IPC에 근거한 래퍼 설계 방법론

윤 창 열†·장 경 선††

요 약

IP를 재사용하기 위해서는 테스트 벤치의 작성과 인터페이스 프로토콜 변환 회로 설계 등과 같은 인터페이스 프로토콜에 관련된 작업이 필요하다. 이러한 작업의 결과물은 버스기능모델에서 버스 프로토콜 컴포넌트에 대응하는 인터페이스 프로토콜 컴포넌트를 포함한다. 본 논문에서는 재사용 될 수 있는 인터페이스 프로토콜 컴포넌트를 사용하여 인터페이스 회로를 설계하는 방법론을 제안한다. 인터페이스 프로토콜 컴포넌트는 주어진 인터페이스 포트를 통해서 오는 트랜잭션을 인식하거나 트랜잭션을 사이클 수준으로 실행시켜 준다. 트랜잭션 중심으로 인터페이스 프로토콜을 기술하는 언어를 소개하고, 이 언어로부터 인터페이스 프로토콜 컴포넌트를 합성가능한 VHDL 형태로 생성하는 방법을 설명한다. 실험 결과를 통해, 인터페이스 프로토콜 컴포넌트를 이용한 인터페이스 회로 설계 방식이 그렇지 않은 설계 방식에 비하여 많은 추가 면적을 요구하지 않음을 보인다. 제안된 인터페이스 설계 방식에서는 설계자가 IP의 인터페이스 프로토콜을 상세히 이해하지 않아도, 인터페이스 프로토콜 컴포넌트를 재사용할 수 있으므로, 인터페이스 설계 시간을 줄이는데 공헌 할 수 있을 것이다.

A Wrapper Design Methodology Based On IPCs

Chang-Ryul Yun† · Kyoung-Son Jhang††

ABSTRACT

Reusing IPs requires interface protocol related tasks such as writing test benches and designing interface protocol conversion circuits, e.g. wrappers for IPs. The results of those tasks usually include IPC(interface protocol component)s for the corresponding IPs, similar to bus protocol components of the bus functional models. This paper proposes a methodology for the interface circuit design using synthesizable IPC that can be re-used. IPC recognizes or executes transactions over the given interface ports. So we present a transaction-oriented interface protocol description language, and a method to convert the description into an IPC in synthesizable VHDL code. With experiments, we show that the interface design using IPC does not cause significant area overhead compared with the interface design without IPC. The proposed IPC-based approach can be employed to reduce the interface design time since the designers can reuse IPCs without understanding the detailed interface protocols.

키워드 : 인터페이스 프로토콜 컴포넌트(interface protocol component), 재사용(reuse), 트랜잭션(transaction)

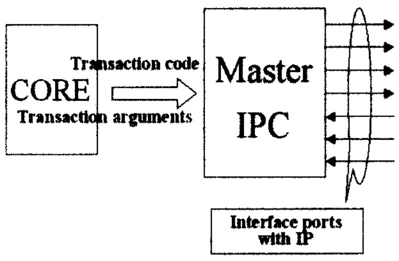
1. Introduction

Reusing IPs (Intellectual Property) requires designers to perform interface protocol related tasks such as writing test benches and designing interface protocol conversion circuits, e.g. wrappers for IPs. Designers need to understand the detailed signaling of the corresponding interface protocols to do such tasks. BFM (bus functional model) is helpful to reduce times to verify IP if it is available. Results of most interface related tasks include the IPCs (interface protocol component) for the corresponding IPs, similar to bus protocol components [1] of BFMs. Interface protocols of most IPs can be abstracted in transactions. The function of the master IPC is to execute transactions over the given interface ports according to the demands of the core as shown in (Figure 1). On the contrary, the slave IPC recognizes the transactions being executed over the interface ports, which is then notified to the core.

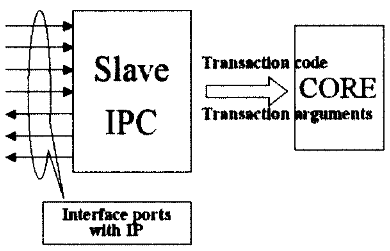
There have been proposed several researches about interface synthesis. The approach taken by Narayan and Gajski [2], interface protocol is described with five types of atomic operations : (1) waiting for an event on an input control line, (2) assigning a value to an output control line, (3) reading a value from input data line, (4) assigning a value to an output data line, and (5) waiting for a fixed time interval.

※ Korea Research Foundation under Grant (KRF-2000-003-E00222) supported this work. Logic synthesis tools and VHDL simulation tools that we used for experiments were provided by IDEC (Integrated Design Education Center) in Korea.
† 준 회 원 : 충남대학교 대학원
†† 정 회 원 : 충남대학교 정보통신공학부 교수
논문접수 : 2002년 8월 20일, 심사완료 : 2002년 11월 4일

Then the protocol is represented as an ordered set of relation whose execution is guarded by a condition or by a time delay. PFG (Protocol Flow Graph) method proposed by Madsen and Hald [3] presents interface protocol by progress of cycles and events of signal. In event graph method taken by Borriello and Katz [4], nodes means event of signal, edge means order between nodes. Event graph method presents interface protocols by these nodes and edges. One of approaches is PIG proposed by Passerone, Rowson and *et al.* [5]. In PIG, the protocol is described in regular expression that is equivalent with FSM. Then the PIG computes the cross product machine of two interface protocols. PIG makes the wrapper from the cross product machine generated with some pruning schemes. These methods have limitation in the synthesis of wrappers for IPs having multiple transactions. For example, PIG has a possibility of state explosion when each interface protocol has multiple transactions. To overcome this problem, we propose an interface design methodology using IPCs, where wrappers consist of IPCs and a core. The core may be designed easily based on the correspondence between transactions of IPs. We will show the procedure in section 3.



(a) Master IPC



(b) Slave IPC

(Figure 1) IPC

Test benches for slave IPs incorporate master IPCs. BFM also includes a master IPC. IPCs can be reused if it is represented in a formal description. For example, TBV (transaction-based verification) [6, 7] raises the level of writing test benches by abstracting the interface operations in transactions using the object-oriented concept. TVM (transaction verification model) [7] in TBV methodology corresponds to an IPC. But, they are used only for simulation. SystmeC [8] that are based on C++ has a transactor concept or class that is similar to IPC. Transactor seems to be only for simulation. SpecC [9] may not have construct for interface behavior or IPC itself. The reuse of IPCs seems to be important in the sense of IP reuse. IPCs are usually assumed to be simulated but not necessarily synthesizable. However, those components can be reused in the design of interface protocol conversion circuits, e.g. wrappers, if they are synthesizable. This paper presents a transaction-oriented interface protocol description language that models the IPC that is generated as a synthesizable VHDL entity.

The proposed language differs from TVM in several points. First of all, the major application of the language is for the generation of the synthesizable IPCs while TVM is designed for simulation. As mentioned above, we classify IPCs as master or slave. Additionally, we define a generic scheme of generating interface ports between core and IPC based on transactions and their arguments. To reduce the number of generated ports, we add a construct to share ports among several transaction arguments. We employ C-like syntax since hardware engineers as well as software engineers are usually familiar with C language. IPCs in a higher-level form can be easily converted to VHDL or Verilog code with various external interfaces according to designers' demands. Synthesizable IPCs generated from the description can be used in the design of interface protocol conversion circuits, e.g. wrappers. We believe that the proposed approach reduces re-works on the interface protocol components. We show that the interface design using IPCs does not cause significant area overhead compared with the interface design without IPCs.

In section 2, we illustrate the proposed language with a UTOPIA [10] transmit interface protocol described in the master side, which is followed by the brief explanation on the generation steps and the general structure of a synthesizable IPC as well as the port sharing with the core port construct. In section 3, we illustrate how the synthesizable IPCs can be used in the interface design using a PVCI [11] compatible DES wrapper. We show the comparison result of the interface design using IPCs with the interface design without IPCs, based on the three different-sized wrapper design examples. Finally, we summarize this paper and describe future works.

2. An IPC Modeling Language

2.1 Syntax

We use a master side interface protocol description for the UTOPIA transmit interface shown in (Figure 2) to illustrate the proposed language constructs. UTOPIA transmit interface protocol is employed to transfer a cell (53 bytes) from ATM layer to physical layer.

There are two types of IPCs, i.e. master and slave. The keyword 'master' ('slave') indicates that the interface protocol following the keyword is described in master (slave) side. IPC must communicate with IP side in cycle accurate level while it corresponds with the core side in transaction level. IPC has interface ports with IP side and core ports in core side. As shown in the portion (1) of (Figure 2), interface ports are described first. Then, in portions labeled with (2) and (3), reset and clock signals are listed with some optional parameters such as 'low' (active low reset), 'async' (asynchronous reset), 'single' (single edge clocking), and etc. Transaction construct labeled with (4) is used to describe transaction name, transaction arguments, and its behavior. Transactions can be defined one by one or in a combined description. In the latter case, prototype declarations of the corresponding transactions should precede the combined transaction definition. Transaction arguments are sent or received via core ports that are made during the synthesizable IPC generation steps. Port sharing with the core port construct is described in section 2.2.

```
interface master TxMaster {
            out bit TxSOC ;     /* (1) interface ports */
            out bit TxEnbn ;
            out byte TxData ;
            out bit TxClk
            in bit TxFulln ;
            in bit TxClav ;
  reset rst_n low async ;          /* (2) reset */
  clock clock ( TxClk ) single ;   /* (3) clock */
transaction Transmit ( out FIFO byte Data [53] ) {
     int i, j ;                    /* (4) transaction definition */
     while ( TxClav != 1 )
          wait_edge ( clock, POS ) ;    /* (5) cycle boundary */
      TxEnbn = 0 ;  TxSOC = 1 ;
      Data.delete = '1' ;               /* (6) FIFO operation */
      wait_edge ( clock, POS ) ;
      TxSOC = 0 ;
      for ( i = 1 ; i <= 52 ; i++ ) {
           if ( TxFulln == 0 ) {
              for ( j = i ; j < i + 4 ; j++ ) {
                  Data.delete = '1' ;     /* (6) FIFO operation */
                  assert ( TxFulln = 0 ) ;    /* (7) assertion */
                  wait_edge ( clock, POS ) ;
              }
              TxEnbn = 1 ;
              i = j - 1 ;
              wait_edge ( clock, POS ) ;
          }
          TxEnbn = 0 ;
          Data.delete = '1' ;          /* (6) FIFO operation */
          wait_edge ( clock, POS ) ;
      }
      netlists {                       /* (8) netlist construction */
            TxData <= Data ;
      }
   }
}
```

(Figure 2) A UTOPIA transmit interface protocol description

We assume that each transaction argument is associated with the same type of a register or a FIFO that may reside in the core side. The direction 'in'('out') of the transaction argument means the core is to receive (send) the argument. The transaction 'Transmit' has single FIFO argument 'Data' that may contain 53 bytes. Local variables such as 'i', and 'j' may be defined within the transaction definition.

We employ C-language like syntax to describe the behavior of transactions, since hardware engineers are usually familiar with C-language. The transaction description is in the cycle-accurate level. The cycle boundary is indicated by 'wait_edge' statement. The statement wait_edge (clock, POS) corresponds to the rising clock edge and wait_edge (clock, NEG) to the falling clock edge. Basically, a transaction description may be considered as a single VHDL behavioral process statement with multiple 'wait's and control constructs. Therefore, one transaction definition corresponds to an ASM (Algorithmic State Machine) chart description [12], i.e. one FSM (Finite State Machine). For example, statements starting from an wait_edge (clock, POS) just before the next wait_edge (clock, POS) constitute a transition action for the state corresponding to the former wait_edge (clock, POS). Interface protocols requiring two concurrent FSMs may be described with two separate protocol descriptions.

The transaction starts when TxClav (cell available signal) is asserted. Each byte from the FIFO Data is transferred to TxData port in the statements labeled with (6).

In order to describe interface protocol behaviors dependent on the status of the core, we allow the status checking expressions such as Data.full, Data.empty, and Data.valid. Data.full (Data.empty) means that the corresponding FIFO is full (empty). Data.valid means the corresponding register has a valid value.

IPC can be used in simulations to verify the correctness of IP interface protocols. The assert() statement labeled with

(7) is useful to detect the protocol violations during simulation.

The 'netlist' construct labeled with (8) is added to reduce the area of the generated IPC. The assignments in netlist construct are converted to concurrent signal assignments in VHDL. The effectiveness of the netlist construct is shown in <Table 1> of section 3.2.

2.2 Synthesizable IPC

The synthesizable IPC generation steps from interface protocol description are organized as follows

1) Parsing and constructing control flow graph
2) Processing variables (when the description contains variables)
3) Processing to avoid the inferred latches or unexpected storages
4) Generating VHDL code

First, we parse the description and construct a control flow graph. Then, if the description contains variables, the generator executes a special procedure to deal with variables. In step 3, the generator executes a step to avoid inferred latches or unexpected storages in generated IPC. Finally IPC is generated in synthesizable VHDL.

The generated IPC consists of three processes and concurrent signal assignment statements. The first process is a sequential VHDL process executed when it meets the rising or the falling clock edge. This process causes flip- flops and internal registers to be inferred. In addition, the process resets the values of corresponding signals when the reset signal is activated. The second process is a combinational process that determines next state and outputs based on current state and inputs. The third process is also a combinational process where combinational signals are generated to deal with variables. A single variable in C language usually corresponds to a register and a number of combinational signals in VHDL code. The concurrent signal assignments are generated from the 'netlist' construct that directly bypasses an interface port value to a core port or transaction argument port, or vice versa. It can help reduce the area of synthesizable IPCs as shown in section 3.2.

The generated IPC has two sets of interface ports. One set of interface ports includes ports that are employed to execute or to recognize transactions. The other set of interface ports is generated to communicate with the core based upon transactions. The latter ports includes TRCODE (transaction code), TREND (indicating the end of the current transaction), and ports necessary to send or to receive transaction arguments.

Besides the signals corresponding to arguments, we need additional signals to let the core know the timing when arguments are sent to the core, or arguments are requested from the core. For the example in (Figure 2), Data_request signal is generated to let the core decrement the counter or front pointer of FIFO. Data_load signal will be generated when IPC writes Data to the core. Both signals are asserted at the same cycle when the corresponding data is read from or written to the core. In addition, we generate the signals such as Data.full, Data.empty, and Data.valid only when such expressions are used in the transaction description. Those expressions are necessary to check the status of the register or the FIFO corresponding to an argument Data.

Too many similar ports may be generated in cases where many similar transactions are defined. To avoid the proliferation of ports, we add a construct starting with keywords 'core port' to share ports among transaction arguments. (Figure 3) shows an example usage of 'core port' constructs. DES IP has two transactions ENCR and DECR. The 'core port' construct at the end of (Figure 3) defines two new ports 'odata_p' and 'idata_p'. The port 'odata_p' is shared by four transaction arguments and the port 'idata_p' by two transaction arguments. The timing signals 'request' and 'load' should be redefined considering the port sharing. For example, 'odata_p_request' signal will be two bits wide to represent the order number of the corresponding transaction argument. A possible encoding of 'odata_p_request' would be as follows : "00" represents no request, "01" ("10") means the first (the second) argument, and "11" is not used. We do not need to incorporate the transaction code into the encoding since the code comes from the core (in case of master IPC), or the code has already been notified to the core (in case of slave IPC).

```
interface master DES {
      ...
    out bit [ 63 : 0 ] pkey, ptext ;
      ...
transaction ENCR ( out bit [ 63 : 0 ] key, out bit [ 63 : 0 ] data,
in bit [ 63 : 0 ] cdata) {
            ...
            pkey = key ;
            ...
            ptext = data ;
            ...
}
transaction DECR ( out bit [ 63 : 0 ] key, out bit [ 63 : 0 ] cdata,
                        in bit [ 63 : 0 ] data ) {
            ...
            ptext = cdata ;
            ...
}
core port ( out bit [ 63 : 0 ] odata_p =
{ ENCR.key, ENCR.data, DECR.key, DECR.cdata },
in bit [ 63 : 0 ] idata_p = { ENCR.cdata, DECR.data } ) ;
}
```

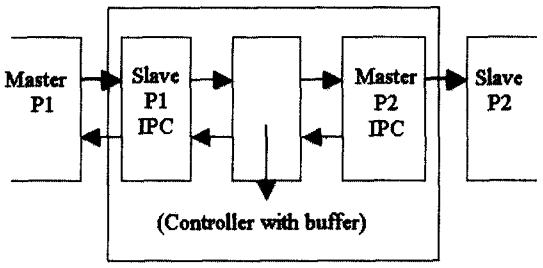
(Figure 3) A UTOPIA transmit interface protocol description

In the core side, the signal 'odata_p_request' may be used to select one of four registers corresponding to four transaction arguments to generate the signal 'odata_p'.

3. Wrapper design using IPCs

3.1 A PVCI Compatible DES Wrapper Design

The following (Figure 4) shows how IPCs are used in the design of the interface circuit between two different interface protocols, e.g. wrappers. Any interface circuit between two different interface protocols may take the similar structure consisting of a slave or target IPC, a master or initiator IPC, and a controller with buffers or registers.



(Figure 4) A general interface circuit structure between two different interface protocols P1 and P2

A DES wrapper compatible with PVCI[1] may take the same structure, where P1 (P2) becomes PVCI (DES). Slave PVCI IPC recognizes PVCI transactions such as 'READ' and 'WRITE' while master DES IPC executes transactions such as 'encryption' and 'decryption' according to the request of the controller. PVCI is different from DES in several points. For example, PVCI employs address-mapped transfer while DES has no address-map. So, there should be some kind of descriptions that translate PVCI addresses to DES functions, for the design of the interface circuit. The following (Figure 5) shows a specific PVCI protocol description and a DES protocol description that are used to build a wrapper design example between PVCI and DES protocol.

```
Interface slave PVCI {
        out bit ACK ;     out bit [ 31 : 0 ] RData ;
        in bit VAL ;      in bit RNW ;
        in bit [ 7 : 0 ] ADDRESS ;    in bit EOP ;
        in bit [ 31 : 0 ] WData ;
transaction READ ( in bit [ 31 : 0 ] ADDR,
                    out bit [ 31 : 0 ] Data) ;
transaction WRITE ( in bit [ 31 : 0 ] ADDR,
                     in bit [ 31 : 0 ] Data) ;
core port ( out bit [ 7 : 0 ] addr_p = { READ.ADDR,
                                    WRITE.ADDR } ) ;
}
interface master DES {
            out bit start ;            out bit enc_dec ;
            out bit [ 63 : 0 ] pkey ;
            out bit [ 63 : 0 ] ptext ;
            in bit done ;       in bit busy ;
            in bit [ 63 : 0 ] ctext ;
transaction ENCR ( in bit [ 63 : 0 ] key, in bit [ 63 : 0 ] data,
                                        out bit [ 63 : 0 ] cdata) ;
transaction DECR ( in bit [ 63 : 0 ] key, in bit [ 63 : 0 ] cdata,
                                         out bit [ 63 : 0 ] data) ;
core port ( out bit [ 63 : 0 ] key_p = { ENCR.key, DECR.key },
             out bit [ 63 : 0 ] odata_p = { ENCR.data,
                                    DECR.cdata },
             in bit [ 63 : 0 ] idata_p = { ENCR.cdata,
                                    DECR.data } ) ;
}
```

(Figure 5) A part of interface protocol descriptions of DES master and PVCI slave

PVCI has a different data width from DES. PVCI compatible DES wrapper has to do several conversion tasks. WRITE transactions in PVCI side may result in storing 'Data' argument in some registers in the wrapper. ENCR (DECR) transaction can be started only when 'key' and 'data' ('cdata') arguments are ready to send. Encrypted 'cdata' or decrypted 'data' argument may be stored in a register, which is then read out by READ transaction. We can formally describe these kind of wrapper behaviors as in (Figure 6), where registers are declared and used in pairing the transactions of both sides. PVCI transactions deal with 32-bit registers while transaction arguments of DES have 64-bit data width. Therefore, two 32-bit registers, e.g. key[0] and key[1], are combined to form an argument 'key' to DES transaction. Uppercase strings such as K0ADDR, K1ADDR, and etc are address constants of PVCI. The second statement describes that a series of transactions of the slave PVCI side corresponds to a transaction ENCR() of the master DES side. WRITE (K0ADDR, key[0]) means that 'Data' is written to a register key[0] when WRITE transaction to the address

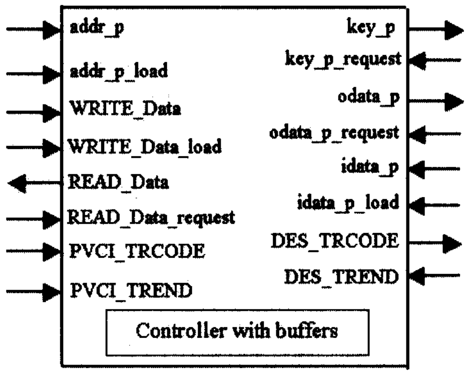
1) PVCI is a simplest version of VCI [5] protocol to be used between IP and bus agent module.

K0ADDR is recognized at the slave PVCI side. Data availability drives the execution of transactions in master DES side, i.e. ENCR or DECR starts only when the registers key and text have valid data.

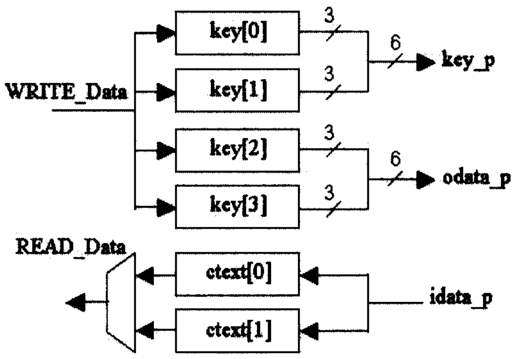
```
bit [31 : 0] key[2], text[2], ctext[2] ;
WRITE ( K0ADDR, key[0] ), WRITE ( K1ADDR, key[1] ),
WRITE ( T0ADDR, text[0] ), WRITE ( T1ADDR, text[1] ),
READ ( C0ADDR, ctext[0] ), READ ( C1ADDR, ctext[1] )
   : ENCR ( key, text, ctext ) ;
WRITE ( K0ADDR, key[0] ), WRITE ( K1ADDR, key[1] ),
WRITE ( C0ADDR, text[0] ), WRITE ( C1ADDR, text[1] ),
READ ( T0ADDR, ctext[0] ), READ ( T1ADDR, ctext[1] )
            : DECR ( key, text, ctext ) ;
```

(Figure 6) A wrapper behavior description

The input or output ports of the wrapper can be generated as in (Figure 7), based on the wrapper behavior description. The port 'p_load' ('p_request') is generated if the transaction argument 'p' is an input (output) to the wrapper.



(Figure 7) The I/O ports of the wrapper



(Figure 8) The data path of the wrapper

(Figure 8) shows the data path of the wrapper. Other signals that do not appear in (Figure 8) are used to generate control signals to the data path. The signal 'p_load' will be used to generate the load signal to the register where the argument 'p' is to be stored. The address signal 'addr_p' is used to select one of the registers where the signal 'WRITE_Data' is to be stored. The signal 'p_request' will be used in the similar way. The signal 'READ_Data_request' together with the signal 'addr_p' is used to generate the select signal to the MUX.

Different address-to-register mapping in the wrapper behavior description may lead to different data path. The mapping also affects the generation of load signals to the corresponding registers.

Besides registers, we need to add flip-flops indicating the data validities of the registers in the data path.

3.2 Experiments and Discussions

<Table 1> lists the interface protocol descriptions translated into the corresponding IPCs in synthesizable VHDL with our IPC generator. The generator was implemented in about 6000 lines of C language code on Solaris 2.7 operating system. <Table 1> summarizes experimental results. The column 'master/slave means' the corresponding description is in master/slave side. The column '# lines' means the number of lines of the interface protocol descriptions. The columns '# FFs' and Area indicate the number of flip-flops and the area respectively when the generated IPC is synthesized with the design compiler of Synopsys Inc. with HYNIX 0.35um stand cell (Area-A) and ALTERA FLEX10k (Area-B) as the target library. The columns Area-A and Area-B are the total area of the generated IPCs from the descriptions employing 'netlist' constructs while the column Area-B' is the area from the descriptions without 'netlist' constructs. By comparing the columns Area-B and Area-B', we notice that the 'netlist' construct is effective in reducing the area of the generated IPCs.

We observed that no latches are inferred. Our algorithm generates VHDL codes so that latches or unnecessary storage elements are not inferred. For example, UTOPIA transmit master interface description shown in (Figure 2) results in 17 bits of flip-flops. The list of variables or ports inferred as FFs are 'i' (6 bits), 'j' (6 bits), 'TxEnbn' (1 bit), 'TxSoC' (1 bit), and FSM state (3 bits).

DES interface has three 64-bit ports with the core and another three 64-bit ports as interface ports as shown in (Figure 5). Therefore, the synthesized IPCs take relatively large area.

We also verified the functionality of the PVCI-to-DES wrapper shown in the section 3.1. Our proposed language assumes that registers or FIFOs exist in the core side for the transaction arguments. This naturally leads to the wrapper architecture with registers or FIFOs between two IPCs such as in PVCI-to-DES or 60x-to-PVCI of <Table 2>.

<Table 1> The list of interface protocol descriptions

Protocol name	master /slave	# lines	# FFs	Area-A	Area-B	Area-B'
DES	Master	44	8	115	62	253
	Slave	55	9	131	64	259
PVCI	Master	52	3	47	24	123
	Slave	57	5	49	31	134
UTOPIA Tx	Master	46	17	292	157	178
	Slave	38	16	204	111	137
UTOPIA Rx	Master	52	11	195	104	112
	Slave	48	11	169	100	108
Wishbone	Master	51	3	45	33	220
	Slave	56	6	46	37	268

In some cases, we need to directly connect two IPCs without registers or FIFOs in between. It means that a transaction in the slave side has been recognized and run concurrently with a transaction in the master side. Both transactions have an ability to insert wait states according to the status of registers or FIFOs in the controller. For example, we can connect an input port 'a' in the slave side directly to an output port 'b' in the master side if the load signal 'a_load' is connected to the valid signal 'b_valid'. The example PVCI-to-SRAM shown in <Table 2> is one of such examples.

<Table 2> The comparison of IPC-based interface design with non-IPC versions

Wrapper Name	Non-IPC (Total area)	Using IPC (Total Area)		
		IPC1 (Slave)	IPC2 (Master)	Core
PVCI-to -DES	2479	2498		
		PVCI : 49	DES : 115	Core : 2334
60x-to -PVCI	398	463		
		60x : 204	PVCI : 47	Core : 212
PVCI-to -SRAM	154	208		
		PVCI : 49	SRAM 157	Core : 0

The designers may reduce the interface design times by employing the interface design methodology using IPCs, since they do not need to understand the detailed signaling of the interface protocols of the corresponding IPs. IPCs can be considered as cells of a library. However, the proposed method may be considered as to cause severe area overhead. In order to see how much area overhead the proposed method can cause, we performed two wrapper design experiments, one with IPCs and the other without IPCs. In the former design experiments, we employ IPCs and design only the core portion. In the latter experiments, we design the whole wrappers without employing IPCs. In <Table 2>, the column 'Non-IPC' indicates the total area obtained with the latter approach while the column 'Using IPC' is the total area by the former approach. The HYNIX 0.35um standard cell library is used to run the Synopsys design compiler as the target library. Though the former approach shows relatively large area overhead in the small wrapper designs, it seems to show the similar area as the wrapper size or its behavior becomes complicated. This means that the area overhead caused by employing the interface design using IPCs is not so significant compared with the interface design without employing IPCs, especially when the wrapper becomes large.

4. Summary And Future Works

This paper proposes a methodology for the interface circuit design using synthesizable IPC that can be re-used. We present a transaction-oriented interface protocol description language and a method to convert the description into an IPC in synthesizable VHDL code. With experiments, we show that the interface design using IPC does not cause significant area overhead compared with the interface design without IPC. The proposed IPC-based approach can be employed to reduce the interface design time since the designers can reuse IPCs without understanding the detailed interface protocols.

One of our future works is to realize the proposed wrapper design methodology based on the synthesizable IPCs. IPCs can also be used to construct bus functional models and protocol compliance checking modules.

References

[1] Ben Cohen, "VHDL Answers to Frequently Asked Questions," Kluwer Academic Publishers, 1997.
[2] Sanjiv Narayan and D. D. Gajski, "Interfacing incompatible protocols using interface process generation," Proc. of DAC, pp.468-473, 1995.
[3] Jan Madsen and Bjarne Hald, "An Approach to Interface Synthesis," in Proc. of ISSS, 1995.
[4] Gaetano Borriello, and Randy H. Katz, "Synthesis and Op-

timization of Interface Transducer Logic," Proc. ICCAD '87, pp.274-277, 1987.

[5] Roberto Passerone, James A. Rowson, Alberto Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in Proc. Of DAC '98, 1998.

[6] D. S. Brahme, et. al, "Transaction-Based Verification Methodology," Cadence Berkeley Labs, Technical Report # CDNL-TR-2000-0825, Aug., 2000.

[7] Cadence Design Systems, Inc., Transaction-Based Verification : TestBuilder Reference Manual, Sep., 2001.

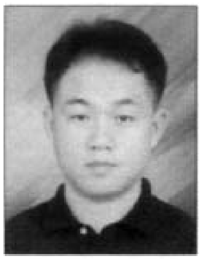
[8] User guide SystemC 2.0.

[9] Rainer Domer, Andreas Gerstlauer, Daniel Gajski, SpecC Language Reference Manual Version 1.0, March, 2001.

[10] TranSwitch Corporation, UTOPIA Interface for the SARA Chipset, Application Note, Document Number TXC-05501-0002-AN, 1.0, April, 1995.

[11] R. Lysecky, F. Vahid, T. Givargis, "Experiments with the Peripheral Virtual Component Interface," International Symposium on System Synthesis, 2000.

[12] M. Moris Mano, "Digital Design," 2nd Edition, Prentice-Hall International, Inc., 1991.



윤 창 열

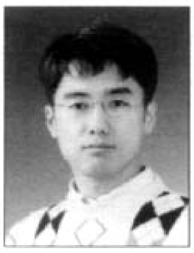
e-mail : yun@ce.cnu.ac.kr

2000년 한남대학교 컴퓨터공학과 졸업 (학사)

2002년 한남대학교 대학원 컴퓨터공학과 (공학석사)

2002년~현재 충남대학교 대학원 박사과정

관심분야 : VLSI 설계자동화, 컴퓨터구조



장 경 선

e-mail : ksjhang@computer.org

1986년 서울대학교 전자계산기공학과 졸업 (학사)

1988년 서울대학교 대학원 컴퓨터공학과 (공학석사)

1995년 서울대학교 대학원 컴퓨터공학과 (공학박사)

1996년~2001년 한남대학교 컴퓨터공학과 전임교원

2001년~현재 충남대학교 정보통신공학부 조교수

관심분야 : VLSI 설계자동화, 컴퓨터구조