

Interface Synthesis for Heterogeneous Multi-Core Systems from Transaction Level Models

Hansu Cho, Samar Abdi, Daniel Gajski

Center for Embedded Computer Systems University of California, Irvine, CA USA
{hscho,sabdi,gajski}@uci.edu

Abstract

This paper presents a tool for automatic synthesis of RTL interfaces for heterogeneous MPSoC from transaction level models (TLMs). The tool captures the communication parameters in the platform and generates interface modules called universal bridges between buses in the design. The design and configuration of the bridges depend on several platform components including heterogeneity of the components, traffic on the bus, size of messages and so on. We define these parameters and show how the synthesizable RTL code for the bridge can be automatically derived based on these parameters. We use industrial strength design drivers such as an MP3 decoder to test our automatically generated bridges for a variety of platforms and compare them to manually designed bridges on different quality metrics. Our experimental results show that performance of automatically generated bridges are within 5% of manual design for simple platforms but surpasses them for more complex platforms. The area and RTL code size is consistently better than manual design while giving 5 orders of improvement in development time.

Categories and Subject Descriptors J.6 [Computer-Aided Engineering]: Computer-Aided Engineering - Computer-aided design (CAD)

General Terms Performance, Design, Reliability, Experimentation, Verification.

Keywords HW-SW Co-design, Communication synthesis, Interface synthesis, Universal bridge, Transaction level model, Channel

1. Introduction

The rising complexity and increasing heterogeneity of modern systems has forced designers to explore new abstraction levels above RTL. To overcome this problem, designers are increasingly resorting to modeling such complex systems at higher levels of abstraction. As a result of this trend, transaction level model (TLM) is emerging as the new abstraction level above RTL. TLM simulation alone will not deliver the advantage of moving to a higher abstraction. It is crucial that we have synthesizable TLMs which can be brought down to RTL descriptions that can be input to conventional EDA tools such as logic synthesis tools.

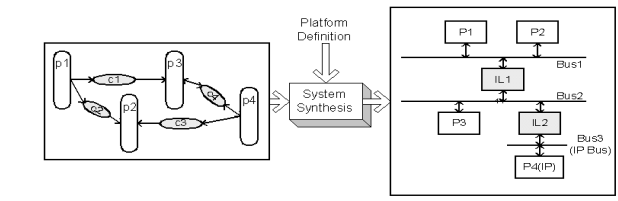


Figure 1. System synthesis from TLM

Figure 1 shows the system synthesis problem with a simple example. On the left, we have the input TLM consisting of several processes, each capturing computation in the application. Communication between these processes is captured using primitives called channels. The channels are simply a repository for *send* and *receive* functions that are called by the processes connected to those channels.

On the right of Figure 1 we have a model of the multi-core platform consisting of processing elements (P1, P2 etc.) that execute the respective processes (p1, p2 etc.) in the TLM. The abstract channel based communication of the TLM now needs to be implemented on the communication architecture of buses and bridges. Let's assume that P4 is an IP with a unique interface protocol not supported by Bus2. To enable communication between P3 and P4, we need some interface logic (IL2) that will translate the protocol between Bus2 and the custom IP bus Bus3. Also, we need communication between P1 and P3 due to channel *c3* between p1 and p3. Hence, another interface logic (IL1) is needed between Bus1 and Bus2 to enable this communication, irrespective of protocols of Bus1 and Bus2. In this paper we will show how both types of interfaces (IL1 and IL2) can be automatically generated using the concept of an interface logic called the universal bridge.

Interface synthesis has been studied by different groups. [1] presents abstraction levels of interfaces at various design stages. A bus generation algorithm was presented in [2]. Automatic wrapper generation was proposed, in [3]. Parameter based interface generation is proposed in [4]. These early approaches have mainly focused on translating one bus protocol to the other. But, these approaches have not been applied to TLMs. [5] resulted in the concept of networks on a Chips (NoC). However, NoC methods depend on standard router based communication architectures with limited opportunity for application specific communication optimizations.

2. TRANSACTION LEVEL MODEL

The TLM paradigm is based on the premise that communication between components may be simulated faster by abstracting away the unnecessary pin level details from the model. The proposed way of modeling communication in TLMs is through channels that essentially consist of functions that provide platform communica-

tion services. These functions build on events to synchronize the data transfer. Processes executing inside abstractly modeled components access these functions using interfaces provided by the channels. TLMs using channels are well established concept in SystemC and several different modeling styles have been proposed to provide faster simulation or higher modeling accuracy. From a synthesis perspective, we are interested only in the abstract communication services of sending and receiving data without any timing modeled inside the channel. Since all our channels are point-to-point, there is no need to model addressing and arbitration.

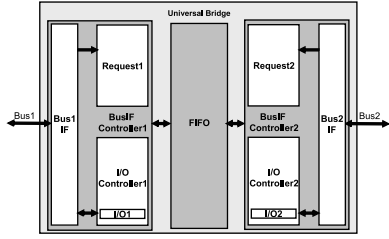


Figure 2. Universal bridge architecture

3. UNIVERSAL BRIDGE ARCHITECTURE

The universal bridge consists of a FIFO with dedicated controller for each bus protocol as shown in Figure 2. BusIF is a controller which reads the data on the bus and writes it to the registers in the request buffer (Request1, Request2) or I/O controller. The request buffer is a register file partitioned by routing path. Each route path is assigned on one register in the request buffer. The communicating PEs write "send/receive request" to dedicated register for the chosen route. The "send/receive request" has the message size. I/O controller has an I/O register for the message. It stores data from BusIF Controller before writing/reading to the FIFO. Also, it sends an interrupt to notify the communicating PE whether the universal bridge is available or not. The last sub-module is a FIFO. FIFO is also partitioned by route path and stores messages between two bus I/O controllers.

4. UNIVERSAL BRIDGE SYNTHESIS

The platform decisions provide the parameters to describe the structure of the universal bridge and message format. The universal bridge generator takes the TLM and platform definition along with protocol library to generate synthesizable RTL code for the universal bridge.

4.1 Universal bridge parameters

Universal bridge has parameters to define bus and the universal bridge. Bus parameter has two tables. First, SYNCTABLE defines the synchronization method between two components on the bus. SYNCENTRY in the table has master, slave, synchflag name, and synchronization method. Second, ADDRTABLE defines addressing and data transfer method. CHENTRY in the table also defines source, destination, transfer method and bus address for all channels in TLM. There are multiple SYNCENTRIES and CHENTRIES in the table.

The universal bridge parameter defines the internal structure of the universal bridge. It has bus interface and FIFO. The request buffers are defined in the bus interface. There are three parameters in it. First, *routes* tells which routes are going to use the request buffer. Second, *size* is the number of bytes of the request entry. Third, *type* specifies if the particular request entry for the given

route indicates a send or receive request. FIFO is also defined in similar way. It is divided into partitions and each partition has routes and size information in it. *Routes* enumerates all the routes in the platform. It specifies all components from the source to final destination.

4.2 Universal bridge generation procedure

The universal bridge generator has four stages. First, TLM analyzer reads the platform and bridge parameters from the XML data structure. Second, BusIF controller is generated. Third, the FIFO generator uses the parameters described in the TLM universal bridge and generates FIFO. At the end, the top module generator connects all sub modules and generates universal bridge.

Procedure 1 Request buffer generation procedure

```

1: Definition
2:  $R$  : Routes
3:  $B$  : Buses
4:  $IF_i$  : Bus interface controller for bus  $B_i$  in the universal bridge
5:  $RB_{ij}$  :  $j^{th}$  Request register for bus  $B_i$  in the universal bridge
6:
7: //generate Request Buffer
8: for all ( $RB_{ij} \in IF_i$ ) do
9:    $RB\_size = IF_i.RB_j\_size$ 
10:  add  $RB_{ij\_register}(RB\_size)$ 
11:  for all ( $R$  using  $RB_{ij}$ ) do
12:    if  $RB_{ij\_type} = SEND$  then
13:       $Route\_ID_{ij} = R\_send$ 
14:    else
15:       $Route\_ID_{ij} = R\_recv$ 
16:    end if
17:    add  $RB_{ij\_route\_list}(Route\_ID_{ij})$ 
18:  end for
19: end for
20:  $RB_i\_scheduler\_policy = IF_i.RB\_Sched\_policy$ 
21: generate  $IF_i.RB\_scheduler(RB_i\_scheduler\_policy,$ 
22:    $RB_i\_route\_ID\_list)$ 

```

4.2.1 Request buffer generation

Procedure 1 shows the step to generate a request buffer. The request buffer is an array of the request registers. In line 9 and line 10, it reads the physical size of the request register from the parameter and add corresponding register in the request buffer. In line 12 to 16, it reads all routes which is using the request register and assign route ID. One request register is used for "receive request", and the other request register is used for "send request" for the route given. In line 17, it creates route list for RB_{ij} since one request register can be shared by multiple routes. In line 20, the tool read the scheduling policy for the request buffer. In line 21, the request buffer scheduler is generated based on the scheduling policy and route ID list. The route ID list ($RB_i_route_ID_list$) includes all $RB_{ij_route_list}$ in the request buffer. The route ID list is used to generate request register reset signal after scheduling.

4.2.2 I/O controller generation

Generating I/O controller has two functions. First, it adds I/O register to store the data. Second, it has to add synchronization mechanism to the universal bridge. If synchronization method is interrupt, then it add logic to issue the interrupt. Otherwise, the tool adds the synchronization flag for polling and add extra logic to set the flag.

4.2.3 FIFO and Top module generation

To generate the memory, the FIFO generator just reads the partition size and assigns top and bottom addresses for each partition. Top

| | Lines of Codes | | | Number of Slices | | | Max clock speed (Mhz) | | | Throughput(Bytes/us) | | |
|-------------|----------------|-------|--------|------------------|--------|--------|-----------------------|--------|--------|----------------------|--------|-------|
| | SW+1 | SW+2 | SW+3 | SW+1 | SW+2 | SW+3 | SW+1 | SW+2 | SW+3 | SW+1 | SW+2 | SW+3 |
| Manual | 2878 | 3305 | 4003 | 1300 | 1616 | 2400 | 38 | 36 | 34 | 110.37 | 105.44 | 87.41 |
| Automatic | 2912 | 3068 | 3404 | 1127 | 1444 | 1790 | 34 | 33 | 32 | 105.26 | 102.69 | 89.71 |
| Improvement | -1.18% | 7.17% | 14.96% | 13.31% | 10.64% | 25.42% | -10.53% | -8.33% | -5.88% | -4.63% | -2.61% | 2.63% |

Figure 3. Design quality

Table 1. Development time of universal bridge

| Development time | SW+1 | SW+2 | SW+4 |
|------------------|------------|------------|------------|
| Manual | 40(hour) | 52(hour) | 60(hour) |
| Automatic | 0.156(sec) | 0.187(sec) | 0.234(sec) |

module generation is composed of two loops. During the first loop, for every bus, the tool runs the request buffer generator and I/O controller generator. In the second loop, it runs FIFO generator for every FIFO in the universal bridge. After running two loops, the tool connects all internal signals to finalize the universal bridge generation.

5. EXPERIMENTAL RESULT

5.1 Design Quality

In this section, we report the universal bridge implementation result for mp3 player. We use $SW + 1$ to denote platform with Microblaze, 1 bridge and 1 HW DCT. $SW + 2$ denotes platform with Microblaze, 1 bridge and 2 HW DCTs. Finally, $SW + 4$ denotes platform with Microblaze, 1 bridge, 2 HW DCTs and 2 HW IMDCTs. In Figure 3, first two rows show design quality for manual and automatic design. The last row shows the improvement the automatic design over the manual design in percentage. First group shows the lines of codes. It increases rapidly in manual design as design complexity increases, but it increases gradually in automatic design. As a result, in "SW+4", the automatic design generates 15% smaller code. Second group shows the number of slices used in FPGA. Automatic design uses less slices than the manual design for all test cases. The automatic design uses 13% to 25% less slices. Third group shows the maximum clock speed for the universal bridge. The maximum frequency of the automatic design rarely changes regardless of design complexity. While the maximum frequency of manual design decreases as design complexity goes up. As a result, Manual design is 10% faster in SW+1, but there is only 5.88% difference in SW+4. These results tell that automatic design is suitable for more complex design. The last group shows the performance in byte/us. The throughput (performance) of the universal bridge is obtained as follows:

$$\text{Throughput} = \frac{\text{The total size of messages}}{\text{The total time consumed in UB}} \text{ (Bytes/us)} \quad (1)$$

we used maximum clock speed to obtain the performance using the Equation 1. There are less than 5% differences. The result also indicates that the performance of automatic design surpasses the performance of manual design as number of PE increases in SW+4.

5.2 Development Time

Table 1 shows our development time for manual design and automatic design. Our approach has great advantage especially in design time by removing the time consuming manual RTL coding. Note that the time unit for automatic RTL development time is in seconds since the input file for the automatic design is also generated automatically. The time unit for the manual design is in hour. Synthesizable RTL code generation using the automatic tool took just a second while manual design took 40 to 60 person-hours.

Therefore by generating synthesizable RTL from TL model automatically, we can design more than 10,000 X faster than manual design.

6. CONCLUSION

In this paper we presented synthesis of universal bridge from TLM. We synthesized it for various MP3 player architectures and run it on FPGA board. The result shows that automatically generated universal bridge performs as good as manual design while using 15% to 25% less slices. Our result also shows the other benefits of automatic interface generation. Since RTL verification is the most time consuming and error prone step amongst all design stages, interface synthesis from TLM gives us huge benefits in development time and reliability. Also, the automatic synthesis of the universal bridge makes it possible to attach any types PE to the system in no time that the system designer can explore more design space. In the future, we would like to enhance our TL synthesis methods to support even more complex design drivers such as Networks on Chip.

References

- [1] G. Borriello, L. Lavagno, and R. B. Ortega, "Interface synthesis: a vertical slice from digital logic to software components", *In Proc. of ICCAD, 1998*
- [2] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces.", *Proc. European Design and Test Conference, 1997* Pages 395 - 399
- [3] Passerone, R., Rowson, J., Sangiovanni-Vincentelli, A., "Automatic Synthesis of Interfaces between Incompatible Protocols", *Proc. 35th Design Automation Conf. (DAC 98)*, ACM Press, San Francisco, CA, 1998.
- [4] Yin-Tsung Hwang; Sung-Chun Lin, "Automatic protocol translation and template based interface synthesis for IP reuse in SoC", *The 2004 IEEE Asia-Pacific Conference on Circuits and Systems, 2004*. Proceedings. pages 565-568
- [5] L. Benini, G. De Micheli, "Networks on Chips: A New Soc Paradigm", *IEEE Computer*, vol.35, January 2002 pages 70-78