

# IP-Core generator for the WISHBONE bus

Davide Candiloro - 681911 - [davide.candiloro@gmail.com](mailto:davide.candiloro@gmail.com)

June 20, 2006

# Contents

<b>1</b>	<b>Rationale</b>	<b>4</b>
1.1	From Core to IP-Core . . . . .	4
1.2	The aim of IP-Core generator . . . . .	5
<b>2</b>	<b>The Wishbone Bus</b>	<b>6</b>
2.1	Wishbone Interfaces . . . . .	6
2.2	Communication protocol . . . . .	8
2.3	Connection Types: . . . . .	9
<b>3</b>	<b>Usage of IP-Core Generator</b>	<b>12</b>
3.1	Methodology Overview . . . . .	12
3.2	How to write a core compliant to IPGEN . . . . .	12
3.3	How other signals are treated by IPGEN . . . . .	13
3.4	Interface of the program . . . . .	13
<b>4</b>	<b>Implementation of the tool</b>	<b>14</b>
4.1	Overall structure . . . . .	14
4.2	The Wishbone writer in detail . . . . .	14
4.3	A simple example . . . . .	17
<b>5</b>	<b>Enhancements of the tool for more complex architectures</b>	<b>21</b>
5.1	Changes in the YaRA version. . . . .	21
5.2	The new addressing space . . . . .	21
5.3	Architecture Database . . . . .	22
<b>6</b>	<b>Conclusions and future work</b>	<b>23</b>

A very time-consuming task in the process of integrating a Core into an existing design is that of interfacing it to the communication infrastructure of the reference architecture in which that Core has to be tried into, in order to test the effectiveness of the functionality it implements.

In this scenario the provided Core has to be interfaced to a particular bus in order to exchange data with the processor of the architecture, but, to do that, the designer must also take into account the details of the bus he is interfacing to, and, besides the Core itself, a suitable communication interface has to be developed for the design.

In this work a tool to automatically interface a Core to a bus is presented, with the aim of relieving (in part) the duty of implementing a hardware functionality on to FPGAs.

Previous work was done in this direction by realizing an automatic tool for interfacing Cores to the OPB bus with IPIF and Pselect interfaces, in the scope of testing a single Core in a point-to-point master-slave architecture, with the newly designed Core acting as slave.

The efforts done to extend the previous work are twofold:

1. To extend the existing tool to automatically interface Cores to the **Wishbone bus**, which is a more lightweight communication infrastructure with respect to OPB, and has the advantage of being an open specification available in the public domain.
2. To extend the usage of the tool in architectures that are more complex than the simple master-slave testing environment described previously, to provide support also for **exploiting multiple hardware functionalities** in the same architecture, in a more complex environment such as a dynamically reconfigurable architecture on FPGA.

The scope of the project and its aims will be stated more precisely in section 1; A brief introduction to the Wishbone Bus is made in section 2, then the practical usage of the tool is described in section 3; section 4 points out some implementation details of the tool while section 5 presents an extension for more complex architectures. Conclusions and future work are drawn in section 6.

# 1 Rationale

## 1.1 From Core to IP-Core

When implementing hardware functionalities on to FPGAs we usually refer with the term **Core** to the basic component developed to realize a certain function, taken as a standalone device. This component is characterized by the inputs required by the specific functionality and the corresponding generated outputs, plus some signals related to basic synchronization such as a clock or a reset signal.

A further step in the development of hardware functionalities is to make the produced Core part of an architecture, to effectively put to use the functionality that has been realized. This implies providing the Core with some extra hardware to allow communication with a standard bus that can be in turn interfaced with a processor to provide control for the functionality <sup>1</sup>; we thus distinguish the Core itself from its counterpart which includes extra communication hardware. We call this enhanced component with the name of **IP-Core**.

Figure 1 depicts a Core and a IP-Core, both providing the same functionality. In the first half of figure 1, is represented a Core as it is originally devised:

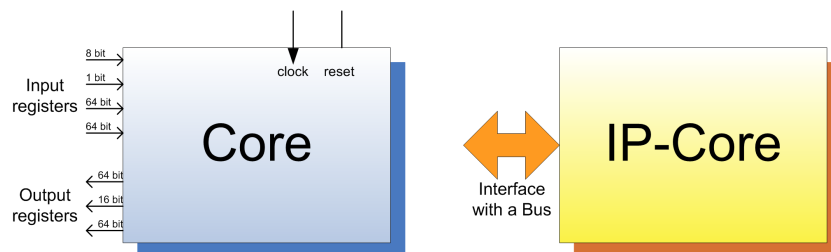


Figure 1: A pictorial representation of the difference between a Core and an IP-Core.

besides the clock and reset signals, some registers specific of an hypothetical functionality are represented. These registers are highly dependent on the functionality of the Core and therefore can have different widths and meanings.

In the other half of the figure the same hardware functionality is represented. The difference with the first half of the figure is that here an interface with a bus is present, to which all the registers and signals of the original Core are mapped, thus composing the actual IP-Core.

---

<sup>1</sup>basically to push input data and to gather outputs

## 1.2 The aim of IP-Core generator

At this point the aim of this project can be stated more clearly: automatically passing from a Core to the corresponding IP-Core without any additional programming effort to be done by the Core designer.

The tool will thus overtake the process of interfacing a core with a particular bus by making the procedure automatic, relieving the programmer from that duty and making it possible to abstract away from the specifications of a Bus when designing a Core.

As mentioned in the introduction, the tool had already been developed for the OPB Bus, both using the IPIF and Pselect interfaces. In this work an enhancement of IP-Core generator also capable of handling the Wishbone bus is described.

## 2 The Wishbone Bus

This section reports the fundamentals of the Wishbone specification used throughout this work. The bus has been chosen to be the communication backbone of some of the architectures currently under development in the micro-architecture lab at Politecnico di Milano; the basic reasons for this choice are outlined next.

The Wishbone standard is a free interconnection architecture for interfacing IP-Cores developed by Silicore.

One important thing about wishbone is that it is not copyrighted, and it is in the public domain. It may be freely copied and distributed by any means. Furthermore, it may be used for the design and production of integrated circuit components to the actual silicon product without paying for royalties or other financial obligations.

Moreover it is simple and lightweight enough not to require bulky or complex hardware as it was for the OPB bus adopted in our earlier designs, as it can be seen from the specification essentials provided hereafter; the following information has been derived from [1], which can be consulted for extensive details about the bus specification.

The standard defines these aspects:

1. an interface to which both master and slave components must adhere;
2. a communication protocol between masters and slaves;
3. four different implementations for interconnecting masters and slaves.

### 2.1 Wishbone Interfaces

The standard defines the interfaces for master and slave components as illustrated in figure 2.

These interfaces must be adopted by every component that will be attached to a Wishbone communication infrastructure. As a naming convention the direction of a signal is indicated by its last letter, in particular the postfix 'I' stands for an input signal and prefix 'O' stands for an output.

The signals of both master and slave are identical but for their direction, with the exception of the clock (CLK) and reset (RST) signals which are provided as an input to each module.

Referring to the master component, the description of each signal is here provided:

- **RST\_I**: when high indicates that the component must reset its memory and its state to their initial condition.

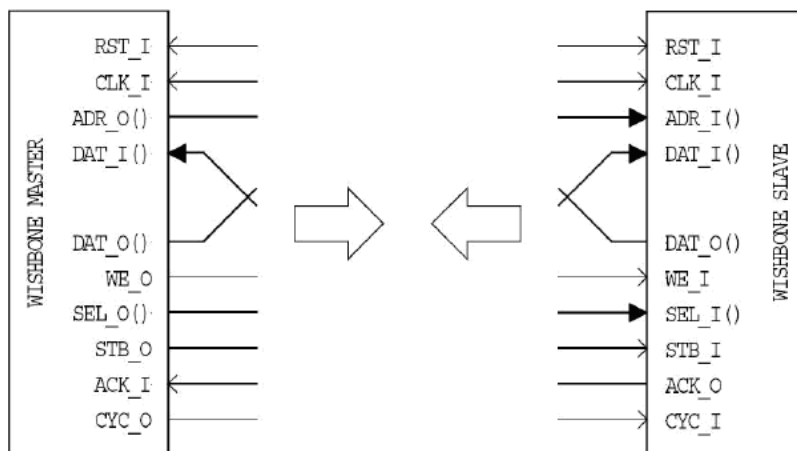


Figure 2: A point to point wishbone-compatible connection

- **CLK\_I**: the clock signal which synchronizes all wishbone components.
- **ADR\_O**: this is the address bus, it is 4 bit<sup>2</sup> wide and it is written by the master to refer to a particular register of a particular slave component.
- **DAT\_I**: 32-bit input data bus.
- **DAT\_O**: 32-bit output data bus.
- **WE\_O**: write enable signal: when high the slave must read from its data bus, when low the slave must write.
- **SEL\_O**: indicates which bytes of the data bus are valid. Since the data bus is 32-bit wide, this signal is composed of 4 bits, one for each byte of the data bus.
- **ACK\_I**: indicates a correct response of the slave.
- **ERR\_I**: indicates that the slave components has suffered some malfunctioning.
- **RTY\_I**: indicates that the slave is not available and tells to retry back later.
- **STB\_O**: when high indicates that all the control signals, data and address set by the master are stable.

<sup>2</sup>The Wishbone specification defines this address bus to be of variable width up to 32 bits. A 4 bit value is used in this first introduction for simplicity's sake. IPGEN will remain flexible toward different address bus widths as specified in the rest of the documentation.

- **CYC\_O**: indicates the beginning of a control cycle of the master.

Slave components have this identical set of signals but with opposite directions, so what is an input for the master becomes an output for the slave and vice-versa, with the exception of the clock and reset signals which are inputs for each of the components.

## 2.2 Communication protocol

In this section the basic control cycles of the wishbone bus are described.

### Reset cycle:

1. RST signal is raised by the wishbone arbiter or by another component.
2. every component reads RST\_I on the raising edge of the clock. If the signal is high, then these components reset themselves to their initial state.

### Reading cycle:

1. The master raises the signal CYC\_O to indicate it has taken the control of the bus.
2. It writes in ADR\_O the address of the slave it wants to read data from, and sets WE\_O low.
3. raises STB\_O to indicate everything is stable.
4. The addressed slave component, when both CYC and STB are high, writes on DAT\_O its data, which will be read by the master on its DAT\_I signal.
5. Slave then raises ACK\_O.
6. When the master detects ACK\_I high on a raising clock edge reads the data on DAT\_I.
7. The master lowers CYC\_O and STB\_O.
8. When the slave detects CYC and STB low, puts back ACK\_O to low level.

### **Writing cycle:**

1. The master raises the signal `CYC_O` to indicate it has taken the control of the bus.
2. It writes in `ADR_O` the address of the slave it wants to write to, and sets `WE_O` high and writes the data on `DAT_O`.
3. Raises `STB_O` to indicate everything is stable.
4. The addressed slave component, when both `CYC` and `STB` are high, reads on `DAT_I` the data.
5. Slave then raises `ACK_O`.
6. When the master detects `ACK_I` high on a raising clock edge acknowledges the write operation as done.
7. The master lowers `CYC_O` and `STB_O`.
8. When the slave detects `CYC` and `STB` low, puts back `ACK_O` to low level.

## **2.3 Connection Types:**

Several interconnection types between master and slave components are possible within the Wishbone specification at various levels of complexity and functionality, from a simple point to point master-slave connection to the sophisticated crossbar switch scheme. A brief presentation of these interconnection modes is presented hereon.

**Point to Point:** This is the simplest possible connection in which a master is simply attached to a single slave. It hasn't no overhead logic for address decoding or collision detection on the communication channel. This interconnection is the same of figure 2.

This configuration is suitable in the IPGen context to test the functionalities of a single IP-Core and will be the reference connection for the implementation of this work

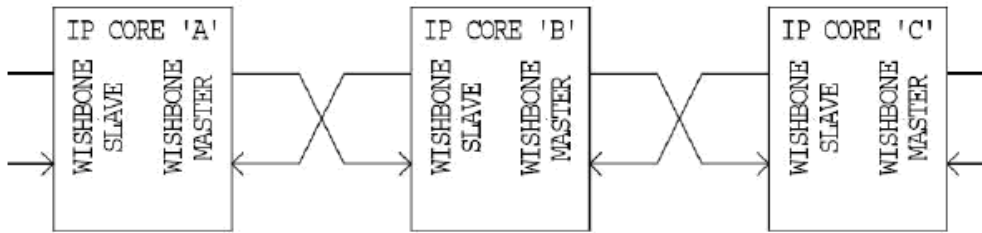


Figure 3: The data flow interconnection mode.

**Data Flow:** It's an evolution of point to point interconnection, particularly suitable for pipeline implementations in which different IP-Cores work in a waterfall fashion on the same stream of data, for example in an image processing application. This scheme requires some address decoding logic to address the data to the right component.

Figure 3 is a schematic of this connection.

**Shared Bus interconnection:** A classic Bus interconnection type between components is also defined in the specification. The bus is a shared resource among components, so address decoding and collision detection (arbitrated bus if there are multiple masters) are needed in order for the connection to work properly.

Figure 4 illustrates a possible interconnection belonging to the Shared Bus type.

**Crossbar Interconnection:** This is the most complex among the possible interconnections: it allows simultaneous communications between different couples master-slave, with not only a single shared bus like the previous case, but a network of connections among components in which some switches connect the proper components according to communication needs. Figure 5 illustrates a possible interconnection belonging to the Shared Bus type.

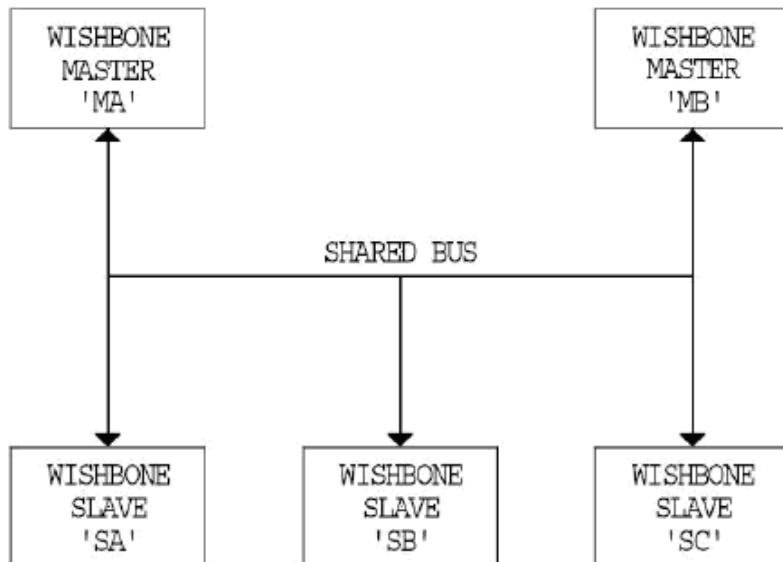


Figure 4: Shared Bus connection example.

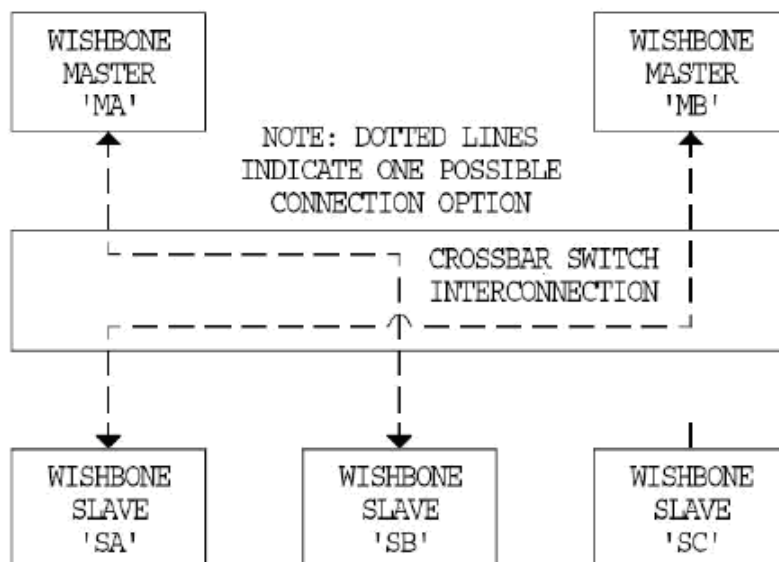


Figure 5: Crossbar interconnection.

## 3 Usage of IP-Core Generator

### 3.1 Methodology Overview

As introduced in section 1 the aim of IP-Core Generator<sup>3</sup> is to provide the designer with a tool that quickly integrates a component in an architecture exploiting a particular bus. Once the core has been completely specified in VHDL language, the relative file can be submitted to the IPGEN tool which in turn will generate an IP-Core complying to the specified BUS.

This IP-Core will then be ready to be integrated and evaluated into the test architecture.

However some basic knowledge about the mechanisms involved in the tool are amenable for the designer to know: it is in fact unfeasible to feed the tool with a Core and expect the consequent IP-Core to be generated without following any constraint in the designing of the original Core. These basic considerations are explained in this section.

### 3.2 How to write a core compliant to IPGEN

Some basic rules are to be followed when writing a core which can be provided as an input to our tool.

These rules are basically some naming conventions to correctly map particular core signals to the correct bus lines: if these rules are followed, the tool recognizes that one of those signals is declared in the core and maps it to the appropriate bus line.

Core ports not recognized as special signals are treated as core registers, and so they cannot be bound to a particular bus line.

A simple example illustrating this fact is provided by the clock signal. If this signal is named in the Core 'clk', then IPGEN will correctly drive it to the IP-Core port interfacing with the wishbone clock. Otherwise, if another name is used, IPGEN won't recognize the signal, and so it will be treated as a normal core register and an address will be assigned to it for memory mapped access, which is obviously unfeasible for the clock signal of a synchronous Core.

The signals recognized as special signals by IPGEN are '*clk*' and '*reset*', which have to be named like this by the core designer.

---

<sup>3</sup>IP-Core Generator is also referred in the remaining part of the documentation as IPGEN

### **3.3 How other signals are treated by IPGEN**

Besides the special signals, the Core will also have its registers to exchange data with the system. The number and the width of these registers is up to the designer, and highly depends on core functionalities.

IPGEN assigns each of these registers an address, in order to be referred by the master during read and write operations.

Registers too wide to fit the 32-bit data word of the wishbone bus are split into smaller words and each of them will be assigned a different address. In this way the core designer is in one more way relieved from taking into account the bus specifications, and has a degree of freedom also in choosing register width.

In essence the designer will be able to define any register of any width (up to filling completely the address space) without caring of constraints imposed by the Bus choice.

### **3.4 Interface of the program**

From the user's point of view, the tool presents itself as a command-line program. The program is invoked typing IPGEN followed by the name of the VHDL file containing the Core to transform. The user is then presented with the choice of the desired bus of the reference architecture and finally the result VHDL file containing the generated IP-Core is created in the folder of the program.

## 4 Implementation of the tool

### 4.1 Overall structure

The tool has been implemented as a C++ program and compiled with the free G++ compiler.

Its basic modules are a *Reader* and a *Writer*, implemented as two classes used by the main program. The task of the former is to parse the submitted VHDL file and to derive a list of the ports of the Core and its generics. This list is then passed to the writer program which will then generate the appropriate IP-Core filling a basic template by processing the information gathered by the reader.

The structure of the program is modular and open to further enhancements: the reader class is the same for all the particular communication facilities the tool has been implemented for; what changes in the different versions is the writer class, which is customized to use a particular communication protocol. The information needed by the writers is however always the same, i.e.: a list of *signal* objects generated from the scanning of the original Core file.

In essence adding support for a new communication infrastructure doesn't imply rewriting the tool from scratch, but instead it means writing a new writer class that takes into account the details of that communication protocol.

The flow diagram in figure 6 describes the overall functioning of the program. Basically, when the program is started, the read method of the reader class is invoked against the file passed as input. If the process of generating the list of signals is successful, the write method of the chosen Writer class is called, and the output file is generated. If any of these operations fails, an error message will be shown to the user.

### 4.2 The Wishbone writer in detail

The writer class leverages on a basic template that constitutes the skeleton of the IP-Core to be generated. This template has placeholders in it that will be replaced by the execution of the program with the appropriate VHDL statements built from time to time from the list of signals previously gathered by the Reader.

The template imposes a structure to the generated IP-Core, which can be seen in figure 7. In this figure all the elements of the architecture in which the core will be put for testing are represented: the processor and the bus of the architecture in the left side of the picture, and the core itself in the lower-right corner.

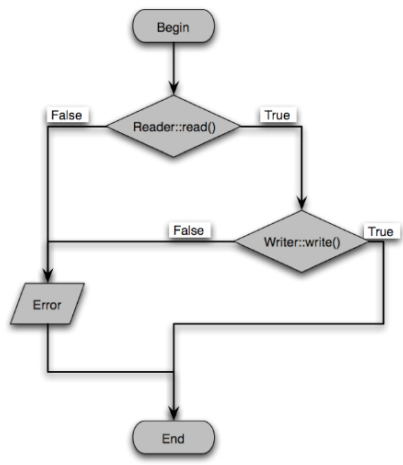


Figure 6: The overall flow of the program

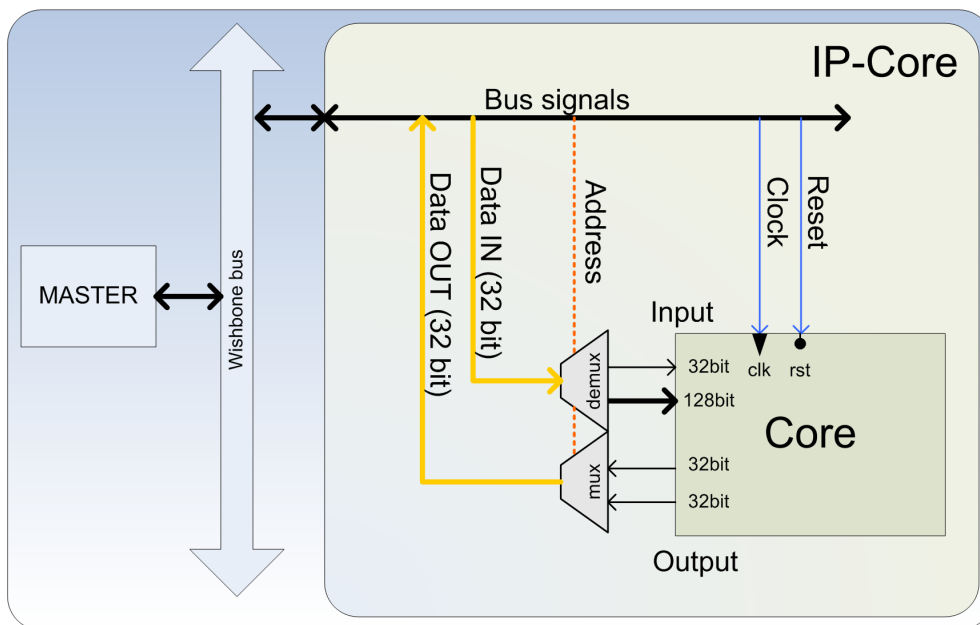


Figure 7: Structure of the generated IP-Core.

The bridge between the Core and the communication bus is the IP-Core, which acts as a wrapper to provide the Core with communication capabilities. Basically the task of the writer is to multiplex and demultiplex the data lines of the bus according to the register addressed on the bus by the processor. In particular, when there is a read cycle on the bus, the correct register of the Core is mapped to the *data out* line of the IP-Core, making it possible for the processor to read that data. The same thing happens for write cycles: when a write cycle is initiated, the IP-Core maps the *data in* port of the Wishbone bus to the register of the core identified by the address word. This task is represented in figure 7 by the multiplexer and the demultiplexer, which respectively map the output registers of the Core to the *data out* port and feed the correct register with data from the *data in* port.

This is done in practice by assigning an address value to each of the registers, being them read or write registers, in such a way that each register has a different memory address that can be referred to by the processor on the other side of the bus. A progressive number is thus generated starting from zero, and this number is then assigned to the register as soon as they're taken from the list previously generated by the reader class, realizing in this way a memory mapping of the core registers.

A case to be taken into account is also when one or more registers of the Core are too wide to be put entirely on the data lines of the bus. Assuming that the *data in* and *data out* lines of the bus have a width of 32 bit<sup>4</sup> a 64 bit wide signal will not be transmitted wholly onto the communication bus, but half of it will be discarded.

To solve this problem IP-Core Generator splits the register into chunks of 32 bit data, and assigns each of them an address in the way illustrated before, so that each portion of longer register words is accessible separately. It will be a duty of the driver to correctly reconstruct the words of data directed to and from the Core.

Finally, a direct mapping between the bus reset and clock signals is made with the respective ones of the core (provided they followed the naming convention illustrated previously).

From an implementation point of view, the program flow of the Writer part of the program is represented in figure 8. This portion starts with the list of the signals of the Core as an input, and scans a template containing a stub of what sketched in figure 7 to complete it and consequently generate the output file.

---

<sup>4</sup>The Wishbone specification defines other widths for the data lines, however we stuck to 32 bit in the architectures we used which implement Wishbone. Changing this parameter in IP-Core generator is a matter of tweaking a value in the code, indeed.

The flow of this portion is thus regulated by the template itself: when a normal character is read, this is copied straight to the output file; when a placeholder<sup>5</sup> is encountered, however, the next character is scanned to see what is the structure to put in that place. In figure 8 the block named "Elaborate signal list and write in the output" serves this purpose. According to the number next to the placeholder, the program generates the appropriate VHDL construct based on the signal list it received at the beginning. Some examples of these constructs are the port mapping, the generation of internal signals of the IP-Core, the read and write processes signal assignments and so on.

When the scanning of the template has finished, the output file will be generated, and the program will terminate leaving in its folder the desired VHDL file of the desired IP-Core.

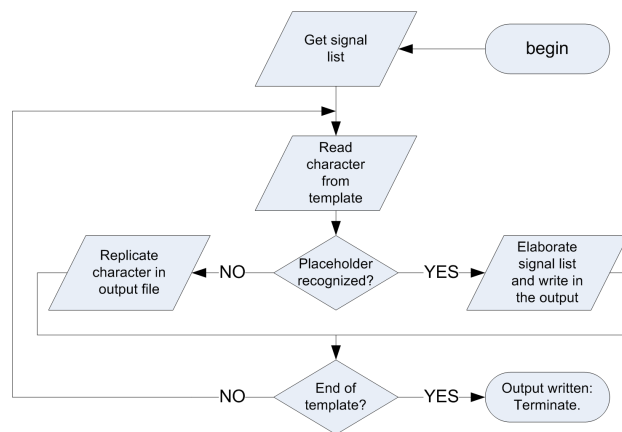


Figure 8: The program flow of the writer portion of IPGEN.

### 4.3 A simple example

To illustrate the results of the execution of IP-Core Generator a simple example is provided, in which a core is submitted to the program and a the corresponding Wishbone-ready IP-Core is generated.

The VHDL code describing the original core is reported here:

```

entity shifter is
port(
    clk    : in    std_logic;

```

<sup>5</sup>This implementation used the # character as a placeholder

```

        reset  : in    std_logic;
        X      : in    std_logic_vector(15 downto 0);
        S      : in    std_logic_vector(2 downto 0);
        Y      : out   std_logic_vector(15 downto 0)
    );
end shifter;

```

```

architecture rrgen of shifter is
begin
    ...
end rrgen;

```

This code describes a basic Core with a clock and reset signals, for synchronization with the rest of the system, and some registers specific to the functionality of the core. Since the functionality described in the implementation of this entity is that of a shifter, the meaning of the registers is the following: register X is the 16 bit input data to be shifted, and S represents the number of the positions of the shifting. Y will hold the shifted value of X after the processing.

Even if this is a very basic Core, it is sufficient to demonstrate the usage of the tool; here is reported the relevant part of the corresponding generated IP-Core:

```

PORT MAP(
    clk => CLK_I,
    reset => RST_I,
    X => register_X,
    S => register_S,
    Y => register_Y
);

reg_rw_process : process(CLK_I) is
begin
    if CLK_I'event and CLK_I = '1' then
        if CYC_I = '1' and STB_I = '1' then
            if WE_I = '1' then
                if Bus2IP_Reset = '1' then
                    register_X <= (others => '0');
                    register_S <= (others => '0');
                else
                    case ADR_I is

```

```

        when "0000" => register_X <= DAT_I(0 to 15);
        when "0001" => register_S <= DAT_I(0 to 2);
        when others =>
            end case;
        end if;

    else if WE_I = '0' then
        case ADR_I is
            when "0010" => DAT_0(0 to 15) <= register_Y;
            when others =>
                end case;
        end if;
        ACK_0 <= '1';

    else if CLK_I'event and CLK_I = '1' and (CYC_I = '0' or STB_I = '0') then
        ACK_0 <= '0';
    end if;
end if;
end process;

```

In the first part (non reported) of this produced file, the original Core is instantiated, the Wishbone ports are declared and the needed internal registers are created. Then a port mapping between the special Core registers (*clock* and *reset*) is made with the proper bus signals, while all the other core registers are mapped to their dedicated internal signal of the IP-Core.

Then main process of writing and reading the registers is declared; On a rising edge of the clock signal, this process checks if CYC and STB are both high, meaning that interaction with the Core is demanded by the bus master. If this situation happens, the WE signal is checked to see if the processor is demanding a read (WE low) or a write (WE high). Depending on the value of WE, thus, the read or write operations take place on the register selected by the ADDR field. After read and write operations an the ACK signal is raised. Finally the last process takes care of lowering back the ACK at the end of the operation, when the bus master has lowered CYC or STB indicating that the operation is concluded.

As it can be seen from the code, the Core registers X, S, and Y are mapped with three different addresses starting from address 0000<sup>6</sup> to provide unique identification of them by the processor.

---

<sup>6</sup>In this implementation the width of the address bus has been fixed to 4 bit. Therefore up to 16 registers can be memory mapped in the core.

Some occupation results of the synthesis of the Core and of its corresponding IP-Core are reported below:

Shifter Core

Number of Slices:	26	out of	4928
Number of 4 input LUTs:	46	out of	9856
Number of bonded IOBs:	35	out of	396

Shifter IP-Core

Number of Slices:	38	out of	4928
Number of Slice Flip Flops:	38	out of	9856
Number of 4 input LUTs:	68	out of	9856
Number of bonded IOBs:	80	out of	396
Number of GCLKs:	1	out of	16

These results have been gathered synthesizing for the AVNET evaluation Board, based on the Virtex II pro FPGA xc2vp7 - ff896, speed grade 5. As it can be seen from the results, the extra hardware inserted around the Core to allow communication with the Wishbone Bus moderately increases the occupation area on the FPGA.

## 5 Enhancements of the tool for more complex architectures

IP-Core Generator as described to this point has been thought for trying Cores into a simple testing architecture for evaluation purposes. However the tool might be very useful also in other contexts, such as the real usage of cores in complex architectures. This section thus presents the improvements and the decisions made to adapt the tool to the usage into a more complex methodology such as that involving an architecture for dynamic reconfigurability on FPGAs, concentrating on the YaRA architecture developed in our lab.

The YaRA architecture is a platform to support dynamic reconfigurability in FPGAs developed at the Politecnico di Milano. IP-Core Generator as been developed to be part of the tool chain of the YaRA architecture, in order to provide an easy instrument to port new cores in the architecture. As the communication backbone of this architecture is the Wishbone bus, several improvements have been made to IP-Core Generator to make it suitable when used in this wider context.

### 5.1 Changes in the YaRA version.

The scenario in which IP-Core generator will be inserted is modified with respect to the version described previously. The former version was in fact suitable for testing a Core only in a well defined architecture where a processor and the Wishbone bus constituted the framework in which to deploy the IP-Core made with the tool, with no more added complexity than this, since the purpose was to quickly test a functionality and not to integrate the Core in a complex system.

With adapting IPGEN for YaRA instead, the tool is required to do some more complex work, because basically multiple cores are attached to the same bus, and memory mapping starting from base address 0 is no more adequate. In the rest of this section the solutions adopted to deal with these new requirements are introduced, being them a new address space model and the architecture database.

### 5.2 The new addressing space

To cope with the possibility of having multiple Cores in the memory mapping, each of them with its own registers, the address space has been extended from the 4 bit of the previous basic version to a wider 8 bit space. This gives the

possibility to address Cores and registers avoiding conflicts. This address space has been divided into two parts, the first three bits representing the Core in the architecture and the last five less significant bits identifying a register (or a chunk of a register) within a Core. Thus the maximum of addressable Cores per architecture is 8, and the maximum number of addressable registers per core is 32. Figure 9 represents an address according

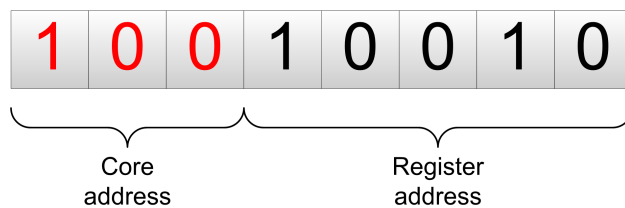


Figure 9: The structure of the address word used in YaRA

to this addressing scheme.

The generated IP-Core is thus slightly changed in the part regarding read and write operations on registers: instead of multiplexing (or demultiplexing) the correct register to the *data in* or *data out* ports, first a check is performed to see if the core is addressed by the master process on the bus checking the first three bits of the address word. Then the other five bits are used to determine the correct register for the read or write operation as described above.

### 5.3 Architecture Database

A further enhancement proposed in this improved version of IPGEN is to take automatically into account the generation of the address of the Core: in fact when dealing with multiple Cores, the tool has to know which is the range of the free Core addresses to which the generated IP-Core has to be mapped.

When designing an architecture, the user would have had to remember the cores previously generated (if he did generate them) or to ask other designers which were the Cores they already prepared for the deployment in the architecture.

The solution proposed tries to solve this problem by maintaining a registry of architectures for which some IP-Cores had already been generated, and for each of these architectures the list of their IP-Cores.

With this sort of *Architecture Database* the tool is provided with enough information to decide the address of an IP-Core automatically, provided the user selects the architecture for which it is generated for.

The address of the IP-Core will be chosen as the first available from the 3 bit word that identifies an IP-Core on the bus.

## 6 Conclusions and future work

In this work a tool for automatic IP-Core generation has been developed, both in a simpler version that allows the testing of a single hardware functionality in a reference architecture exploiting the Wishbone bus in a point-to-point communication fashion, and in a more complex architecture with the Wishbone bus as its communication backbone, with the possibility of interfacing multiple cores to the same bus in a shared bus architecture.

The utility of the proposed tool is to relieve the programmer from the task of bus interfacing, doing this task once for all in the creation of the template for the IP-Core and letting the tool fill properly that template.

Some design choices have supported the creation of the tool, in particular the choice of the widths of the data and address busses. These choices are in no way limiting as the functionality of the program, and, if needed, they can be easily tweaked in the code of the tool to comply with different data and address widths. If needed these parameters can be also exposed to the user interface for a more richness of choices on behalf of the tool user.

In the version of IPGEN for YaRA we will evaluate if these parameters are frequently changing from architecture to architecture, and, if so, they will be saved along with the architecture database.

An interesting and useful future work will be of automatic driver generation for the IP-Cores produced with the tool, in order to take automatically into account the choices made by the program (i.e.: IP-Core address, register address, register splitting) in order to generate a driver template to be filled up by the designer of the Core with the desired functionalities, abstracting away from the details of bus communication in another way.

## References

- [1] **Wishbone Specification** from [opencores.org](http://www.opencores.org)  
<http://www.opencores.org/projects.cgi/web/wishbone/wishbone>;
- [2] Matteo Murgida, Alessandro Panella, Vincenzo Rana, Marco D. Santambrogio, Donatella Sciuto **Fast IPCore Generation in a Partial Dynamic Reconfiguration Workflow**, 2006.

- [3] . Ferrandi, G. Ferrara, R. Palazzo, V. Rana, and M. D. Santambrogio. **Vhdl to fpga automatic ipcore generation: A case study on xilinx design flow.** In 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS06) - Reconfigurable Architecture Workshop - RAW, 2006.