TLM 2.0 simple sockets synthesis to RTL

TLM 2.0 Simple Sockets Synthesis to RTL

Nadereh Hatami[2], Amirali Ghofrani[1], Paolo Prinetto[2], Zainalabedin Navabi[1]

[1]CAD Research Group, ECE Department, University of Tehran, Tehran 14399, Iran

{*ghofrani, navabi*}*@cad.ut.ac.ir*,

[2]Dipartimento di Automatica e Informatica, Politecnico di Torino, I-10129 Torino, Italy

{*nadereh.hatami, paolo.prinetto*}*@polito.it*

Abstract

Convenience sockets are a family of derived sockets provided in utilities namespace of TLM 2.0 library which add additional functionalities to TLM 2.0 sockets. As one of the goals of high level modeling is to part communication from computation, synthesizing communication mechanisms including sockets can be a primary step to synthesize each TLM 2.0 design on RTL. Synthesizing sockets on RTL can provide the designer with the big picture of module's functionality and communication requirements. In this paper, algorithms are proposed to map TLM 2.0 simple sockets down to RTL focusing on TLM 2.0 blocking and non-blocking transport interfaces. The algorithms get TLM 2.0 sockets as an input and generate an intermediate description of sockets in terms of ports. After that, RTL descriptions of the ports are generated using the standard generic payload packet as transaction type. The automation caused by synthesis algorithms in this paper can reduce the simulation speed and the designer's effort.

1. Introduction

Transaction Level Modeling (TLM) approaches have been proposed to describe Systems-On-Chips (SoCs) at a higher abstraction level than RTL. [2, 3, 4] describe TLM as a high level transaction language based on SystemC. TLM simulates faster than RTL, even for complex systems, and it allows embedded software validation and integration testing to be done earlier in the design cycle that is before RTL is complete. TLM modules consist of two distinct parts: TLM core modules which provide us with the functionality of the individual components of the SoC and TLM interfaces which are the means of communication between IP cores. Interface methods are implemented in the modules and can be called via TLM 2.0 sockets to transfer data packets from one module to the other. As the design proceeds in the design flow, the abstract model should be replaced by a more detailed architecture of what is really going to be done. This detailed architecture can then be synthesized to RTL.

Some works are done on TLM synthesis to extract RTL model of the design from the transaction level description. [5] is a high-level solution that integrates electronic system level designs with block-level implementation. [6] also uses a library of synthesizable TLM protocols to synthesize transaction level descriptions into SystemC RTL code. Although these tools facilitate TLM synthesis, they are all designed to work with TLM 1.0 descriptions. Currently, no available tool will synthesize TLM 2.0 architectures down to RTL. This paper tries to prepare a path to synthesis of high level TLM 2.0 architectures down to RTL by introducing algorithms to synthesize TLM 2.0 simple sockets to RTL. Sockets are introduced by TLM 2.0 standard to establish connection between two or more modules. They provide two different paths from initiator to target and vice versa to facilitate transaction in both forward and backward path. This way, synthesizing sockets to RTL description can be the first step to synthesize TLM 2.0 architectures to lower levels of abstraction. In this paper, we focus our attention on blocking and non-blocking transport interfaces and leave other interfaces for the future works.

The next section contains an introduction to TLM 2.0 convenience sockets with focus on simple sockets. Section 3 describes our proposed scheme to synthesize TLM 2.0

978-1-4244-4321-5/09/$25.00 ©2009 IEEE

simple sockets to RTL. Section 4 explains the experimental results in terms of the overhead of the synthesized description of TLM 2.0 simple sockets. Section 5 is the conclusion.

2. Introduction to TLM 2.0 sockets

A *socket* is a high level construct introduced in the TLM 2.0 standard which combines a port with an export to provide designers the facility of sending packet in two ways. An *initiator socket* offers a port for the forward path and an export for the backward path, whilst a *target socket* introduces an export for the forward path and a port for the backward path.

The initiator and target sockets group the TLM 2.0 interfaces including transport blocking and non-blocking transport interfaces for both the forward and backward paths together into a single object. They also provide methods to bind port and export of both the forward and backward paths in a single function call. They also include a bus width parameter as a template input type that may be used to interpret the transaction. The *tlm_initiator_socket* and *tlm_target_socket* classes belong to the interoperability layer the TLM 2.0 standard. Moreover, several derived socket classes are provided in the utilities namespace, known as *convenience sockets* [1].

Convenience sockets are designed to improve the socketÑ communication ability by providing several additional functionalities as well as the ones offered by normal socket classes. There are five distinct convenience sockets with different capacities in providing connection between two modules. The *simple sockets* are members of convenience socket family which are derived from the *tlm_initiator_socket* and *tlm_target_socket* classes and can directly bind to them.

Simple sockets offer methods to register callbacks for incoming interface method calls. Registering a specific method, the module is responsible for implementing the registered function. In this case, the socket can use combined interfaces to communicate with different modules instead of binding to a single interface. The combined forward and backward transport interfaces group the core TLM 2.0 interfaces for use by the initiator and target sockets. Note that the combined interfaces include the transport, DMI and debug transport interfaces, while the TLM 1.0 core interfaces are not included. The forward interface provides method calls on the forward path from initiator socket to target socket, and the backwards interface on the backward path from target to initiator socket. Both the blocking transport interface and the debug transport interface use only the forward path for communication [1].

In addition to register call back facility, simple target sockets are also able to convert incoming transport calls to b_transport into nb_transport_fw calls and vice versa [1].

In addition to sockets, communication between two modules is also dependant to the transaction type which is transferred through sockets. TLM 2.0 introduces a standard transaction type to be used by components.

The generic payload is introduced to improve the interoperability of memory-mapped bus models and facilitate the IP reuse of TLM 2.0 IPs. It provides a general-purpose payload to guarantee the interoperability among TLM 2.0 components when creating abstract models of memory-mapped buses. It also provides the extension mechanism which can be added to generic payload object when generic payload attributes are not adequate to model the full functionality of the architecture. Moreover, it is capable of creating detailed models of specific bus protocols, while reducing the implementation cost and increasing simulation speed of the whole design. Using the standard generic payload, IPs from different vendors are capable of communicating together without the need of unifying the transactions types.

To synthesize TLM 2.0 sockets to RTL, it is also important to provide a method to synthesize generic payload object as a standard transaction type in TLM 2.0 architectures.

3. Socket synthesis approach

To synthesize sockets, the first step is to understand the inner structure of them. As sockets are derived from *tlm_initiator_socket* and *tlm_target_socket* classes which are written in SystemC, it would be desirable to map TLM 2.0 sockets to communication types in SystemC µsc_port and sc_export- and then find a direct path from this level down to RTL. In this case, several details which are not considered in high level TLM 2.0 description can be implemented.

Moreover, socket synthesis is dependant to the type of the transaction. Consequently, we first introduce methods to synthesize generic payload object to RTL. After that, using the RTL description of the generic payload object, algorithms can be proposed to synthesize the initiators and target sockets down to RTL.

3.1. TLM generic payload

From the definition, standard payload type of TLM 2.0 has several attributes. Table 1 shows these attributes with their type and their default value.

Table 1. TLM generic payload attributes

Attribute	Type	Default Value
Command	tlm_command	TLM_IGNORE_COMMAND
Address	sc_dt::uint64	0
Data pointer	unsigned char*	0
Data length	unsigned int	0
Byte enable pointer	unsigned char*	0
Byte enable length	unsigned int	0
Streaming width	unsigned int	0
DMI allowed	bool	false
Response status	tlm_response_status	TLM_INCOMPLETE_RESPONSE
Extension pointers		0

To synthesize sockets, we have to first figure out what is going to be transmitted through them. So, having a picture of generic payload in RT level is useful to propose algorithms for socket synthesis.

Command is a value of type tlm_command. The definition of tlm_command in TLM 2.0 library is shown in Figure 1.

```
enum tlm_command {
   TLM_READ_COMMAND,
   TLM_WRITE_COMMAND,
   TLM_IGNORE_COMMAND
};
```

Figure 1 tlm_command definition

As *tlm_command* is an enumeration type with three values, it can be considered as a 2 bit *std_logic_vector* value in VHDL.

Address is a value of type uint64 and so, in can be considered as *std_logic_vector* type of length 64.

Data pointer is a pointer to data array. As pointers are not supported by VHDL, we can directly use the data array to be transmitted by sockets. The length of the data array can be a variable determined by a generic value. This generic value can be obtained by *Data length* value. Data length is an integer specifying the length of the data, so, it can be used to define the data array:

Data_array: std_logic_vector (data_length-1 downto 0);

Byte enable pointer is a pointer to byte enable array. The byte enable array is applied repeatedly to the data array. The elements in the byte enable array shall be interpreted as follows. A value of 0 shall indicate that that corresponding byte is disabled, and a value of 0xff shall indicate that the corresponding byte is enabled. The meaning of all other values shall be undefined. The value 0xff has been chosen so that the byte enable array can be used directly as a mask. Byte enables may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat, or to place words in selected byte lanes of a bus [1]. As pointers are not supported by VHDL, we can directly use the byte enable array to be transmitted by sockets. The length of the array can be variable determined by a generic value. This generic value can be obtained by *byte enable length* value. Byte enable length is an integer specifying the length of the byte enable array, so, it can be used to define the byte enable array:

BE_array: std_logic_vector (BE_length-1 downto 0);

The *Streaming width* attribute determines the width of the stream, that is, the number of bytes transferred on each beat. Streaming affects the local address associated with each byte in the data array. Streaming keeps the organization of the data array unaffected. This value can be represented by an integer in VHDL.

The *dmi_allowed* attribute determines whether direct memory interface can be used or not. This can be implemented by a single bit of type *std_logic* in VHDL.

Response Status is an enumeration type with 7 values shown in Figure 2. So, it can be described by 3 bits of type *std_logic_vector* in VHDL.

```
enum tlm_response_status {
   TLM_OK_RESPONSE = 1,
   TLM_INCOMPLETE_RESPONSE = 0,
   TLM_GENERIC_ERROR_RESPONSE = -1,
   TLM_ADDRESS_ERROR_RESPONSE = -2,
   TLM_COMMAND_ERROR_RESPONSE = -3,
   TLM_BURST_ERROR_RESPONSE = -4,
   TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
};
```

Figure 2 tlm_response_status definition

Extension pointer(s) are pointers to ignorable extensions. To fit all the requirements, *tlm_generic_payload* provides the facility to append extensions of any type to generic payload object as ignorable extensions. As this will complicate the synthesis process, we assume to have no extensions in this paper.

Put everything together, the summary of generic payload synthesis is shown in Table 2.

Table 2. Summary of generic payload synthesis

Attribute	Required bits
Command	Std_logic_vector(1 downto 0)
Address	Std_logic_vector(63 downto 0)

Data pointer	Std_logic_vector(Data length downto 0)
Data length	-
Byte enable pointer	Std_logic_vector(Byte enable length downto 0)
Byte enable length	-
Streaming width	Int
DMI allowed	Std_logic
Response status	Std_logic_vector(2 downto 0)
Extension pointers	-

3.2. Simple sockets

As mentioned previously, simple sockets provide the designer with the possibility of registering call backs. In this case, the socket can use the grouped interfaces of TLM 2.0 as communication interface by registering them in the initiator. This way, the socket should implements incoming interface method calls only by registering callbacks, not by being bound hierarchically to another socket.

```
1  int k = 0;
2  for (int i=0; i<LUT.numberOfRows; i++)
3  {
4     RLUT[k]. type = port;
5     if (b_transport){
6        RLUT[k].interface = tlm_blocking_transport_if;
7        k++;
8     }
9     if (nb_transport_fw){
10       RLUT[k].interface = tlm_nonblocking_transport_fw_if;
11       k++;
12    }
13    if (nb_transport_bw){
14       RLUT[k].interface =
tlm_nonblocking_transport_bw_if;
15       k++;
16    }
17    if (DMI_fw){
18          RLUT[k].interface =
tlm_direct_memory_fw_if;
19          k++;
20    }
21    if (DMI_bw){
22          RLUT[k].interface = tlm_direct_memory_bw_if;
23          k++;
```

Figure 3 socket to port mapping algorithm

To synthesize simple sockets, the first step is to map them to SystemC ports to provide the synthesis algorithm with the details which are not considered in TLM 2.0 high level description. Specifying the interfaces each socket uses is part of the job. After this step, we propose an RTL description of the translated sockets to synthesize the ports with the blocking and non-blocking transport interfaces. To propose the algorithm, we take advantage of a look up table as data structure which will be filled during first compiler pass. This table has the following attributes:

a. **Char* Socket:** The name of the declared socket.
b. **Bool Type:** È0Š if the socket is an initiator socket and È1Š if it is a target socket.
c. **Bool b_transport:** true if the blocking transport interface is registered for the socket, else false.
d. **Bool nb_transport:** true if the non-blocking transport interface is registered for the socket, else false.
e. **Bool DMI:** true if the direct memory interface is registered for the socket, else false.

This data structure is called *LUT*. The results will be stored in another table called RLUT (Result LUT) with the following attributes:

a. **Char* Port:** the name of the ports or exports of the mapped architecture
b. **Bool Type:** È0Š for port and È1Š for export
c. **Char* Interface:** the portÑ related interface

Using these data structures, Figure 3 shows the mapping algorithm.

Each row in RLUT stores the name of a port derived from the socket, its type (port or export) and the interface bind to it. After this primary step which results in a filled RLUT table, we can propose algorithms to synthesize TLM 2.0 sockets to synthesizable input and output ports.

We should mention that as the mapping is done by processing the registered interfaces, the backward path will automatically be considered in the synthesis process and there is no need to check for backward paths after the algorithm is executed.

The next step is to synthesize the obtained ports to RTL. Figure 4 shows the socket synthesis process graphically.

```
1  entity NBPort is
2     port (
3        inPacket  : in  dataRecord ;
4        inResponse : in  NBResponse ;
5        outPacket: out dataRecord ;
6        outResponse   : out NBResponse
7     ) ;
8  end entity ;

9   architecture IMP of NBPort is
10   signal  PFifo   : dataBuffer(BufferLen downto 0) ;
11   signal  PHead : integer range 0 to BufferLen := 0 ;
12   begin
13      outResponse   <= inResponse ;
14      outPacket<= PFifo(PHead) ;
15      process (inResponse, inPacket, PFifo, PHead)
16      variable  check  : boolean ;
17      begin
18         if (not (PFifo(PHead - 1) = inPacket)) then
19            if (Phead < BufferLen) then
20               PFifo(PHead)  <= inPacket ;
21               PHead        <= PHead + 1 ;
22            end if ;
23         else
24            check  := false ;
25            if (inResponse.response = processFinished) then
26               for i in 0 to BufferLen - 1 loop
27                  if (inResponse.ID = PFifo(i).ID) then
28                     check  := true ;
29                  end if ;
30                  if (check) then
31                     PFifo(i)   <= PFifo(i + 1) ;
32                  end if ;
33               end loop ;
34               if (check) then
35                  PHead <= PHead - 1 ;
36               end if ;
37            end if ;
38         end if ;
39      end process ;
40   end architecture IMP ;
```

Figure 6 Implementation of the corresponding RTL non-blocking interface

4. Experimental results

We synthesized our proposed synthesis model of the TLM 2.0 sockets on Cyclone EP1C12Q240C8 FPGA. The synthesis results show the overhead of moving from high abstraction level down to RTL. Table 2 shows the amount of logic cells which must be added to the RTL design to implement the blocking and non-blocking transport interfaces.

Table 3 Hardware overhead of the synthesized sockets

Interface Type	Total Logic Elements
TLM Blocking Transport Interface	647
TLM Non-blocking Transport Interface	3827

The fifo used in the non-blocking controller was assumed to be of size 4. Bigger fifos will result in more overhead. Also note that part of such an overhead is due to the numerous I/Os resulting from the nature of the design and not to the transition from higher to lower abstraction level.

5. Summary and future works

In this paper, we have proposed algorithms to synthesize TLM 2.0 simple sockets with blocking and non-blocking transport interfaces to RTL. We start our work by implementing an algorithm to map TLM 2.0 simple initiator and target sockets to an intermediate structure with port and exports supported by SystemC. Then we continue the work by synthesizing the mapped designs to RTL using VHDL.

We are now working on other interfaces such as direct memory interface and we are going to provide the same procedure to synthesize these ports to RTL. Extending the method to other kinds of convenience sockets is the next step for future works. We also planned to extend the method to other parts of a TLM design and finally propose methods to synthesize TLM 2.0 architectures.

6. References

[1] M. Montoreano. ÈTransaction Level Modeling using OSCI TLM 2.0Š, Technical report, Open SystemC Initiative, 31 May 2007.

[2] F. Ghenassia, **Transaction Level Modeling with SystemC**. Springer, Dordrecht, Netherlands, 2005.

[3] J. Cornet, F. Maraninchi, L. M. Contoz, ÈA Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-ChipŠ, Design, DATE 2008.

[4] http://www.forteds.com/products/tlmsynthesis.asp, October 26, 2008.

[5] http://utcadlab.net/Projects.aspx, October 26, 2008.

[6] Z. Navabi, **VHDL: Analysis and Modeling of Digital Systems**, Second Edition, McGraw-Hill, 1998.

[7] SystemC 2.0.1 Language Reference Manual Revision 1.0, Open SystemC Initiative, San Jose, California, 2003.

[8] http://www.systemC.org, November 16, 2008.

[9] M. Montoreano, ÈTransaction Level Modeling using OSCI TLM 2.0Š, Synopsys, Inc. May 31, 2007.

[10] A. Rose, S. Swan, J. Pierce, and J. Fernandez, ÈTransaction Level Modeling in SystemCŠ, OSCI TLM Working Group, 2004.

[11] D. C. Black, J. Donovan, B. Bunton, **SystemC from the Ground Up**, Kluwer Academic Publishers, 2004.

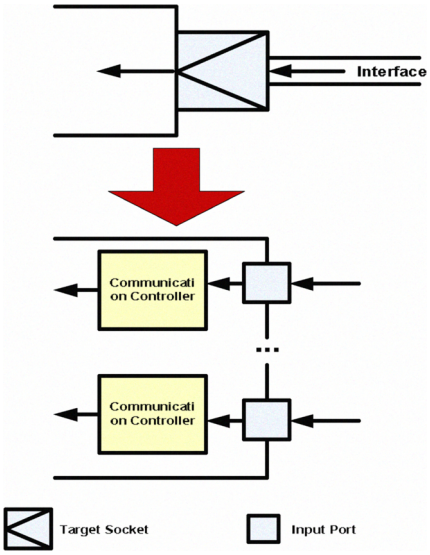

Figure 4 TLM 2.0 Socket Synthesis process

As it can be seen in Figure 4, the target socket with combined communication interface is synthesized to several input ports. The input ports are connected directly to a communication controller. The functionality of the communication controller is to define interface rules to the RTL module. Each input signal should be processed by communication controller. The system inputs are connected to the output of the communication controller to provide the whole architecture with the behavior of the corresponding socket. In this paper, we focus our attention under blocking and non-blocking interfaces and propose communication controllers to model the behavior of these two kinds of interfaces in RT level.

To do the job, we assume a socket as a component with inputs and outputs. The idea is to embed the interface in the input or output port structure to manage the communication protocols. In case of simple initiator sockets, the socket is a component with inputs from the module itself and outputs to a target socket. In other case, the process is reverse. The TLM target socket is a component with inputs from an initiator socket and outputs to connect to module signals. In VHDL implementation, we pack the input and output signals to an input and output records and write a controller in the module side of the socket to apply communication rules to the component. We categorize the synthesized sockets with respect to the interface to which they are bound.

In both cases, we have assumed that both the packet and the response will remain on the port input until a new packet arrives.

```
1  entity BPort is
2     port (
3        inPacket  : in     dataRecord ;
4        inResponse : in     BResponse ;
5        outPacket: bufferdataRecord ;
6        outResponse    : bufferBResponse
7     ) ;
8  end entity ;

9     architecture IMP of BPort is
10    begin
11       outResponse    <= inResponse ;
12       process (inResponse, inPacket)
13       begin
14          if (inResponse = BResponse(processFinished)
             and NOT (inPacket = outPacket)) then
15             outPacket<= inPacket ;
16          else
17             outPacket<= outPacket ;
18          end if ;
19       end process ;
20    end architecture IMP ;
```

Figure 5 implementation of the corresponding RTL blocking interface

3.2.1. TLM blocking transport interface

For the blocking transport interface, no new packet can be accepted before the *processFinish* response on the previously received one. Figure 5 shows the complete process.

3.2.2. TLM non-blocking transport interface

The synthesized ports with TLM non-blocking transport interface obtained from the TLM 2.0 simple socket mapping algorithm of 3.2 is demonstrated in Figure 6.

As shown in Figure 6, for the non-blocking transport interface, there a buffer of predefined size is used to store the input packets.. Each time a new packet arrives, it will be stored in the head of this buffer. Each time a response is received from the contributing module, from *inResponse* port, this response will be transmitted to the *outResponse* port unchangeably. This response is monitored to check whether it is a *processFinished* response, or not. In this case, the corresponding packet in the buffer should be removed, as its processing is done. To do so, the ID of the *inResponse* will be checked with the ID of the packets stored in the buffer, and in case of a match, the fifo will be shifted afterwards.