A Method of High-Level Synthesis and Verification with SystemC Language

Volodymyr Obrizan, Kharkov National University of Radio Electronics, Ukraine

***Abstract*—This paper presents a method for automatic RTL-interface synthesis for a given C++ function as well as for a given SystemC-interface. This task is very im-portant in High-Level Synthesis design flow where design entry is usually done in some abstract language (e.g. C++). As a source high-level description targets different SoC architectures or protocols, so it is needed to generate relevant pin-level interfaces and protocols automatically.**

***Index Terms*— Computer languages. High level synthesis, System-level design, System testing.**

I. Introduction

A system level interface can be mapped to different RTL (or pin-accurate) interfaces. The mapping depends on a selected architecture and protocol.

The goal of the presented research is to reduce design time and human efforts needed to generate a pin-accurate interface and a protocol for a given arbitrary high-level description.

The research tasks are:

a) to research the state of the art;

b) to research a mapping between high-level inter-faces and RTL interfaces;

c) to develop a method for generation of a communication protocol;

d) to test the proposed solution.

This paper is organized as follows. State of the art is presented in the second section. The third section defines the prerequisites of the proposed method: a) a mapping between system-level and RT-level interfaces; b) a way how the communication protocol is specified. We conclude in the fourth section.

II. State Of The Art

There are several methods for interface synthesis. The interface synthesis task is defined as follows: to generate an interface between two processes with arbitrary protocols [1]. In [2] authors present a method of interface generation for a given waveform. In [3] authors propose a method to generate interface circuits. The proposed solution produces flexible microarchitectures from FSM descriptions. The FSM descriptions are derived from a formal language called SIMPLE. In [4] authors present a method to generate hardware interfaces and protocols depending on a VHDL description. In [5] authors report on a formal language to specify protocols for further synthesis. None of the above methods solve the problem, if a system model is described in C++.

Among modern System-on-Chip communication protocols we can outline the following: AMBA [6], Wishbone [7], CoreConnect, Open Core Protocol, Avalon [8]. The main disadvantage of these specifications is that they provide the description of the protocols in verbal form with waveforms. There are no algorithms, FSMs, or transactors.

III. A Format Of A Source Description

Let's consider prerequisites which make this method possible. We will consider two main points:

a) a definition of an interface;

b) a definition of a communication protocol.

A. A definition of an interface

Let F — is a function defined with a high-level description language, X = (x1, x2, …, x_i) — is a vector of arguments , Y = (y_1, y_2, …, y_k) — is a function's output vector. Then, a function to be synthesized looks like:

$$Y = F(X). \quad (1)$$

On later design stages, the low-level interface of a module must be defined. However, a number of different low-level interfaces can be associated with a single high-level interface.

In a source model written in C++, a module interface can be defined in the following ways:

a) as a function or member function declaration (SystemC-interface);

b) as a SystemC module (SC_MODULE) with a SystemC pin-accurate interface;

c) in some other way, different from the above.

A function (member function) declaration looks like the following:

```
return_type function_name (argument_list);
```

Manuscript received December 15, 2010.

Volodymyr Obrizan is with Kharkov National University of Radio Electronics, Kharkov, Ukraine (e-mail: volodymyr.obrizan@gmail.com).

here return_type — is a function return value type, argument_list — is a list of arguments, including each argument's type and optional argument's identifier. A C++ function can return values of arbitrary built-in types and user types, pointers or references to them. There is one exception — type void means function doesn't return a value. A mapping between C++ and VHDL types is shown in Table 1.

TABLE I
A MAPPING BETWEEN C++ AND VHDL TYPES

C++ type	Width, bit	VHDL type
char, unsigned char, signed char, bool	8	std_logic_vector (7 downto 0)
unsigned short int, float, short int	16	std_logic_vector (15 downto 0)
int, unsigned int, unsigned long int, long int, double	32	std_logic_vector (31 downto 0)
*T (pointer to T)	32	std_logic_vector (31 downto 0)
long long, long double	64	std_logic_vector (63 downto 0)
wchar_t, void	—	n/a, non-synthesizable

Thus, a high-level synthesis program has the following options: 8, 16, 32, and 64-bit types without intermediate values. These types will be translated into registers and buses of respective widths. Sometimes, it may result in unreasonable hardware expenses. For example, let's suppose that we need to encode 10 values, so we use a variable of type char mapping to an 8-bit register. The register can encode 256 values at the most. But to encode 10 values it is needed only 4 bits. Thus, a half of the register will not be used. In general case, if we need to encode n values, then number of required bits (k) is:

$$k = \lceil \log_2 n \rceil. \quad (2)$$

To address this issue, SystemC provides two type sets to work with arithmetic values with arbitrary precision: from 1- to 64-bit types, 64-bit and above types. Classes sc_int (signed integer) and sc_uint (unsigned integer) model integer arithmetic types in a range from 1 to 64 bits. These classes are recommended for use only for synthesis if unambiguity between high-level model simulation and RTL simulation is needed. Sole-ly for high-level simulations, it is better to use native int type, because its simulation speed is higher. Classes sc_bigint and sc_biguint model 64-bit types and above.

Let's consider the greatest common factor function. Its SystemC-interface is defined as a virtual member function (see below). Then, one needs to inherit this interface and implement its behavior. (This task isn't considered in this paper.)

```
class find_gcf_if : public sc_interface
{
public:
   virtual int find_gcf (int a, int b) = 0;
}
```

A synthesis program can extract number and types with relative bit-widths of the parameters using such description. Fig. 1 shows the result of such synthesis. We see the well-defined RTL interface.

```
int find_gcf(int a, int b);
```

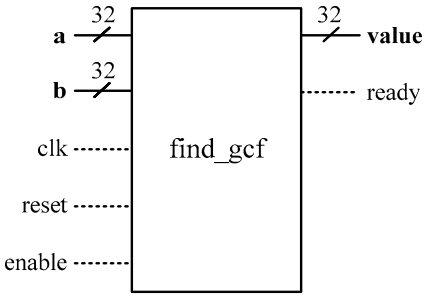

Fig. 1. A function declaration and its RTL interface

There are three groups of ports:

a) informational: a, b, value — the number, names and sizes depend on a given C++ function decla-ration;

b) global control: clk, reset — usually the same for any function.

c) protocol: enable, ready — the number and mean-ing depend on a given protocol.

B. A definition of a communication protocol

Usually, a communication protocol is given as a textual description of rules which one must follow to successfully communicate with a device. Also, such description is supplemented with waveforms. However, these kinds of description aren't good for automatic translation, so a designer should manually specify a communication protocol in VHDL or Verilog.

Let's consider a simple communication protocol. The waveform is shown in fig. 2. The communication rules are the following.

1. Set up the parameters to the inputs 'a' and 'b'.
2. Set up 'enable' signal.
3. Wait until 'ready' signal is '1'.
4. Read the result of the calculation at 'value' port.

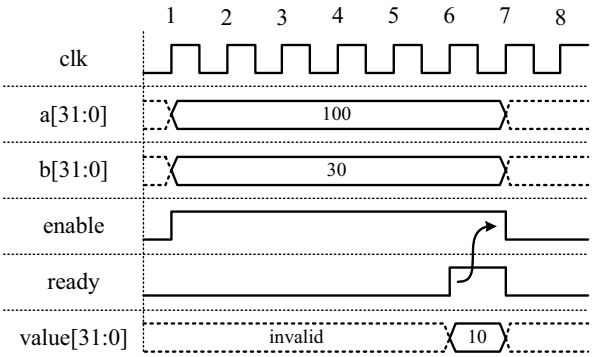

Fig. 2. A simple communication protocol

On the slave's side these rules are defined as follows.

1. Wait until 'ready' signal is set.
2. Read the inputs 'a' and 'b'.
3. Perform the calculation.
4. When the calculation is done, set 'value' output and 'ready' signal.
5. Wait one clock cycle and reset 'ready' signal.

We can represent these rules in a form of the following algorithms. There are two algorithms: one for the master process (fig. 3) and one for the slave process (fig. 5). There are corresponding finite-state machines for these algorithms: fig. 4 and fig. 6 respectively.

The loop at the state a1 in the master process' FSM means that the master process must keep the arguments and 'enable' signal unchanged till 'ready' signal is set. This makes the master process to wait until the slave process is ready with calculations and the result is stable.

Actually, at the FSM in the fig. 5, the state a_1 is a group of states, because most calculations will be done in more than one clock cycle. So this group state can consist of many operation and decision vertices needed to implement the function being synthesized.

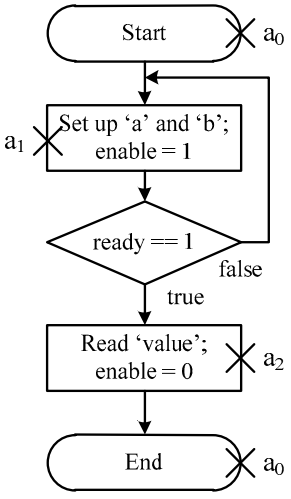

Fig. 3. An algorithm of the master process

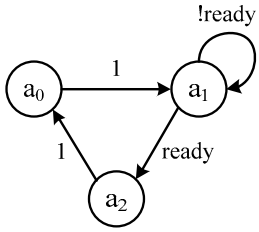

Fig. 4. An FSM of the master process

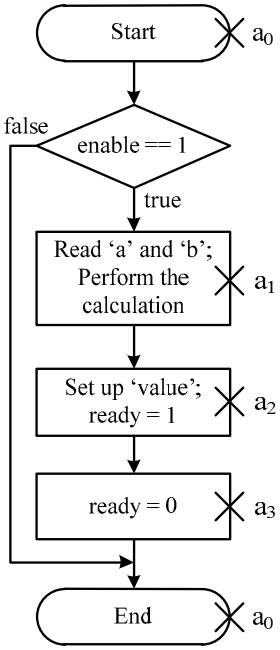

Fig. 5. An algorithm of the slave process

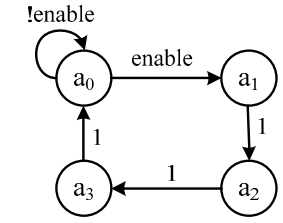

Fig. 6. An FSM of the slave process

In general case, these algorithms depend on a number of arguments and body logic of the function. So it is not a complex task to make the method work for arbitrary function declarations.

These algorithms can be easily synthesized into FSMs and further into RTL interfaces and corresponding protocol logic.

Also, the master process' algorithm can be used to generate a transactor to verify the results of synthesis. This transactor converts higher-level requests (function calls) to lower-level events (logic signals activations). To do so, it is needed to implement the master process algorithm as a SystemC-transactor. For example, for the given function the

master transactor will look as follows:

```
class find_gcf_trans : public find_gcf_if,
   public sc_module {
public:
   // RTL-interface
   sc_out<int> a, b;
   sc_in<int> value;
   sc_in<bool> clock, reset, ready;
   sc_out<bool> enable;

   SC_CTOR(find_gcf_trans) { }

   // High-level interface
   virtual int find_gcf(int a, int b) {
      wait(clock->posedge_event());
      this.a = a;
      this.b = b;
      enable = true;
      while(!ready)
         wait(clk->posedge_event());
      enable = false;
      return value;
   }
};
```

The complete system of transactors and modules is shown in fig. 7. The 'Testbench' module is a high level test-bench written using SystemC language. It has a single port of the *find_gcf_if* type (see section III). The 'H2L' module is a transactor converting high-level calls to low-level binary signals. It works accordingly with the algorithm shown in fig. 3. and the FSM shown in fig. 4. The 'L2H' module is a transactor converting low-level binary signals to high-level system calls. It works accordingly with the algorithm shown in fig. 5 and the FSM shown in fig. 6. The 'find_gcf' module is a high-level module written in SystemC language. However, in this system, the L2H module can be substituted with an RTL or gate-level module. So, the high-level test-bench can be reused on all levels of abstraction.

It should be noted, that the test-bench is a clock-less module. The notion of time is specified in transactors only.

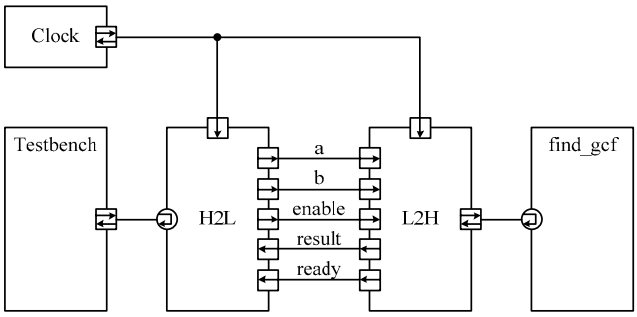

Fig. 7. The complete system of modules and transactors.

IV. Conclusion

A method to generate RTL-interfaces with the simple communication protocol for a given function declaration is presented. This method defines the required informational, control and protocol ports and their bit-widths. We propose to specify communication protocols in a form of algorithms to simplify the interface synthesis task. The method implemented as a part of high-level synthesis tool significantly reduces time to RTL and designer efforts.

Also, the proposed method is useful for automatic generation of transactors to verify synthesized solutions. This enables reusing of high-level tests.

This method doesn't consider a case when it is needed to access data in a shared memory (pointers).

Further research lays in analyzing and defining algorithms and FSMs for popular on-chip protocols (AMBA, Open Core Protocol, CoreConnect, etc.). Also, it is needed to refine the method to handle shared memory access via pointers.

References

[1] Narayan, S. and Gajski, D. D. "Interfacing incompatible protocols using interface process generation". In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, ACM, San Francisco, California, United States, June 12 - 16, 1995, pp. 468-473.

[2] Chung, K., Gupta, R. K., and Liu, C. L. "An algorithm for synthesis of system-level interface circuits". In *Proceedings of the 1996 IEEE/ACM international Conference on Computer-Aided Design*, IEEE Computer Society, San Jose, California, United States, November 10 - 14, 1996, pp. 442-447.

[3] Yun, C., Kang, D., Bae, Y., Cho, H., and Jhang, K. "Automatic interface synthesis based on the classification of interface protocols of IPs". In *Proceedings of the 2008 Conference on Asia and South Pacific Design Automation*, IEEE Computer Society Press, Seoul, Korea, January 21 - 24, 2008, Los Alamitos, pp. 589-594.

[4] Smith, J. and De Micheli, G. "Automated composition of hardware components". In *Proceedings of the 35th Annual Conference on Design Automation*, ACM, San Francisco, California, United States, June 15 - 19, 1998, pp. 14-19.

[5] Madsen, J. and Hald, B. "An approach to interface synthesis". In *Proceedings of the 8th international Symposium on System Synthesis*, ACM, Cannes, France, September 13 - 15, 1995, pp. 16-21.

[6] "AMBA Open Specifications". arm.com/products/system-ip/amba/amba-open-specifications.php

[7] "Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores". Revision: B.3, Released: September 7, 2002 opencores.org/cdn/downloads/wbspec_b3.pdf