Abstract—This paper presents a methodology for generating interface of a co-emulation system where processor and emulator execute testbench and design unit, respectively while interacting with each other. To reduce the communication time between the processor and emulator, data transfers are performed in transaction level instead of signal level. To do this, transactor should be located near the DUT mapped on the hardware emulator. Consequently, transactor is described in a synthesizable way. Moreover, the transactor design depends on both emulator system protocol and DUT protocol. Therefore, transactor description would not only be time-consuming but also error-prone task. Based on the layered architecture, we propose an automated procedure for generating co-emulation interface from platform-independent transactor. We have also discussed about the practical issues on multiple channel and clock skew problem.

INTRODUCTION

Transaction-level interface has been used to improve performance and design efficiency[1]. Although tools for transaction-level verification have been developed, these are targeted for fully software environment, i.e., high-level testbench connected with HDL simulator[2][5]. These methods locate a transactor within the testbench. Since transactor is described in high-level language such as $C/C++$, transactor design is easy to develop. However, when we apply these methods to co-emulation system, communication between processor and emulator should be done on a clock cycle base instead of transaction base. This incurs large communication time, which in turn degrades overall performance of co-emulation system.

For this reason, in hardware/software co-emulation system, transactor is located within the emulator[4][5]. Data transfer in transaction-level enables efficient communication between processor and emulator. However, in this configuration, transactor should interface with unfamiliar emulator system bus to communicate with the processor. Moreover transactor should be described in synthesizable way to be mapped on hardware emulator. Consequently, design of transactor would be time-consuming and error-prone task.

In this paper, to minimize the efforts for transactor design, we propose an approach to automate the generation of the interface of hardware/software co-emulation system. Starting with emulator platform-independent DUT, transactor and testbench, we propose an approach to complete the co-emulation interface between the testbench and DUT to be applicable to a specific emulation system.

CO-EMULATION SYSTEM ARCHITECTURE

Hardware/Software co-emulation system incorporates two processing engines, which are processor and hardware emulator as shown in Figure 1(a). To implement transaction-based interface on these engines, we use the layered architecture shown in Figure 1(b). To reduce communication overhead between processor and hardware emulator, communication is done in message-based transfer. Thus, we locate the transactor in the hardware emulator where message is decoded and resolved into cycle-accurate signals. The architecture has four layers. Each layer performs a well-defined function. We will discuss each layer of the architecture in turn, starting from the top layer.

From the application layer’s view, testbench is directly connected to DUT, but testbench is described in test case-oriented untimed model ignoring detailed signal protocol. For example, in case where DUT is a kind of memory device, testbench concentrates on the data to store in memory without detailed memory interface protocol signals. DUT is described in cycle-accurate signal-level model such as RTL (Register-Transfer Level) model. The protocol layer is responsible for abstraction-level conversion. Socket creates a message from stimuli of a testbench. The transactor decodes the message...
which in turn is resolved into the detailed cycle-accurate signals. The infrastructure layer is responsible for transferring messages. Messages are transferred through several independent logical channels, each of which connects a software proxy with a associated port macro. The physical layer coordinates the functions required to transmit a bit stream over a physical medium such as PCI bus.

In this layered architecture, user can design transactor without considering the detailed emulator implementations. Emulation user connects transactor to the software testbench through the standard transactor interface (STI) without knowledge of emulation-dependent interface (EDI) protocol.

**CO-EMULATION INTERFACE GENERATION**

In this section, we present an automated process to generate two bottom layers shown in Figure 1(b). Protocol layer should be provided by the designer of protocol which is used for DUT interface. We use SCE-MI (Standard Co-Emulation and Modeling Interface) as an interface between protocol and infrastructure layer[6]. This standard defines API function prototypes for software testbench and protocol definition of macro module for standard transactor interface (STI). However, as these are just shells, we still need to complete the actual design of API and hardware macro.

To perform co-emulation in this environment, emulator user has to prepare bridge netlist. As shown in the top middle of Figure 2, bridge netlist interconnects DUT, transactor and standard macro modules in RTL description using Verilog or VHDL. Through the instantiated standard macros, user can connect transactor to the software testbench without knowledge of the detailed implementation of emulation system, i.e. user don’t need to know the operation of emulation-dependent interface (EDI) protocol for interconnection between processor and emulator. Actual interconnection is done by the automated processes to be presented in the rest of this section.

**Parameter Extraction**

Figure 2 shows the design flow of automatic interface generation. In the first step, the parameter extractor parses emulation user-supplied bridge netlist. Parameter extractor finds out how many transactors are used and macro modules are connected to each transactor. In addition, it extracts the parameters specified on the instantiated macros. On the basis of these information, it generate the parameter file.

Parameter file includes names of transactors connected to DUT, information on the port macros (i.e. name, bit-width) connected to a transactor, information on clock macros (i.e. clock frequency, duty, phase and number of reset cycles) connected to DUT. This generated parameter file is used for software and hardware side to generate co-emulation interface.

**Interface Generation for Hardware Side**

In this step, we complete the bridge netlist through the following two processes. The first process is ‘macro generation’. Bridge netlist just instantiates standard macro module without actual macro design. The design of macro is automatically generated in this process. There are two types of macro: port macro and clock macro. A port macro has two interfaces: one is STI for transactor and the other is EDI for physical channel connection. A port macro converts the EDI protocol into the STI protocol. Port macro can have various bit-width of STI according to the parameter of macro, while the bit-width of EDI is fixed according to the emulation system platform. Assume that bit-width of STI is 128-bit, while bit-width of EDI is 32-bit. The input port macro module forms one STI cycle by aggregating four EDI cycles. Likewise macro generator generates actual RTL design of port macros according to the information in the parameter file. A clock macro generates clocks which is fed into DUT. Macro generator creates clock macros according to the extracted clock parameters such as clock frequency, duty cycle and phase.

The second process is ‘netlist injection’. In user-supplied bridge netlist, there is no netlist related to the physical channel (i.e. EDI). User describes bridge netlist without considering about lower layer implementation (i.e. EDI protocol). In this process, EDI is automatically connected to the standard macros. In the middle of the Figure 2, netlist injector adds new ports to bridge netlist for physical channel connection. This physical channel is distributed by Router module for several port macros. Detailed function of Router module will be explained in section named ‘multiple channel implementation’.

**Interface Generation for Software Side**

In left side of Figure 2, testbench and socket can be implemented as a SystemC module. As testbench here is a untimed model, we can connect these two modules through master-slave interface used for abstract interface in SystemC description. The socket is the counterpart of the transactor in hardware side. When the testbench feeds test cases into the socket, the socket assembles input signals and creates a message - The counter operation is performed by the transactor in hardware side. To send this message to the corresponding macro in hardware side, the socket puts the message into the proxy, which is the counterpart of the macro in hardware side. The proxy is a C++ object which is composed of standard API. The proxy is automatically generated from the parameter file which was created by the parameter extractor in the first step of the design.
flow. API functions are linked with user test environment in compile and link step. In run-time operation, when the socket sends data to proxy, API function refers to the parameter file to find out the destination macro at which the message is expected to arrive.

**ISSUES ON HW/SW CO-EMULATION**

**Multiple Channel Implementation**

At the emulator user’s view point, each proxy is connected with the corresponding macro through individual virtual channel as shown in Figure 3(a). Some application needs a number of virtual channels. For example, when DUT has several interfaces such as serial, USB and ethernet, we need to implement the virtual channels for each interface. Channels are implemented on a physical channel (EDI) such as PCI, which is determined according to the emulation platform. Unfortunately, physical resources are limited. To remove the limitation on the number of virtual channels, virtual channels are implemented on a physical channel in a time-sharing fashion.

In Figure 3(b), we assign a unique number to each proxy and macro pairs for identification. We will use port number to denote this number. Using this number, virtual channels share the same physical channel in a packet switching fashion. Each proxy has own FIFO memory to store incoming messages. Dispatcher routine fetches message from a proxy and attaches header to form a packet, where a header includes port number of source proxy and debugging information such as event time. Original message and attached header forms a packet. Then, dispatcher routine sends the packet through the physical channel. The router module in the hardware side analyzes the header to find out the destination port number. Finally, the message arrives at the right macro module.

In case of software-to-hardware message transfer, dead-lock condition can occur. Assume that two messages are waiting for transfer from the testbench. One message is headed for macro #1 while the other is headed for macro #2. Let us assume that message #1 first arrives at the router followed by the message #2. However, macro #1 is not ready to receive a message until message #2 is fed into transactor. Since two messages are transferred over the same physical channel, message #2 is blocked by the waiting message #1. This deadlock condition can occur according to the design of transactor. To avoid deadlock condition, dispatcher routine checks whether each macro is ready to receive a message. Then, if a macro is ready, dispatcher routine calls a ready callback function previously registered to each proxy object. Before testbench sends a message to a proxy, it checks whether the corresponding ready callback function was called or not.

**Clock Skew Problem**

Transactor and DUT are operated by separate clock signals due to the following two reasons. (We use herein tclk for transactor clock and dclk for DUT clock.) Firstly, they have different clock frequency. Transactor commonly has the same or faster clock frequency than that of DUT, i.e., freq_dclk >= freq_tclk. The reason is that the transactor controls not only the port of DUT but also the clock of DUT. The controlling module (transactor) should have faster clock than controlled module (DUT). Secondly, it is due to possible dclk freezing. Sometimes transactor should hold dclk to hide the status of lower layer from application layer, i.e., testbench and DUT. For instance, if transactor tries to send a data to testbench but channel is not available, then transactor holds dclk to stop operation of DUT until the channel becomes ready. In this way, DUT can interface with testbench without considering the status of the lower layer such as protocol or infrastructure layer. Due to the above reasons, we need to use separate clock signals for transactor and DUT. Moreover, as dclk can be sometimes frozen, we can not employ DLL or PLL circuit to generate clock signals. Figure 4(a) shows the circuit mapped on hardware accelerator such as FPGA. Two clocks (i.e. tclk, dclk) should be synchronized somehow in order to transfer data.
between the transactor and DUT. The alignment of the rising edges of the different clocks (tclk and dclk), in our case, is not physically implementable. For the case of off-the-shelf FPGA, these two clocks use global clock resources in order to minimize clock skew. Due to the difference of routing delay to the global resources, two clocks cannot have the same active edge at the same time. This can cause data transfer error. To solve this problem, we applied the same clock (tclk) to not only transactor but also DUT. To emulate dclk behavior, we automatically replace all flip-flops in DUT into the enabled flip-flops which become active only when dclk edge is supposed to occur as shown in Figure 4(b). To implement this function, HDL parser-based program modifies original flip-flops to be enabled only when enable signal is active.

**Experimental Result**

The automated macro and netlist generation described in the previous sections were implemented in Objective Caml language that is one of functional language, which is suitable to develop HDL parser based program. In software side C++ language is used to implement API which in turn utilizes device driver to access physical channel. In our experiment, the software side is executed on Sun UltraSPARC-II processor. Hardware emulator is implemented as a PCI card featuring 6 million gate Xilinx Virtex-II FPGA. Figure 5 shows the simulation speed of cycle-based and transaction-based co-emulation of a given DUT.

In cycle-based emulation, HDL testbench is running on Cadence NC-Sim simulator, which communicates with the hardware emulator through PLI (Programming Language Interface) interface. Simulator sends input stimulus and advances a clock tick of DUT, then reads the output port to check result at every clock cycles. These data transfers are time-consuming due to the long channel path between the simulator and emulator. Moreover, this synchronization operation is performed at every clock cycle. Therefore, the overall emulation speed is limited by synchronization operation.

On the other hand, in transaction-based emulation, testbench communicates with DUT by sending messages with each other. A single message can generate multiple clock cycles within DUT. We use the notation N to denote the number of clock cycles per message. The time required to send a message through physical channel is defined as \( t_{message} \). Here, \( T_{setup} \) denotes the time to set up a bus transaction over a physical medium. For example, in order to make one PCI transaction, arbitration process is required to use the bus. Once bus is acquired, a number of words can be transmitted in burst data transfer. In this case, \( T_{payload} \) is associated with bus arbitration time and \( T_{payload} \) is the time to transmit one additional word through the bus. When \( B \) is the number of words in a message, finally, the time required to send a message is given as

\[
t_{message} = T_{setup} + T_{payload} \ast B
\]

We can get the speed (cycles per second) as follows:

\[
speed = \frac{(DUT \_clock \_cycles)}{(time \_to \_send \_a \_message)} = \frac{N}{T_{setup} + T_{payload} \ast B}
\]

From the above equation, the emulation speed is proportional to \( N \). Although a message with big \( N \) tends to have big \( B \), the value of \( T_{payload} \) is much smaller than that of \( T_{setup} \). In our experiment, \( T_{payload} \) is hundreds times smaller than \( T_{setup} \). As shown in Figure 5, as \( N \) increases, we can get more emulation speed-up as compared to the cycle-based co-emulation. In our co-emulation experiment, we can get 17.3 kCPS (Cycle Per Second) for the cycle-based emulation. Transaction-based emulation archives up to 813.0 kCPS, which is 47.0 time faster than the cycle-based emulation.

**Conclusion**

In this paper, we present a technique for automatic interface generation in hardware/software co-emulation system, which provides the highly flexible test environment combined with hardware-emulated design unit. To reduce the communication overhead between software and hardware, we use transaction-based transfer. For efficient communication, our method locates the transactor in hardware side. Design of synthesizable transactor in hardware brings efforts to emulation-user. Our approach is based on the layered architecture, which separates DUT-protocol-dependent part from emulation-system-dependent part. We proposed the automated procedure of generating entire co-emulation interface from user-supplied testbench, transactor and DUT. We have also discussed about the practical issues on multiple channels and clock skew problem. Our experiments demonstrated that emulation speed increases by a factor of up to 47.0 compared to the cycle-based co-emulation method.

**References**


