Interface Synthesis for IP Based Design

Bong-Il Park, In-Cheol Park, and Chong-Min Kyung
Div. of EE, Dept. of EECS, KAIST, Taejon 305-701, Korea
E-mail: bipark@duo.kaist.ac.kr, {icpark,kyung}@eekaist.kaist.ac.kr

Abstract— **In system-on-a-chip design, interfacing of Intellectual Property(IP) blocks is one of the most important issues. Since most IPs are provided by different vendors, they have different interface schemes and different operating frequencies. In this paper, we propose a new interface synthesis method that enables one not only to handle the interface between IPs with different operating frequencies but also to minimize the hardware resource required for the interface. We have demonstrated the proposed algorithm by applying it to a real design example, MP3 decoder, and verified the IIS-to-PCI protocol converter on a real hardware system.**

I. Introduction

Steady advances in design methodology and semiconductor technology has allowed the complexity of a single chip to contain more than one million transistors [1]. Time to market issue in the complex system-on-a-chip(SOC) design can only be solved by dealing with complex building blocks commonly called Intellectual Properties(IPs). In the design of SOC, IP-based design methodology is prevailing and requires large design space exploration as well as efficient design verification. However unfortunately most IPs are designed with a different interface protocol.

The problem of interface synthesis, as originated from the network protocol conversion, has been addressed in various literatures [2–8]. In computer communication networks, it is often necessary to connect network components which stand on different network architectures. Many researchers have done the several works with an analysis of conversion schemes [2, 3]. However, most approaches in this field have been mainly focused on the software aspects without considering the hardware aspects.

Borriello and Katz [9] introduced the event graph to establish the correct synchronization and data sequencing. The approach proposed by Sun and Brodersen [7] provides a library of components to free the user from considering lower-level details, which is very similar to the ASIC design process. The approach proposed by Akella and McMillan [6] provides a protocol specification consisting of two FSMs for describing the producer/consumer part of the interface and specification describing the valid state transitions of the interface.

Since most IPs are provided by different vendors with different design environment, they have inevitably different characteristics which include operating frequency, interface schemes, etc. However, most works related to interface synthesis assume the two communicating parties are driven by the same clock. In this case, if a designer has some IPs with mutually different operating frequencies, all the IPs must be operated at lowest frequency causing the performance degradation. In our approach, this problem is overcome by allowing different clocks in communication and optimizing the hardware resource. Supporting different clocks produces some advantage. First, it is possible to increase the number of candidate IPs to be integrated on a chip. The assumption of identical operating frequency is a too hard constraint for selecting IPs and produces poor design quality in the eventual SOC design. Next, our approach directly synthesizes interface logic from the source description without any additional modification for frequency matching, which produces better performance and ease of use.

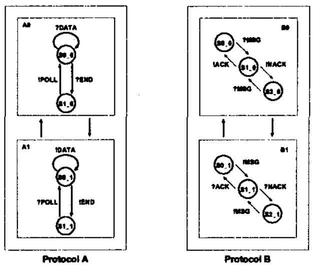

Fig. 1. An example of each of the two model protocols: each protocol is represented as a pair of communicating FSMs where protocol A is the so-called *polling model* and protocol B is the *ack-nack model.*

This paper is organized as follows. In Section II, we formulate the interface synthesis problem and present interface synthesis for the system-on-a-chip with considering different operating frequencies. Section III describes new algorithms developed for the selection of operating frequency and internal queue sizing. The results of application of the proposed algorithm to the MPEG decoder is shown in Section IV, followed by the conclusion.

II. Interface Synthesis between IPs with Different Operating Frequency

In this section, we introduce the interface synthesis process and propose a new process for handling IPs with different operating frequencies.

A. Problem formulation

Fig. 1 shows a simple example of two protocols where each protocol is shown as a pair of communicating finite state machines; Protocol *A* is the so-called *polling model* and protocol *B* is the *ack-nack model.* Each state is denoted by a circle with the state name and transition conditions and outputs shown on edges. The string with '?' character represents the transition condition and the string with '!' character means the output.

The aim of interface synthesis is to allow the communicating component of one protocol to communicate with that of another protocol. In Fig. 1, let us assume that A_0 and B_1 need to communicate with each other. To synthesize an interface between A_0 and B_1 is to find a new FSM C between A_0 and B_1 as shown in Fig. 2. In Fig. 2, A_0 thinks that it is communicating with A_1 which consists of C and B_1. Similarly, B_1 regards C and A_0 as B_0. Therefore interface synthesis is to find such an FSM C for given protocols A and B. In addition, the proposed approach do not have any constraint for operating frequency, while the given two protocols are operated at the same operating frequency in conventional approaches.

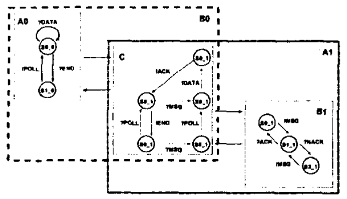

Fig. 2. An example of interface logic between A_0 and B_1: C and B_1 mimics A_1, while C and A_0 mimics B_0

B. IPs with different operating frequencies

In Fig. 3, let us assume that the producer,A_0, is operated at the clock frequency f_0, while the consumer,B_1, is operated at f_1. Let us assume that f_0 is smaller than f_1, that is, the producer is operated at lower clock frequency without loss of the generality. The simplest approach for the protocol converter is very trivial. All the LTSs, *i.e.*, the producer, the consumer, and the protocol converter, are operated at the same clock, which is the slowest clock. This approach does not cause any problem except for the performance degradation.

The next approach provides the better performance by allowing different clock frequencies. The producer and the consumer are operated at their own operating frequency, f_0 and f_1 respectively, while the protocol converter is operated at certain frequency which may be either f_1 or f_0. However, this solution causes some problems. If it is operated at f_1, the protocol converter receives too many redundant messages from the producer. As the clock period of the producer is longer than that of the protocol converter. On the other hand, it sends too many redundant messages to the consumer. In the following we classified the situation in two cases according to the value of $\frac{f_1}{f_0}$.

C. Case I: $\frac{f_1}{f_0}$ is an integer number

Fig. 3 shows the case when f_1 is some integer multiple of f_0. The protocol converter, which consists of two FSMs(A_0^d and B_1^d), operates at f_1, because if it operates at the lower clock frequency, f_0, the signal of the fast FSM(B_1) changes within a clock period and , therefore, interfacing is not working. Thus it operates at the higher clock frequency, f_1. Since the interface between B_1 and B_1^d operates at the same clock frequency, we do not need any special consideration. However, some modifications are necessary between A_0 and A_0^d.

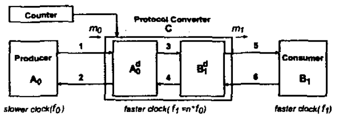

Fig. 3. Case I: $f_1 = n \times f_0$, where n is an integer.

Interface synthesis is performed after the states and edges are modified according to the following path. Thus let us focus on the output generation paths of A_0^d which can be classified into the following four paths:

Path 1(signal 1$\rightarrow$ signal 3): For this path the input signal(signal 1) has the longer period than the output signal(signal 3). An obvious and efficient solution for this case is to insert a frequency matching counter which counts from 0 to $(n-1) = (\frac{f_1}{f_0} - 1)$. If f_0 and f_1 are 10MHz and 20MHz, respectively, an input signal has to be sampled at 10MHz. Otherwise, *i.e.*, if sampled at 20MHz, the FSM of A_0^d transits twice as if an input signal has been asserted for two successive cycles. This situation can be avoided with the clock cycle counter shown in Fig. 3. The output of the counter is asserted to enable input sampling at every $T_0(= n \cdot T_1)$. T_0 and T_1 represents the clock periods corresponding to f_0 and f_1, respectively. In Fig. 4 which shows the modified FSM for frequency matching, (a) shows an original FSM which waits for the signal `poll` and transmits `data` until tie signal `last` is asserted, while (b) shows the modified FSM in which the transition of state occurs only when the output signal of the counter(`cntN`) is asserted.

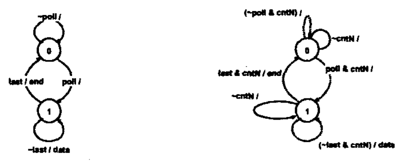

Fig. 4. Path 1: (a) original FSM, and (b) Modified FSM for frequency matching.

Path 2(signal 4 $\rightarrow$ signal 3): This path does not need to be considered since the input(signal 4) and output(signal 3) signals are connected between the inter module of the protocol converter operating at the same operating frequency.

Path 3(signal 1 $\rightarrow$ signal 2): Similar to the path 1, the input signal needs to be sub-sampled. In addition, since the output signal is the input to A_0, it must be extended to cover one T_0. The transition edges of the FSM corresponding to input signals are modified in the same way as path 1, and those corresponding to output signals are also inserted. The inserted edges help assert the output signals during one T_0. In Fig. 5, newly inserted transition edges are represented as solid arrows.

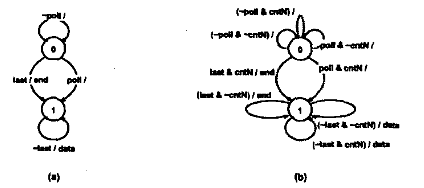

Fig. 5. Path 3: (a) original FSM, and (b) its Modified FSM for frequency matching obtained by inserting new edges

Path 4(signal 4 $\rightarrow$ signal 2): This requires more complex modifications which include some new states and transition edges. In this path, the input signals(signal 4) are valid during only T_1. If an input signal is changed, this information is stored within a new state and is restored when the two clocks are matched every $T_0(= n \cdot T_1)$. Thus one input signal makes two new states which are used for storing a signal transition and asserting an output signal during $T_0(= n \cdot T_1)$. In the example of Fig. 6(b), if the signal `last` changes and the current state is '`1`', state transits to '`1_1`' or '`1_2`' according to the time when the signal changes. Then new input signal can be asserted after the output signal of the counter(`cntN`) is asserted.

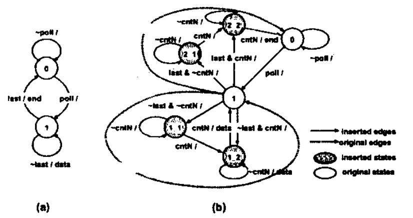

Fig. 6. Path 4: (a) original FSM, and (b) its Modified FSM obtaining by inserting new states and edges

D. Case II: $\frac{f_1}{f_0}$ is a real number

If the ratio, $\frac{f_1}{f_0}$, is not an integer but a real number, then the several solutions are possible. The simplest solution is that both IPs are operated at the lower clock frequency. This requires no additional hardware although it degrades some performance. The next solution is to find the least common multiplier(LCM) of both clock frequencies since any real number can be represented as a ratio of two integers. For example, the ratio of 1.5 can be represented as 2:3. Then the LCM clock is inserted to all the blocks including the protocol converter, the producer, and the consumer. In this case, the interface between the protocol converter and the producer(consumer) is synthesized as in the case I. Thus the protocol converter needs two frequency matching counters as shown in Fig. 7.

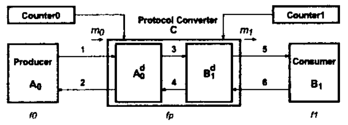

Fig. 7. Case II: $f_0 = r \times f_1$, where r is a real number

However, if the two clock frequencies are slightly different, the LCM clock may be infeasible to implement. For example, if f_0 and f_1 are 59MHz and 60MHz respectively, then the 3.54GHz clock must be generated! It is nearly impossible to make such high frequency clock. In this case, the clock frequency should be slightly changed a *priori* to make the LCM a reasonable value.

III. More Considerations for Interface Synthesis

Until now, we assumed that IPs operate at certain single clock frequency. However, each IP has generally its operating range, bounded by the lowest and the highest operating frequency. In this condition, we must first determine the operating frequency. Fortunately, if the operating ranges of two IPs are overlapped, we can just select any frequency within the overlapped range. In our interface synthesis approach, if two IPs are operated at different frequencies, then we must find an LCM frequency. Here our synthesis approach is extended to the case having the operating range of IPs.

A. Determining of operating frequency

Fig. 8 shows the pseudo code for determining the operating frequency. In this algorithm, inputs of the algorithm are the operating ranges of two IPs ,*i.e.*, a range,(f_{L1}, f_{H1}) and another range,(f_{L2}, f_{H2}). The algorithm determines the common operating frequency which equals an integer times the operating frequency of each IP. In the inner **while** loop, it searches an overlapping range between the ranges as the slow IP range is widened by multiplying an integer number. If the overlapping range cannot be found in the current range of the fast IP, the fast range is widened in the outer **while** loop. This iteration is continued until the overlapped range is found. For example, one IP operates from 15MHz to 20MHz, and the other IP operates from 42MHz to 44MHz. Then, in the first inner loop, overlapped ranged is not found, since the integer multiple ranges of the lower IP are (30MHz, 40MHz), (45MHz, 60MHz), etc. In outer loop, the range of the fast IP is changed into (84MHz, 88MHz). The range is overlapped with (75MHz,100MHz) which equals the five times of the range (15MHz, 20MHz). Finally, the frequency, 88MHz, is determined as the operating frequency.

```
input: $f_{L1}, f_{H1}, f_{L2}, f_{H2}$
output: $f_{common}$
while $f_{L2} <= f_{max}$ do
    while $n \cdot f_{L1} <= m \cdot f_{H2}$ do
        if ($n \cdot f_{L1}, n \cdot f_{H1}$) overlap with ($m \cdot f_{L2}, m \cdot f_{H2}$)
            return $f_{common}$ = min( $n \cdot f_{L1}, m \cdot f_{H2}$ );
        else
            $n = n + 1$;
        end if
    end while
    $m = m + 1$;
end while
return $f_{common}$ = can_not_find;
```

Fig. 8. Algorithm for determining of operating frequency, where assume that $f_{L1} < f_{L2}$ and $f_{H1} < f_{H2}$

B. Determining of queue sizing

Some protocols may have strict timing constraints between one message and the next message. If IPs based on these protocols are communicated through the protocol converter and have different operating frequencies, the protocol converter needs internal storage elements. As shown in Fig. 3, if a message must be sent every three clocks and received every single clock, the protocol of the consumer can be violated because messages are not available in time. This problem is solved with an internal queue which first stores all messages for sending and sends the stored messages according to the consumer protocol. In this case, the objective is to minimize the internal queue size which can be formulated as follows:

$$\text{Queue Size} \quad = \quad D_{total} - \lfloor D_{total} \frac{R_{m0}}{R_{m1}} \rfloor \qquad (1)$$

where D_{total} is the total amount of message sent, R_{m0} represents the rate of message sending, R_{m1} represents the rate of message reception, where we assumed that receiving rate is higher than the sending rate.

If a queue size equals D_{total}, then all data can be stored in the queue and the stored data are sent to the consumer sequentially. Since a queue can handle receiving and sending of data concurrently, sending operation can be started just after certain amount of data is stored. In the equation 1, the right term, $\lfloor D_{total} \cdot R_{m0}/R_{m1} \rfloor$, corresponds to the data operated concurrently. For example, if we have a 256 bit messages and the sending rate and receiving rate are 1Mbps and 2Mbps respectively, then the required queue size is 128 bits. The consumer protocol starts just after 128 bit messages are all stored in the internal queue. After this time, the internal queue stores the remaining 128 bits and sends messages concurrently.

IV. Experiment

The proposed approach is applied to an MPEG 2 audio layer 3(MP3) player system through Virtual Chip System [10, 11]. In the MP3 decoder example, the interface part deals with three ports consisting of one IIC port, two IIS ports, and PCI interface connects between these ports and the software model. PCI interface operates at 33MHz, while one IIC and two IIS operate at 500KHz and 3MHz, respectively. Moreover, the data width of each protocol is not matched.

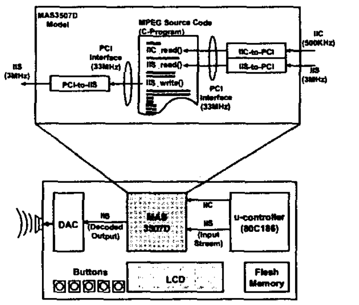

Fig. 9. The lower part shows the block diagram of an MPEG 2 audio layer 3(MP3) player system. MP3 decoder(MAS3507D) has three interface ports which consist of one IIC port, two IIS ports

The decoded bit stream must be supplied continuously to the D/A Converter(DAC) of the MP3 player system at the bit rate of 3Mbps using IIS protocol. If the decoded bit stream is infinite, an infinite size queue which stores all the decoded bit stream is required, which is infeasible. Moreover, since the MP3 software model requires time for decoding a input bit stream, a chunk of 16-bit data should be occasionally supplied through PCI interface. Therefore, the decoded bit stream must be divided by certain amount. In this example, the decoded bit stream is divided by 90.9K bits which correspond to 1M cycles at 33MHz as shown in equation 2, though the protocol converter must infinitely supply it. Thus the required queue size can be calculated as follows:

$$D_{total} = \frac{1M\text{-cycle}}{33MHz} \times 3M\text{bps} = 90.9K\text{bit} \quad (2)$$

$$R_{m0} = 3M\text{bps} \quad (3)$$

$$R_{m1} = \frac{33MHz \times 16\text{bit}}{P0 + P1} = 132M\text{bps} \quad (4)$$

$$\text{Queue Size} = D_{total} - \lfloor D_{total} \frac{R_{m0}}{R_{m1}} \rfloor = 88.8K\text{bit} \quad (5)$$

Equation 2 shows the IIS protocol during 1M cycle transfers 90.9Kbits. In equation 4, $(P0 + P1)$ means the number of clock for single PCI write transaction which requires minimum 4 cycles. Although this example does not show much reduction of queue size, if the decoded output stream has higher bit rate, queue size can be significantly reduced. As shown in Fig. 10, the IIS-to-PCI protocol converter is operated at 33MHz. Thus the PCI FSM makes a data valid signal for IIS protocol using a counter which counts from 0 to 10.

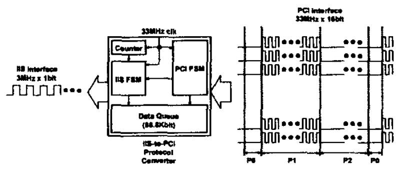

Fig. 10. IIS-to-PCI protocol converter: decoded bit stream is connected to DAC using IIS protocol which is 1bit serial output and is received from software using PCI protocol which consists of three phases. $P0$-phase is required to initiate PCI transaction, $P1$-phase is used to send decoded data, and $P2$ is the time for MP3 software decoding of new IIS data.

V. Conclusion

Integrating IPs that are developed by different vendors is an important issues in the SOC design, as the different design environment produces different interface protocols and operating frequencies. For the exploration of large design space in SOC design, the automatic interface synthesis is one of the most important part. The approach proposed in this paper can be useful to a designer not only to make automatic interface synthesis but also to consider multi-clock operating environment. This approach can be started with the operating frequency and queue size determined. Then the input FSM is modified according to the input and output path combinations. Finally the conventional interface synthesis is performed. This approach was successfully applied to Virtual Chip System in order to verify the IIS-to-PCI protocol converter on a real hardware system.

References

[1] Semiconductor Industry Association, San Jose, CA. *International Technology Roadmap for Semiconductors 1998 Update*, 1998.

[2] K.Okumura. A Formal Protocol Conversion Method. In *ACM Symposium on Communications, Architectures, and Protocols(SIGCOMM)*, pages 30–37, Aug. 1986.

[3] S.S.Lam. Protocol Conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, Mar. 1988.

[4] J.A.Nestor and D.E.Thomas. Behavioral Synthesis with Interface. In *IEEE/ACM International Conference on CAD*, pages 112–115, Nov. 1986.

[5] G.Borriello. *A New Interface Specification Methodology and its Applications to Transducer Synthesis.* Ph.D. dissertation, Univ. of California, Berkeley, 1988.

[6] J.Akella and K.McMillan. Synthesizing Converters between Finite State Protocols. In *IEEE International Conference on Computer Design*, pages 410–413, Oct. 1991.

[7] J.S.Sun and R.W.Brodersen. Design of System Interface Modules. In *IEEE/ACM International Conference on CAD*, pages 478–481, Nov. 1992.

[8] S.Narayan and D.D.Gajski. Interfacing Incompatible Protocols using Interface Process Generation. In *IEEE/ACM Design Automation Conference*, pages 468–473, Jun. 1995.

[9] G.Borriello and R.H.Katz. Synthesis and Optimization of Interface Transducer Logic. In *IEEE/ACM International Conference on CAD*, pages 274–277, Nov. 1987.

[10] N.Kim, H.Choi, S.Lee, I.-C.Park, and C.-M.Kyung. Virtual Chip: Making Functional Models Work on Real Target Systems. In *IEEE/ACM Design Automation Conference*, pages 170–173, Jun. 1998.

[11] C.-J.Park, S.Lee, B.-I.Park, H.Choi, J.-G.Lee, Y.-I.Kim, M.-K.Jung, I.-C.Park, and C.-M.Kyung. Early In-System Verification of Behavioural Chip Models. In *IEEE International High Level Design Validation and Test Workshop*, pages 61–65, Nov. 1999.