An advanced Method for Synthesizing TLM2-based Interfaces

Nadereh Hatami, and Zainalabedin Navabi
Electrical and Computer Engineering School, University of Tehran
{nhatami, Navabi}@cad.ut.ac.ir

Abstract

Transaction Level Modeling (TLM) describes system on chips at a high abstraction level in which simulation is faster than the traditional design flow. TLM2 serves the designer with three distinct coding styles among which the untimed style has the best simulation speed with the lowest accuracy. In this paper, we are presenting a method of synthesizing TLM untimed descriptions directly to synthesizable behavioral description from which they can be converted to RTL. This direct mapping skips the intermediate synthesis steps and speed up the design process. We restrict our proposed method to TLM interfaces and specifically work on DMA devices as a common interface in communications sharing a common bus between a processor and its I/O devices.

1. Introduction

Transaction Level Modeling (TLM) approaches are proposed to describe Systems-On-Chips 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, allows embedded software validation and integration testing to be done earlier in the design. A possible approach to Transaction-Level Modeling is to create a single model to fit all purposes: embedded software development, performance evaluation, etc. In such a model, transactions have to correspond directly to the operations performed on interconnect, whose size is the bus width. Timings details as well as arbitration are also present. However, this approach does not cope well with conflicting requirements from the various TLM activities [4]. The alternative solution is to take advantage of blocking interfaces which is independent of the interconnect bus width and timing details. It simulates faster, with the expense of loss of details.

As the design proceeds in the design flow, this abstract model should be replaced with 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.

In this paper, we are going to focus on interfaces as a means of communication in TLM designs, and among all interfaces we focus our attention on DMA as the architecture that allows data to be sent directly from an attached device (such as a disk drive) to the memory. The idea lies behind the fact that with recognizing familiar hardware structures in the design and mapping them to the synthesizable equivalence structure, we would speed up the synthesis process of the design.

In the next section, after an overview of TL modeling in SystemC with respect to TLM2 untimed coding styles, we describe our methodology for TLM synthesis in Section 3. We survey the TLM2 untimed model design followed by the SystemC synthesizable equivalence and introduce a way to test the synthesizable design in the high abstraction level TLM2 structure. In Section 4, we explain the experimental results by considering the coding styles of high level and synthesizable design. Section 5 is the conclusion.

2. TLM2 untimed modeling

TLM2 consists of a set of core interfaces, analysis ports, initiator and target sockets, and the generic payload. The core interfaces support untimed, loosely-timed and approximately-timed coding styles. Each coding style can support a range of abstraction across functionality, timing and communication. The generic payload supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols whilst maximizing interoperability [1].

The untimed coding style is the most abstract type with the least details using blocking interfaces for its implementation. Coming down in the abstraction layers, we gain more details and accuracy with the expense of complexity and time. A possible design flow starts from the untimed coding style, describing the functionality as what is seen by the software designer. The loosely timed model will then be extracted from the untimed model followed by the approximately timed model. Then a cycle accurate model of the design will be extracted in an HDL e.g., SystemC or VHDL which can directly synthesize to RTL. This process is shown in figure 1.

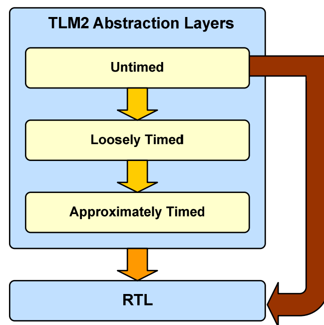

Figure 1. Abstraction Level from TLM to RTL

This mapping is a time consuming process. The main idea of this paper is to reduce this time by recognizing IP cores in the untimed model and replacing them directly with the synthesizable model of the recognized design. This direct way from untimed model to RTL is also demonstrated in Figure 1.

3. Methodology

A TLM design can consist of initiators, targets, and interfaces which connect initiators and targets together. In this paper, we focus our attention to the interfaces. The most common interfaces are routers, arbiters and DMA devices. The latter is the one that we are going to discuss in the rest of this paper.

A DMA structure we have proposed in TLM2 in shown in Figure 2. As shown, the memory has two target ports to communicate with the processor and DMA. The DMA has also a target port to get commands from CPU and one to communicate with the I/O device to get the data. It also has three initiator ports: one to initiate transaction with memory, one to report its status to CPU and one to report the status to the I/O device.

The mechanism of DMA in high level description is as follows;

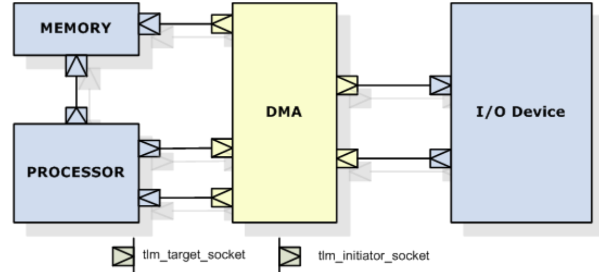

Figure 2. DMA high level structure

The DMA uses *tlm_blocking_transport_if* interface to communicate with the I/O device, the memory and the processor. To start a transaction, the I/O device sends a generic payload packet to the DMA calling the *b_transport* method from *tlm_blocking_transport_if* interface. The DMA extracts the required data for the memory and reconstructs a generic payload packet and send it to memory by calling the *b_transport* method from the same interface.

The TLM_COMMAND field of generic payload packet specifies the type of DMA action described by the I/O device. TLM_READ_COMMAND specifies read action and TLM_WRITE_COMMAND is "write to memory" operation. The data to be written is appended to the generic payload as the *tlm_extension* object. After each write action, the processor will be notified.

3.1. Modeling Approach

To describe a complete hardware system, we use a NoC network with 9 switches in MESH structure as an I/O platform.

We start modeling the entire network in untimed TLM coding style. At this level of abstraction, the functionality of the system without considering the details will be modeled. As demonstrated in figure 2, Each processing element consists of a DMA system, a memory and a processor. The processor is defined at the high abstraction level as a 16 instruction processor. The instruction set is shown in Table 1.

3.2. Synthesis Approach

To synthesize the interface module, some decisions should be made before the process starts. It should be decided whether to use a bus arbitration mechanism to grant the bus to the requesting device or use a dual port memory without any arbitration mechanism. This decision can automatically be made by a synthesis tool or manually by the designer. Here, we assume a bus arbiter for the arbitration process. The bus width of the design is previously mentioned in the TLM_SOCKET

object and can be used to design the proper hardware component.

Table 1 Designed Processor's Instruction Set

Instruction	Action	
`NOP`	No Operation	
`SZF`	Set Zero Flag	
`JNZ address`	Jump if Not Zero	
`MOV op1, op2`	Move two operands	
`LDI immediate`	Load immediate	
`STR address`	Store to address	
`INP address`	Input	
`OUP address`	Output	
`ADD op1, op2`	Add two operands	Op1 = op1 + p2
`CMP op1, op2`	Compare	
`MUL op1, op2`	Multiply	Op1 = op1 * p2
`SUB op1, op2`	Subtract	Op1 = op1 – p2
`NOT op1`	Logical not	Op1 = ⌐ op1
`AND op1, op2`	Logical and	Op1 = op1 Λ p2
`SND address`	Send [address]	
`HLT`	halt	

Figure 3 is the synthesizable hardware equivalent of Figure 2 designed in SystemC. It is a common DMA device used in some hardware structures adopted from [7]. Obviously, other implementations are also possible.

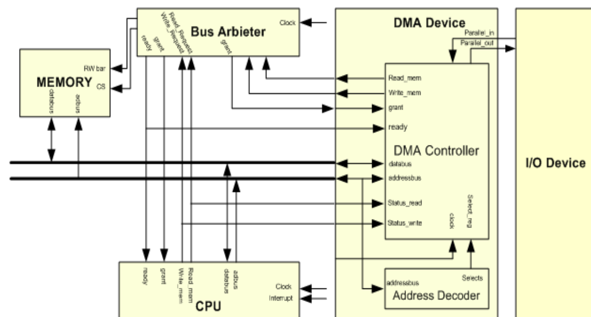

Figure 3. RTL version of DMA device

The above structure can synthesize to RTL using available synthesis tools. The synthesized DMA is shown in Figure 4.

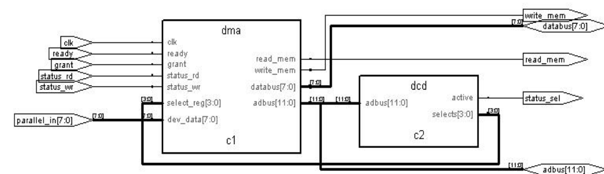

Figure 4. Synthesized DMA device

3.3. DMA for high level design

This DMA design is generalized to variable bus widths in order to be used in various designs with different specifications. To do so, we take advantage of bus width attribute defined in TLM2 socket description and inherited by Simple Socket structures which are shown in Figure 5.

```
template <typename MODULE,
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm::tlm_base_protocol_types>
class simple_initiator_socket :
  public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
{…}
template <typename MODULE,
  unsigned int BUSWIDTH = 32,
  typename TYPES = tlm::tlm_base_protocol_types>
class simple_target_socket :
  public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{…}
```

Figure 5. TLM2 initiator and target socket description

This bus width specifies the width of the databus used by the memory, processor and the DMA device. It can be read in the design to make the DMA compatible with various bus width declarations.

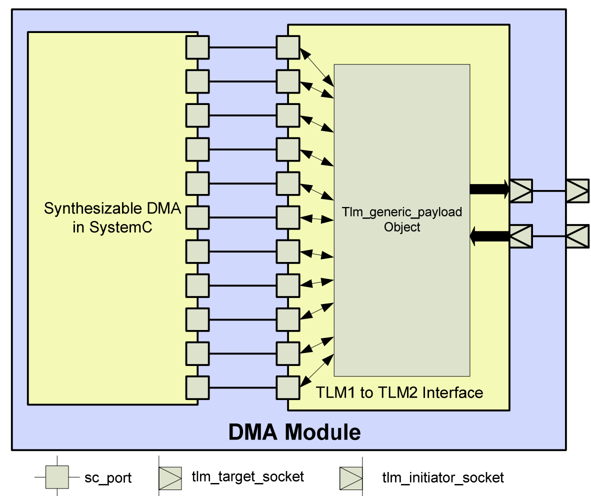

Figure 6. Using TLM1 to TLM2 interface to use synthesizable modules in TLM2 designs

TLM2 provides TLM1 interfaces to make it compatible with the previous version. We take advantage of this interface to test our synthesizable model in SystemC with other high level parts of the design. This replacement can guarantee the correct functionality of the synthesized interface. Figure 6 shows this process.

```
tlm_utils::simple_target_socket<dma> in_port;
tlm_utils::simple_initiator_socket<dma> out_port;
tlm_utils::simple_initiator_socket<dma>
dma_mem_init_port;
tlm_utils::simple_initiator_socket<dma>
dma_cpu_init_port;
```

Figure 7. DMA design in TLM2 port declaration part

As shown in figure 6, the interface provides a series of ports on one side and a set of TLM target and initiator sockets on the other side. The synthesizable DMA outputs are connected to the ports. With each event on DMA ports, the interface fills the *tlm_generic_payload* object attributes with the

transferred data from the DMA and sends the generic payload object through its initiator socket. On the other hand, when the DMA receives a new packet, it extracts attributes from the generic payload packet and sends them to the synthesizable DMA object through its ports.

4. Experimental Results

The purpose of this paper is to find a straight forward path to use TLM models without having to be concerned about the way they synthesize. Figure 7 shows the port declaration part of the DMA device in TLM2.

This declaration is equivalent to what we have described in the synthesizable version of our design shown in Figure 8.

```
sc_in<sc_logic>   clk;
sc_out<sc_logic>  read_mem;
sc_out<sc_logic>  write_mem;
sc_inout_rv<8>    databus;
sc_inout_rv<12>   adbus;
sc_in<sc_logic>   ready;
sc_in<sc_logic>   grant;
sc_in<sc_logic>   status_rd;
sc_in<sc_logic>   status_wr;
sc_out<sc_logic>  status_sel;
sc_in_rv<8>       parallel_in;
sc_out_rv<8>      parallel_out;
```

Figure 8. DMA design in SystemC declaration part

TLM2 interfaces have the responsibility of transferring and receiving data packets in the original design. These interfaces are substituted with *sc_signal* interface in SystemC with more overhead in code complexity (lines of code) and simulation time.

```
void     b_transport(     tlm::tlm_generic_payload&     trans,
sc_core::sc_time& t){
 trans.get_extension(packet_in);
 if (!packet_in){//packet received from processor
 {set address, read command and data length of the packet}
 dma_mem_init_port -> b_transport(generic_packet_out, delay);
 if (generic_packet_out. .is_response_ok()){
 //inject received packet from memory to network
 generic_packet_out.get_extension(instr);
 {setting other packet values to send it to the switch}
 sw_packet_in.set_extension(noc_packet::ID, packet);
 out_port->b_transport(sw_packet_in, delay);}}
 else { //packet received from I/O interface
 if (trans.is_write()){
 generic_packet_out.set_extension(packet_in->pdata);
 {also setting write, address, and length attributes}
 dma_mem_init_port -> b_transport(generic_packet_out, delay);
 if (generic_packet_out.is_response_ok()){
 {set    address,write    and    data    length    attribute    of
cpu_packet_in}
 dma_cpu_init_port->b_transport(cpu_packet_in, delay);} }
 else if (trans.is_read()){
 {setting read, adress and data length attribute}
 dma_mem_init_port -> b_transport(generic_packet_out, delay);
 if (generic_packet_out.is_response_ok()){
 generic_packet_out.get_extension(instr);
 {setting noc_packet attributes}
 sw_packet_in.set_extension(nocpckt);
 out_port -> b_transport(sw_packet_in, delay);
}}}}
```

Figure 9. DMA mechanism to receive data from memory and inject it to network

Figure 9 shows part of the *b_transport* function of DMA implemented in TLM2 containing routing mechanism of the received packet from I/O.

These interface functions should be synthesized to a datapath dividing the modules functionality among some submodules. The synthesizable DMA has two submodules in its datapath to guarantee functionality. These submodules declaration in the top entity of the design are shown in Figure 10.

```
Dma_controller* dma_control;
dcd* address_decoder;
SC_CTOR(dma_serial_device) {
 dma_control = new dma("dma_control");
 (*dma_control)(clk,read_mem,write_mem,databus,adbus,
                ready, grant, select_reg, status_rd,
               status_wr, parallel_in);
 address_decoder = new dcd("address_decoder");
 (*address_decoder)(temp,      adbus,      status_sel,
select_reg); …
}
```

Figure 10. Synthesizable DMA module declaration

5. Conclusion and future work

In this paper, we have presented a new method to synthesize the untimed transaction level models by recognizing familiar modules in the design and substituting them with the synthesizable equivalent model of the device. In particular, our synthesized model was functionally equivalent to the original one with extra timing details. This method simplifies the transition from transaction level to lower levels of implementation, while speeding up the whole process. We have validated our approach for a "Direct Memory Access (DMA)" device implemented in TLM v.2 platform and synthesized with a commercial synthesis tool. Further research is directed towards applying the same principle to the other hardware modules, such as other interface types or familiar devices. We should also be looking for ways to synthesize the TLM components individually.

6. References

[1] Open SystemC Initiative (OSCI), OSCI TLM2 user manual, Copyright © 2007.

[2] F. Ghenassia, **Transaction Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems**. Springer-Verlag, 2005.

[3] L. Cai and D. Gajski, "*Transaction Level Modeling: An Overview*", Center for Embedded Computer Systems, University of California, USA

[4] J. Cornet, F. Maraninchi, L. Maillet-Contoz, *"A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip"*, Date 2008.

[5] http://www.forteds.com/products/tlmsynthesis.asp

[6] http://utcadlab.net/Projects.aspx

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

[8] Open SystemC Initiative. IEEE 1666: SystemC Language Reference Manual, 2005. www.systemc.org.