can be from the lower 16-bits of *rDataReg* or *sDataReg* for receiving and sending, respectively). In order to achieve high performance, all operations are completed in one cycle.

Packetization Unit. The packetization unit (PU) builds the packet header and converts the data in the memory into flits. PU consists of a header builder, a flit controller, a send DMA controller and registers. The header builder forms the packet header based on the information provided by registers such as destination address, data ID, number of body flits and service level. DMA controller generates control over the address and read signal for the internal memory by referring the start address of the memory (*sReadAddrReg*) and the number of data (*sDataReg*) for BLOCK data/program transfer. Flit controller wraps up the head flit and body flits into a packet.

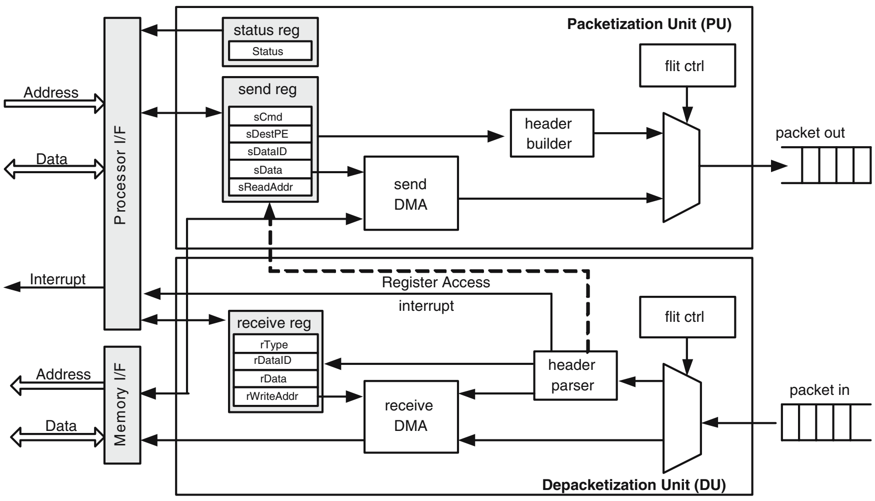

Fig. 4. NI (network interface) block diagram

Table 1. Register Definition of the Network Interface

Name	width	R/W	Offset	Description
sCmdReg	8	W	0x00	command value
sStatusReg	4	R	0x04	status register
sDestPEReg	8	W	0x08	dest_PEaddr of the corresponding packet
sDataIDReg	16	W	0x0C	data_ID /cmd_opcode
sDataReg	32	W	0x10	SINGLE: data/ operand BLOCK: number of flits
sReadAddrReg	32	W	0x14	start address of the sending data
rTypeReg	8	R	0x20	MSB 8 bit of header flit
rDataIDReg	16	R	0x24	data_id/ cmd_opcode of the received packet
rDataReg	32	R	0x28	SINGLE: data/ operand BLOCK: number of flits
rWriteAddrReg	32	W	0x2C	start address for storing BLOCK data

Depacketization Unit. The depacketization unit (DU) performs the receiving data from interconnection network. DU includes a flit controller, a header parser, a DMA controller and registers. The flit controller selects head flit from a packet and passes it to the header parser. The header parser extracts control information from the head flit such as address of source PE, number of body flits, and specific control parameters. Also, it asserts an interrupt signal to the OpenRISC core to get the local memory address for the packet. DMA controller automatically writes the body flit data into the internal memory by accessing *rWriteAddrReg* assigned by OpenRISC.

Program 1. Send SINGLE Packet from OpenRISC

```
write sDestPEReg (destination address)
write sDataIDReg (data id/op code)
write sDataReg (data)
write sCmdReg (command)
```

Program 2. Send BLOCK Packet from OpenRISC

```
write sDestPEReg (destination address)
write sDataIDReg (data id/op code)
write sDataReg (number of data)
write sReadAddrReg (start address of data)
write sCmdReg (command)
```

Program 3. Receive Packet to OpenRISC

```
read rTypeReg
if SINGLE then
   read rDataIDReg
   read rDataReg
else
   read rDataReg
   write rWriteAddrReg (start address of data)
endif
```

3.2 Programming Sequence

Both sending and receiving packets are performed by accessing the corresponding registers. Program 1 shows the programming sequence for OpenRISC core to initiate a SINGLE packet. For sending a SINGLE data/command packet, all the required parameters such as *dest_PEaddr*, *data_ID / cmd_Opcode* and corresponding 32-bit data are set to the associated registers. Finally, when the exact value of MSB 8-bit for the current transmission is set into *sCmdReg*, a complete SINGLE packet is generated by the NI, and injected into the network. For sending a BLOCK packet (Program 2), *sReadAddrReg* is used for the NI to access the internal memory. Latency for SINGLE and BLOCK transmission in NI are 4 and 5 cycles, respectively.

When a SINGLE packet arrives at the node, NI generates an interrupt. Simultaneously the necessary parameters are parsed from the received packet and stored into the associated registers. At the interrupt service routine (Program 3), each stored parameter is accessed by the internal PE. When *rDataReg* is accessed, all the procedures for the current packet is assumed to be complete. On the other hand, for receiving a BLOCK packet, the only difference is to set the corresponding write address (*rWriteAddrReg*) for internal memory access. The NI will use this as the write address for storing the following data into the internal memory. All the operations for receiving data are initiated by the corresponding interrupt generated by the NI. Latency to copy an incoming packet into internal memory is 5 cycles as shown in Program 3.

Table 2. Physical Characteristics

	NI	8-depth FIFO
Voltage	1.0V	1.0V
Frequency	719 MHz	1.8 GHz
Area	18,402 μm^2	17,428 μm^2
Dynamic Power	7 mW	10 mW
Leakage Power	184 μW	161 μW

3.3 Physical Characteristics

The NI was implemented using $Verilog^{TM}$ HDL and a logic description of our design has been obtained by the synthesis tool from the *Synposys*^{TM} using *TSMC* 90nm technology. Table 2 summarizes the physical characteristics of the NI and FIFO. The *Synopsys*^{TM} tool chain provided critical path information for logic within the NI and FIFO up to 719MHz and 1.8GHz, respectively. NI including two FIFOs has an area of approximately 0.053mm^2 (NI Area + FIFO Area $\times$ 2) using the 90nm technology. The *ARM11 MPCore*^{TM} and *PowerPC*^{TM} *E405*, that provide multi CPU designs, occupies 1.8mm^2 and 2.0mm^2 in 90nm technology, respectively [13][14]. If the NI was integrated within a NePA, the area overhead imposed by the NI would be negligible.

4 Generic Network Interface (NI)

Since NePA requires application optimization, different application specific cores may be attached to interconnection network with minimum redesign of the specific interfaces. In the remaining parts of this paper, we classify the possible IP cores for PE and define the parameters for wrapper in the context of the classification. Moreover, we provide a modular wrapper which can be configured at design time.

4.1 Classification of IP Cores for PE

A node in NePA is a specific CPU or IP core (memory, peripheral, specific hardware). We can classify IP cores into two categories: master (active) and slave (passive) IP cores (see Fig. 5). Only the master IP cores can initiate a data transfer over the network and the slave IP cores respond to requests from master IP cores.

Master IP Core. A master IP core initiates communication over interconnection network and controls NI by accessing the associated registers. It sends data packet over the network to be processed by another core and requests for packets to be sent from the other core.

A master IP core can be easily integrated into NoC using current NI architecture because it has the ability to access internal registers in the NI. A wrapper

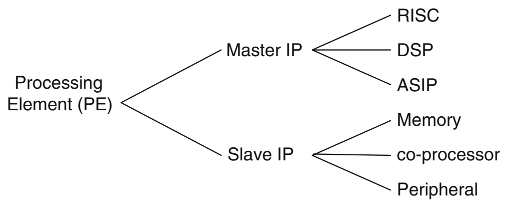

Fig. 5. Classification of IP cores

translates the protocol between IP core and NI. A master IP core is characterized with the following parameters for the purpose of a wrapper design:

– Processor type (RISC, DSP, ASIP, etc.)
– Architecture (Von Neumann, Harvard)
– BUS type (x80 system, 68 system, etc.)
– Memory size and memory map
– BUS configuration (width, data/address interleaving, endian, etc.)

For instance, a wrapper for master IP core should translate different protocols to the NI protocol according to the bus type. Architecture defines the number of interface ports and memory size determines the address width. If there is mismatch in data width, additional logic is required to adjust the data width.

Slave IP Core. A slave IP core can not operates by itself. It receives data sent over network from other cores, processes the data, and sends computed result over the network to another core. Memory, stream buffers, peripherals and co-processors (DCT, FFT, Turbo decoder, etc.) are classified as slave IP cores. Following parameters represent the characteristic of a slave IP core for a wrapper design:

– IP type (memory, co-processor, peripheral, etc.)
– Number of control signals
– Memory size and memory map
– Internal register map
– Set of control output signals (busy, error, done, re-try, interrupt, etc.)
– Data interface (serial/parallel, big/little endian, burst mode, interleaved data, etc.)

4.2 Modular Wrapper for Slave IP Cores

A slave IP core is not able to write registers in a current prototype of NI in order to indicate a destination node or to set command register. With small modification in the NI, these registers can be accessed by other cores through networks, updating the register values. This is easily realized using the predefined instruction set which access these dedicated registers (see dotted line in Fig. 4). The opcode and operand of an instruction are located at *Tag* and *Data* fields in

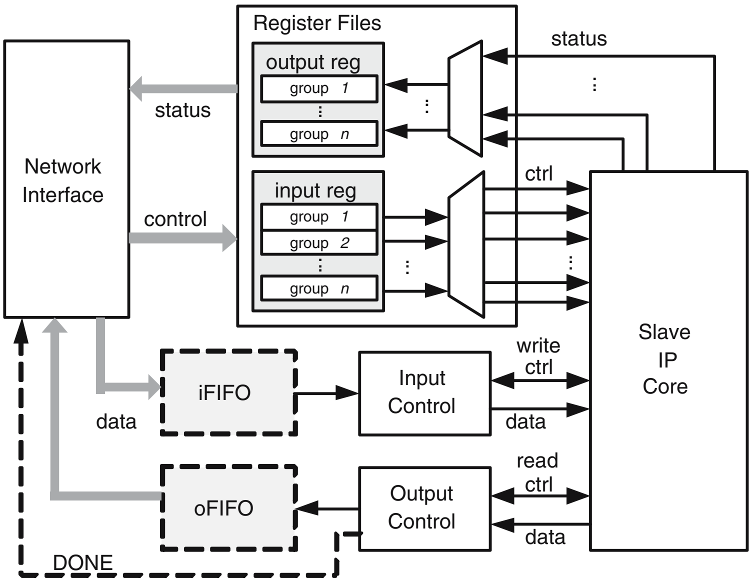

Fig. 6. Micro-architecture of a modular wrapper for a slave IP core

the SINGLE packet, respectively. *Type* field indicates that the packet contains an instruction for NI control. The instruction decoder in the header parser fetches opcode and operand from a packet and updates the internal send registers. For instance, the core (0,0) can set *sDestPEReg* in NI (2,1) to $0x01$ in order to forward the computed results of core (2,1) to core (2,2) by injecting the following packet into the network.

Type	$(\Delta x, \Delta y)$	*Tag*	*Data*
NI access		**Opcode**	**Operand**
SINGLE	0x21	write (*sDestPEReg*)	0x01

There are two signal groups, control and data signals, in a slave IP core. The input control signals initialize and manage a slave IP core. Also, a slave IP might generate status signals to indicate its internal state (busy, error, done, etc.) or to request special services (re-try, interrupt, etc.) for specific operations.

Fig. 6 shows the micro-architecture of a modular wrapper for a slave IP core interface. The input control signals are grouped by their functionality and then assigned to the application specific registers in the wrapper. These registers are accessed by NI using SINGLE packet to initialize the control signals which are allocated to dedicated signals and fed to the slave IP core completing initialization. Status signals have specific functions. For instance, the *error* signal requires special services such as generating trap to another PE or stop the operation of the slave IP core. The *done* signal initiates communication to another PE to transmit the results of the slave IP. These status signals need dedicated logic for each signal. There are a set of status signals and associated control logic to generate the controller for status signals.

Input data for a slave IP core is sent by other cores through network and NI translates the incoming packet for the slave IP core. There are differences in data width between the IP core and flit. In order to handle this mismatch, we present two operation modes for the data interface:

- **Unbuffered Mode:** data is exchanged in data stream without intermediate buffer.
- **Buffered Mode:** data is saved in the intermediate buffer temporarily.

In data interfacing, either unbuffered or buffered mode can be adopted. There are trade offs in network utilization, latency, and hardware overhead. Choosing appropriate interface mode is determined by an application designer and strongly depends on the characteristic of an application. Some cores support reading input data and writing output data concurrently, while they are processing. If the bus width of a core is less than a flit width, the interface is completed in the unbuffered mode removing the intermediate FIFOs in Fig. 6. The unbuffered mode could waste the available bandwidth of the network since it might not utilize the MSB parts of a flit. Other cores start execution after receiving all the input data in a local memory. Similarly, the result of processing is saved in memory and injected into network after completing the processing of data. Wrappers for these cores are designed in the buffered mode adding the intermediate FIFOs in Fig. 6. While the buffered mode operation increases network utilization by packing and unpacking data into a flit according to the data width, it requires additional FIFOs and packing/unpacking logic. The input and output controllers generate signals for the slave IP core completing data exchanges. The input controller reads data from the NI or FIFO and writes data to the slave IP core. On the contrary, the output controller reads data from the slave IP core and passes data to the NI or FIFO. In designing input and output controllers, designer should keep track of the specification of an IP core such as timing, data rate, etc.

For the systematic design flow, we define an allocation table for our wrapper design as shown in Table 3. Each line contains the specific parameters of an IP core for a wrapper design. *TYPE* defines the type of IP core whether it is master or slave. The input control signals are mapped to the *iControl* and the number of *iControl* depends on the number of input control signals in the IP core. The index *i* is used to access the internal register files using the specific instruction. Similarly, *oControl* reflects the status signals from the IP core. *Mode* defines the type of data transmission such as unbuffered and buffered mode. *iData* and *oData* are used to describe the interface signals to the IP core in order to complete data exchanges between NI and IP cores. The allocation table will be used for the configuration of the wrapper and for the programming model through the network.

Table 3. Allocation Table for Wrapper Design

Name	Description
TYPE	type of IP core (master/slave)
iControl i	map for ith input register
oControl i	map for ith output register
Mode	type of data transmission (Unbuffered/Buffered)
iData	signals for input data
oData	signals for output data

Table 4. Allocation Table for Memory and Turbo Decoder

Name	MEMORY	TURBO DECODER
TYPE	SLAVE	SLAVE
iControl 1	NONE	BSIZE[15:0]
2		PHYMODE[1:0], RATE[2:0], IT[4:0]
3		THRESHOLD[7:0], DYN_STOP
oControl 1	NONE	EFF_IT[4:0]
Mode	Unbuffered	Buffered
iData	DIN, ADDRESS, WE(I), CS(I)	D[15:0], DEN(I), DBLK(I), DRDY(O)
oData	DOUT, ADDRESS, OE(O), CS(I)	Q[1:0], QEN(O), QBLK(O), QRDY(I)

4.3 Case Studies

In this section, we show example design flows for a memory and a turbo decoder. We first generate the allocation table for the specific IP cores as shown in Table 4 and present the detail architecture for wrappers based on the modular wrapper.

A wrapper for a memory. Memory elements are important resources in computing systems. Memory cores are embedded in the system in order to maintain data during processing and are shared among a number of processing elements. We assume synchronous SRAM model for the memory core. The core type is slave and there are no control signals for initialization or status monitoring. By assuming the data width to be 64-bits (the same with the flit width), data interface is realized in the unbuffered mode removing the FIFOs between NI and memory. The prototype NI already has the interface to memory core, generating address and control signals. Memory core is integrated in the NePA by wiring to the prototype NI.

In order to access the memory core through the network, a master IP core should activate the node which contains the memory core. In case of writing, the base address is set to the desired value by sending SINGLE packet to the node which contains *WRITE* instruction to the *rWriteAddrReg* register in the NI. Then, BLOCK data is sent to the the memory core (Program 4). For read operation, four registers in the NI are accessed through the network setting destination address, base address of read operation, number of data, and command register. After updating the command register (*sCmdReg*), the NI automatically

Program 4. Write to the memory core through network

```
SINGLE: write (rWriteAddrReg) // set start address
BLOCK: write (Data) // send data to memory
```

Program 5. Read from the memory core through network

```
SINGLE: write (sDestPEReg) // set return PE address
SINGLE: write (sReadAddrReg) //set read address
SINGLE: write (sDataReg) // set number of read data
SINGLE: write (sCmdReg) // initiate read packet
```

reads the data from the memory and sends the data to the destination node (Program 5).

A wrapper for a Turbo decoder. Demands on high data rate in portable wireless applications make error correcting techniques important for a communication system. An error correction technique known as Turbo Coding has a better error correction capability than other known codes [15]. In this paper, turbo decoder [16] used in wireless systems, either in the base station or at terminal side, is embedded in NePA. The core is a stand-alone turbo decoder operating in a block by block process. The core type is slave and there are six signals which are used for initialization and mode selection. We map the input control signals to three groups which are accessed by a packet. The status signal is mapped to a output control signal group. Since we adopt the buffered operation mode, the FIFOs are inserted in the modular wrapper.

The input controller unpacks 64-bits incoming flits into 16-bits input data and generates control signals (*DEN* and *DBLK*). It also observes the signal *DRDY* in order to monitor the state of the core. The output controller packs 2-bits output into 64-bits flit and forwards the flit to the output FIFO. The

Program 6. Initialize the turbo decoder through network

```
SINGLE: write (iControl 1) // set iControl1 value
SINGLE: write (iControl 2) // set iControl2 value
SINGLE: write (iControl 3) // set iControl3 value
SINGLE: write (sDestPEReg) // set return PE address
SINGLE: write (sReadAddrReg) // set address to oFIFO
SINGLE: write (sDataReg) // set number of data
```

Program 7. Write to the turbo decoder through network

```
SINGLE: write (rWriteAddrReg) // set address to iFIFO
BLOCK: write (Data) // write data to iFIFO
```

Program 8. Read from the turbo decoder through network

```
SINGLE: write (sDestPEReg) // set return PE address
SINGLE: read (oControl 1) // read oControl1 value
SINGLE: write (sCmdReg) // initiate read packet
```

data communication is completed by NI accessing the FIFOs. In addition, the output controller generates *DONE* signal to notify that decoding of one block is completed. The *DONE* signal updates the *sCmdReg* and the NI starts to send a packet to the destination node automatically reading the output FIFO.

Before starting turbo decoding, the decoder is initialized by sending packet which access the input control signals (Program 6). We also set up the destination node (*sDestPEReg*) that receives the results of turbo decoding. The read address (*sReadAddrReg*) is set to the output FIFO and the number of data (*sDataReg*) is fixed to the block size.

In order to feed data to the turbo decoder, the write address (*rWriteAddr-Reg*) is set to the input FIFO and BLOCK data is sent to the turbo decoder (Program 7). Internal state of the decoder is accessed using the output control register (*oControl 1*) as shown in Program 8.

5 Conclusions

In this paper, we proposed the network interface architecture and modular wrapper for NoC. The NI decouples communication and computing, hiding the implementation details of an interconnection network. For a generic NI, we have classified the possible IP cores for PE and introduced an allocation table for a wrapper design. The allocation table is used for the configuration of the modular wrapper and for the software adaptation. The case studies in memory and turbo decoder cores demonstrated feasibility and efficiency of the proposed design flow. In addition to being useful for designing NI, the proposed design flow can be used to generate wrapper and NI automatically.

References

1. Dally, W.J., Towles, B.: Route packets, not wires: On-chip interconnection networks. In: Proc. of the DAC 2001, pp. 684–689 (2001)
2. Tabrizi, N., et al.: Mars: A macro-pipelined reconfigurable system. In: Proc. CF 2004, pp. 343–349 (2004)
3. Lee, S.E., Bagherzadeh, N.: Increasing the throughput of an adaptive router in network-on-chip (noc). In: Proc. of the CODES+ISSS 2006, pp. 82–87 (2006)
4. Lee, S.E., Bahn, J.H., Bagherzadeh, N.: Design of a feasible on-chip interconnection network for a chip multiprocessor (cmp). In: SBAC-PAD 2007: Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing, pp. 211–218 (2007)
5. Bhojwani, P., Mahapatra, R.: Interfacing cores with on-chip packet-switched networks. In: Proc. of the VLSID 2003, pp. 382–387 (2003)
6. Bjerregaard, T., et al.: An ocp compliant network adapter for gals-based soc design using the mango network-on-chip. In: Proc. of the 2005 Int'l Symposium on System-on-Chip, pp. 171–174 (2005)
7. Ost, L., et al.: Maia: A framework for networks on chip generation and verification. In: Proc. of the ASP-DAC 2005, pp. 49–52 (2005)
8. Stergiou, S., et al.: xpipes lite: A synthesis oriented design library for networks on chips. In: Proc. of the DATE 2005, pp. 1188–1193 (2005)

9. Bhojwani, P., Mahapatra, R.N.: Core network interface architecture and latency constrained on-chip communication. In: Proc. of the ISQED 2006, pp. 358–363 (2006)
10. Radulescu, A., et al.: An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. IEEE Trans. Computer Aided Design of Integrated Circuits and systems 24(1), 4–17 (2005)
11. Lyonnard, D., et al.: Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In: Proc. of the DAC 2001, pp. 518–523 (2001)
12. Baghdadi, A., et al.: An efficient architecture model for systematic design ofapplication-specific multiprocessor soc. In: Proc. of the DATE 2001, pp. 55–62 (2001)
13. ARM: Arm11 mpcore, `http://www.arm.com`
14. IBM: Ibm powerpc 405 embedded core, `http://www.ibm.com`
15. Vucetic, B., Yuan, J.: Turbo codes: Principles and applications. Kluwer Academic Publishers, Dordrecht (2000)
16. TurboConcept: High speed wimax convolutional turbo decoder, `http://www.turboconcept.com`