

Automatic Interfacing Oriented Core Implementation (For Slave Peripherals)

Matteo Murgida (teomurgi@gmail.com)
Alessandro Panella (breadshe@gmail.com)

January 26, 2006

Contents

1	Introduction	2
2	The State of The Art	2
3	The Proposed Methodology	3
3.1	Standard Signals	3
3.1.1	Clock	3
3.1.2	Reset	3
3.1.3	Interrupts	4
3.2	Naming Conventions	4
4	The Xilinx Case	4
5	The Case Study	4
5.1	32-bit Adder	5
5.2	Counter	6
6	Conclusions and Future Work	6

Abstract

The large amount of time required by core interfacing and the growing demand of reusable code brings the hardware design community to develop methodologies that lead to a better performance design flow. The purpose of this document is to define and explain a set of rules that permit a designer to write a core in VHDL, able to be automatically interfaced with a bus architecture.

After an introduction of the interfacing problem, including a brief presentation of the state of the art, the paper proceed explaining the proposed methodology. In the following section a description of Xilinx architectural solutions precedes two practical examples, together with their synthesis results, that validate the proposed design solution.

1 Introduction

The test of hardware components on reconfigurable devices (such as FPGAs) is one of the most common ways to validate a certain functionality that has to be inserted in a system.

A custom component (which we will refer to with the name "IP-Core") needs to be plugged into an existent system by interfacing it with a bus; it is known that the largest amount of time during an IP-Core implementation is spent solving the interfacing problem. A solution that leads to make fast prototyping by reducing this waste of time is the automatic creation of the interface, operated by a software tool. An efficient methodology is proposed in [1].

In line with this solution, the resulting IP-Core may be composed by three parts:

- The "core", which is the functionality written by the designer;
- The so-called "stub", that implements the following functions:
 - The read process that demultiplexes the input data line to the inputs of the core;
 - The write process that multiplexes the outputs of the core to a single output signal;
- The "component", which contains the interface with the selected bus.

A schematic visualization of these three parts is shown in Figure 1.

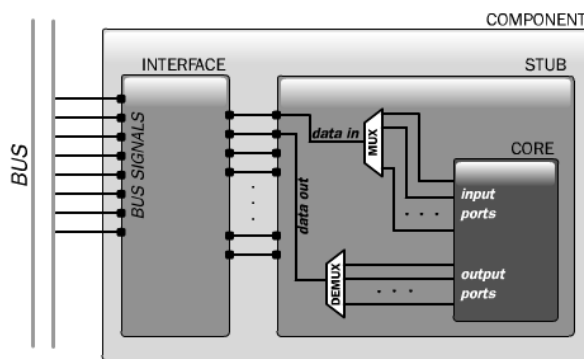


Figure 1: IP-Core general structure.

A problem of primary relevance rises from this methodology: how may be written a core to permit its automatic interfacing? In other words: are there some constraints the designer has to follow in order to make the automatic interfacing of his core possible?

A simple example can explain the problem: make the assumption that the designer wants a clock input for his functionality; it is straightforward to understand that the clock is a "special" signal, and it has to be recognized in order to link it with the bus clock signal. Though the example of the clock signal is very easy to solve, other cases, such as interrupts and the reset signal, could rise more problems.

Moreover, designers are used to implements their cores in relation with the bus architecture chosen for the system. If the designer knows that his component will work on a specific bus, he will integrate the bus signals in his core and he will implement the read and the write processes directly in the core. This is a simple example, from the point of view of automatic interfacing, of a bad written core, since in the solution explained above these functions are included in the stub, that is generated automatically.

The proposed methodology, through the imposition of few writing constraints, has the purpose to free the developer from all the annoying interfacing problems. This will allow him to focus his efforts only on the development of the functionality itself, with no care of the world outside of it.

2 The State of The Art

As can be evinced from [2], actually a common problem the community is facing with is to develop

a generic interface that can: mask to the designer all the signal coming directly from the bus, ease the communication protocol with it and make the IP-cores reusable.

The Virtual Socket Interface Alliance (VSIA) has developed the Virtual Component Interface (VCI) that, separating the interface logic and the behavioral logic, simplifies the connection to different bus architectures. The separation between functionality and interface is operated by a bus wrapper, which implements the connection with the bus. VSIA defines a standard interface between the behavioral logic and the wrapper called VCI. When new bus standard are introduced, a new VCI wrapper is designed. Therefore, the designer has to use the standard signals defined by VSIA in order to connect with VCI.

On the other hand, in this document we deal with the interfacing and IP-core reusability problem on another layer: make the validation of a written core on a reprogrammable board faster using an already implemented commercial interface, without impose to the designer a strict set of signals to be used in his core.

3 The Proposed Methodology

The fundamental idea the designer has to keep in mind while writing his core, is that he does not care about all the issues involved in the component connection with the system. The core must not include any references to the interfacing architecture.

In this way the designer makes an abstraction from the system architectural layer obtaining a remarkable simplification in implementing his functionality.

To give an example the entity declaration of an adder could be:

```
entity core is
  port(
    clk      : in  std_logic;
    reset    : in  std_logic;
    intr1    : out std_logic;
    intr2    : out std_logic;
    intr3    : out std_logic;

    in1      : in  std_logic_vector(0 to 31);
    in2      : in  std_logic;
    out1     : out std_logic_vector(0 to 31);
    out2     : out std_logic
  );
```

```
end core;
```

As it can be seen in the example, there are no ports directly connected with bus signals, such as the bus address line, the bus data line or the current access mode (read or write). All these signals are masked by the stub, which implements all the functions necessary for the correct attachment with the bus, like address decoding.

The set of a component's ports can be divided in two categories:

- Custom Ports, related with the specific functionality, different from core to core (in the former example in1, in2, out1, out2);
- Standard Ports, which are commonly used in every core and carry out basic functions like clock, reset and interrupt (see clk, reset, intr1, intr2 and intr3 in the example).

Custom ports are mapped in the stub by the use of registers that will be written or read through the read and write processes automatically included in the stub.

Due to their special role, standard ports must be recognized by the automatic interfacing tool, in order to permit their correct hooking with the corresponding bus signals. For a slave peripheral these ports are: clock, reset and interrupts. It is important to understand why these ports are to be considered "standard".

3.1 Standard Signals

3.1.1 Clock

Every synchronous device needs a clock signal, which, obviously, can not be mapped in the stub through a register, but needs to be directly linked with the bus clock signal.

3.1.2 Reset

The reset signal is received by the component through the use of a specific port, different from the bus data port. As a consequence, it can not be mapped in the stub like a common data input.

The problem rises because every single core has a particular reset state. In other words, each core interprets the reset signal in a different way. Moreover, the reset has to be caught also by the stub, with the

effect of turning all its registers to zero¹. Therefore, this signal has a double effect on the device, since it affects both the core and the stub.

3.1.3 Interrupts

A device can require the bus master attention launching an interrupt signal. Since this can be generated because of several reasons, the core have to handle more then one interrupt, that has to be mapped by the stub into a single device output signal.

This situation is shown in figure 2.

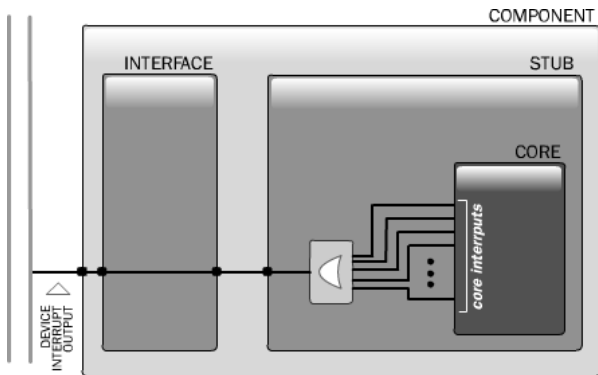


Figure 2: Interrupts propagation from the core to the bus.

Moreover, interface architectures usually provide an interrupt controller. In this case the outgoing interrupt signals have to be filtered by the interrupt controller. For these reasons interrupts can not be considered as custom signals, but have to be detected in order to manage them correctly.

3.2 Naming Conventions

Since the tool which will create the interface must recognize the so-called standard signals, we have to introduce some naming conventions.

- The clock port must be called "**clk**";
- The reset port must be called "**reset**";
- The interrupt ports must be called "**intr1, intr2, ... , intrn**".

The software tool will be able to interpret such signals as standard ones, and will consider them while generating the interface.

¹This effect has been chosen arbitrarily as a convention.

4 The Xilinx Case

In order to propose a methodology that has a practical use into a real design flow of an embedded system, we decided to focus our attention on Xilinx design solutions.

These solutions adopt the IBM CoreConnect® Architecture with the aim of permit the communication between components plugged into the same System-On-a-Chip (SOC). This architecture is composed by three buses [3]:

- PLB: The Processor Local Bus is a 64-bit primary bus designed to be used with devices that need high performances, low latency and design flexibility. It is a direct interface with IBM PowerPC and Microblaze processors.
- OPB: The On-Chip Peripheral Bus is a 32-bit secondary bus architected to alleviate the PLB load and avoid system bottlenecks. It serves lower performance devices like serial and parallel ports, timers, etc.
- DCR: The Device Control Register bus is used only to reduce the capacity loading on the PLB and OPB buses.

Xilinx has developed an interface module, called IP-core InterFace (IPIF), in order to make the interfacing process between IP-core and bus easier; there are two versions of IPIF: PLB IPIF, for PLB attachment, and OPB IPIF, for OPB attachment.

Another way to connect a component to the bus is the use of PSELECT; as the name suggests, it is not a real interface between the IP-Core and the bus, but only an "on-off switch", that permits a component to be selected, and consequently activated, by the bus master.

5 The Case Study

With the intention of validating the proposed methodology we adopted the Xilinx OPB IPIF interface solution [4].

OPB IPIF is a powerful bus interface which provides several services, such as: Address Decoding, Interrupt Source Controller (ISC), Reset / Module Information Register (MIR), Read and Write FIFO, Burst Support and Byte Steering.

In the two tested cores only a set of these services have been used:

- **Address Decoding:** this service operates a translation from the address input to a Chip Enable (CE) `std_logic` array. Each entry of the array corresponds to a register in the device; if an element of the array is set to 1, the corresponding register is activated. There are two separated arrays for write access (WrCE) and for read access (RdCE).
- **Interrupt Source Controller:** this is a complex service able to handle all the interrupts that could be originated by the device. Since interrupts could be generated by the core or by other IPIF services, such as Read and Write FIFO, commonly also a higher level Device ISC, that coalesces the captured interrupts, is needed. In our case the higher level Device ISC is not used because in our device there is no R&W FIFO service. This is shown in figure 3.
- **Reset / MIR:** the reset service allows the system microprocessor to perform a local reset of the device. When the reset input is activated, a reset pulse is sent to the IP-Core and to the various IPIF elements. The reset service can also provide a Module Information Register (MIR), which provides information about the IPIF.

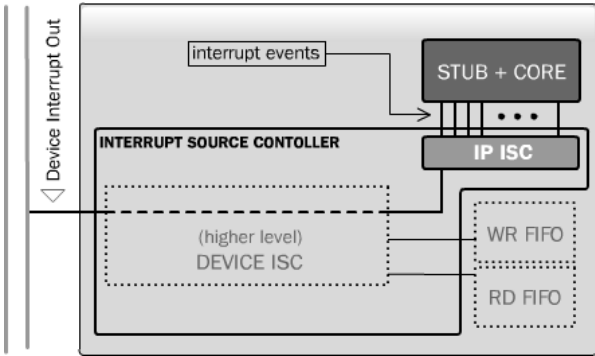


Figure 3: Interrupt Source Control scheme.

We implemented two test components: an 32-bit adder with overflow detection and a simple counter.

5.1 32-bit Adder

This component has three custom ports: two inputs (A, B), the operands, and one output (R), the sum result. The standard ports are: two inputs (clk,

reset) and one output (intr1) which is set to 1 when an overflow occurs.

We synthesized and simulated two version of the component. The first follows the solution adopted in [1] and the proposed methodology, and it can consequently be automatically interfaced. The core entity is:

```
entity adder_core is
  port(
    clk : in std_logic;
    A : in std_logic_vector(0 to 31);
    B : in std_logic_vector(0 to 31);
    reset : in std_logic;
    R : out std_logic_vector(0 to 31);
    intr1 : out std_logic
  );
end adder_core;
```

The second integrates in the same entity both the functionality and the stub. The core entity is:

```
entity adder_core_no_stub is
  generic(
    C_DWIDTH : integer := 32;
    C_NUM_CE : integer := 3;
    C_NUM_INTR : integer := 1
  );
  port(
    Bus2IP_Clk : in std_logic;
    Bus2IP_Reset : in std_logic;
    Bus2IP_Data : in std_logic_vector
      (0 to C_DWIDTH-1);
    Bus2IP_RdCE : in std_logic_vector
      (0 to C_NUM_CE-1);
    Bus2IP_WrCE : in std_logic_vector
      (0 to C_NUM_CE-1);
    IP2Bus_Data : out std_logic_vector
      (0 to C_DWIDTH-1);
    IP2Bus_Ack : out std_logic;
    IP2Bus_Retry : out std_logic;
    IP2Bus_Error : out std_logic;
    IP2Bus_ToutSup : out std_logic;
    IP2Bus_IntrEvent : out std_logic_vector
      (0 to C_NUM_INTR-1)
  );
end entity adder_core_no_stub;
```

As it is clearly understandable from the comparison of the two entities, writing the first one's implementation is easier and less time expensive, because

all the addressing processes are included in the stub which is generated by the software tool, and the designer has to deal only with the operands of its functionality.

The two implementations have an identical behavior as can be evinced from the ModelSIM® simulation wave graphics (See figures 4, 5).

The synthesis final report shows that the two component's versions occupy about the same number of slices on a FPGA: 164 for the first version and 163 for the second one.

5.2 Counter

This component has only two custom input ports(N, start); all the other ports are standard (clk, reset, intr1). When start is set to 1 the counter starts and after N clock periods it rises an interrupt. As done with the former component, also in this case we generated two different implementations, with the same differences between each other as for the adder example.

In this case the slice occupation on FPGA is identical for the two versions, 116 slices.

In both cases a large percentage of the slices occupied by the IP-Core is due to the IPIF, which occupies 92 (of 164) slices for the adder and 71 (of 116) slices for the counter. Anyway, this situation is not representative of a common design situation, since the cores are very small. As a matter of fact, for a large-size core, the number of slices needed by the IPIF remain almost constant and so the percentage the slices occupied by the IPIF within the whole component falls.

6 Conclusions and Future Work

The synthesis of the two components in the previous section shows an important result: the amount of space needed by a device written following the proposed methodology is not larger than a "traditional way" written core. Moreover, as said before, the time to market needed by the core implementation is considerably reduced.

This considerations validate our approach to the interfacing issue. Assuming that it is supported by a portable and flexible automatic generation tool, the solution is efficient also from the point of view of core reusability.

References

- [1] F. Ferrandi, G. Ferrara, R. Palazzo, V. Rana, M. D. Santambrogio. 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06) - Reconfigurable Architecture Workshop - RAW, *VHDL to FPGA automatic IPCore generation: A case study on Xilinx design flow*, April 2005.
- [2] Tien-Lung Lee, Neil W. Bergmann, *An Interface Methodology for Retargettable FPGA Peripherals.*, Engineering of Reconfigurable Systems and Algorithms, 2003: 167-173.
- [3] IBM corporation, *The CoreConnect Bus Architecture, white paper*, International Business Machines Corporation, 2004.
- [4] Xilinx, *OPB IPIF (v3.01b)*, March 21, 2005.