Proposal of the Programming Rules for VHDL Designs

Jan Borgosz and Bogusław Cyganek

AGH – University of Science and Technology
30-059 Krakow, Poland
{borgosz, cyganek}@agh.edu.pl

Abstract. This paper presents novel developments of a programming methodology – in a form of the programming paradigms – for the group of hardware description languages (HDL). On the one hand, this group of languages is very similar to other high-level programming languages. From the other hand a description, which then must fit specific hardware arrays, needs to be coded in a special way with particular attention devoted to the code structure. In this paper it will be shown that the specific programming rules can make this process easier and more efficient. Also clarity of the code can be greatly improved, what is very important in the case of a team work. The complete set of the VHDL programming rules is presented, as well as examples of increased coding efficiency. This approach is much more general and stands in opposition to the other literature sources in which only selective programming advices are provided. Presented ideas were verified in many large telecommunication projects developed on the Xilinx' ISE Foundation platform.

1 Introduction

This paper is devoted to the programming rules for efficient programming in the VHDL language, which belongs to the group of the hardware description languages (HDL) for very high speed integrated circuits (VHSIC) [6]. Coding with clearly set rules is very important part of the properly set team oriented programmers' work. The main results of such approach to the problems are:

1) Shorter time needed for the different people to understand code;
2) Easier maintenance of the code;
3) Shorter time of the code development;
4) Decreased probability of the occurrence of the functional bugs;
5) Faster debugging;
6) An ability to use object oriented and template techniques.

Many effort was put to set programming rules for the high level languages like C, C++. There are many publications and researches about this subject, like [7][8][9][10]. The authors of this work, motivated by benefits listed above, have decided to find similar paradigms for VHDL. The main intention of this document is to show new set of the rules for all the people who are interested in improving coding

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 507–514, 2004.

techniques. This document should be help for all advanced VHDL team members, as well.

2 Review of the Proposals of the VHDL Coding Rules

The programming process is not unique, therefore there are possible many approaches to do the same programming task – some of them are very specific, some provide more benefits than other. In this chapter we have focused on the proposals given by the other programmers and software vendors.

Xilinx is one from the most important software vendors of the modern VHDL tools' market. However, in Xilinx documentation only one appendix treats about this subject. In few words it is possible to say, that Xilinx recommends to keep as close as possible to hardware. Presented rules impose names coming from the FPGA structure. This approach is not very close to programming rules known for high level languages. As a result, projects written in this way can not be easily ported between platforms and scaled. Advices given by Xilinx may be explained by fact, that Xilinx takes care about successful implementation.

It is hard to find any document of these two vendors that would be entirely devoted to the VHDL programming conventions. Of course there are available online helps and tutorials, but after lecture the problem of the naming and programming rules still remains open.

Very interesting proposal of the naming convention may be found in the VFIR project [1]. Authors of this project have made special document with set of rules for VHDL programmers, with simplified ideas close to the ones presented in this paper. They have proposed simplified suffix system to distinguish between objects in VHDL language (strongly hardware dependent). In other work author points a need to use special rules and tries to propose them even for the clock signals edges [2]. There is also an another, and even more general, approach to the programming convention problem [5]. The problem of naming conventions is often omitted or simplified, even in the case of the VHDL handbooks like the VHDL CookBook [2].

3 New VHDL Programming Paradigms

The presented hereafter approach was created and tested in many successfully finalized VHDL projects. Finally, a well balanced solution was achieved that is as general as possible and as close to hardware implementation as needed.

In addition to the naming conventions, some additional rules have been imposed on projects, that brought the whole process closer to the object-oriented approach [4][9][10]. These rules are listed below:

1) Design is divided into building blocks;
2) Each building block is completely independent from the other blocks;
3) Parameters of each block are given with `generic` construction;
4) An interface to each block is constrained with global constants;

5) Blocks are connected in queue fashion, not in the star fashion;
6) Signals which are used in different blocks are passed from block to block;
7) Each block may be replaced or removed without any changes to the neighbor blocks;
8) All synchronous logic in blocks works with the some clock edge.

Usage of keyword `generic` allows to make code closer to the template idea. It is worth to say that a properly set VHDL design is object oriented in its nature.

3.1 General Naming Conventions

In VHDL designs we can distinguish the two most important types of objects:

1) Connections;
2) Building blocks.

Of course, these objects may be divided to subtypes due to its hardware nature. Connections may be signals between building blocks or external input / outputs. Building blocks may be low level elements of the FPGA structure or high level structures composed by the programmer.

General rule for the naming these two kinds of objects is that the instance name describes functionality as shortly as possible. When the direction of signal flow is important, its name should reflect this fact too. Created in this way part of the name will be called a *functionality name*.

So a name of each object in VHDL may be created due to the following concatenation rule:

$$FullName = PrefixName + FunctionalityName + SuffixName(optional)$$

Where *prefix name* comes from type of a VHDL object and *suffix name* is an optional information. Suffix may reflect additional information about associated type of the FPGA's building block hardware.

Authors propose two different rules of combining words into names. In the first rule, words in a name are separated by starting each new word with a capital letter. In the second rule, words are separated with the underscore "_" character. A choice of used convention should be done due to the software environment. For example, for Xilinx software the second convention will be more appropriate, due to the fact that Xilinx software always convert characters to lowercase.

Tables 1-3 contain proposal of the prefixes and suffixes for different objects which can be applied in Xilinx ISE environment (first rule).

Table 1. Set of the rules for design objects – building blocks, words separated with a capital letter

Design element	Rule	Example
Entity	Prefix "E"	EAdder
Architecture	Prefix "A"	ABehavioralAdder
Instantiation	Prefix "I"	IAdder
Package	Prefix "P"	PMyPackage

Table 2. Set of the rules for design objects – signals and miscellaneous, words separated with a capital letter

Design element	Rule	Example
Constant	Prefix “c”	cBitSize
Generic – scalar	Prefix “gs”	gsGenericValue
Generic – vector	Prefix “gv”	GvGenericValue (cBitSize downto 0);
Variable – scalar	Prefix “vs”	vsValue
Variable – vector	Prefix “vv”	vvValue (cBitSize downto 0);
Signal – scalar	Prefix “ss”	ssAlarm
Signal – vector	Prefix “sv”	svAlarm (cBitSize downto 0);
Input port – scalar	Prefix “ips”	ipsStrobe
Output port – scalar	Prefix “ops”	opsStrobe
Input port – vector	Prefix “ipv”	ipvData (cBitSize downto 0);
Output port – vector	Prefix “opv”	opvData (cBitSize downto 0);
Bidirectional port – vector	Prefix “bpv”	bpvData (cBitSize downto 0);
Bidirectional port – vector	Prefix “bpv”	bpvData (cBitSize downto 0);
Array	Prefix “a’	aMemory
Type	Prefix “t”	tDoubleArray
Subtype	Prefix “s” + Type Name	sCollectiontDoubleArray

Table 3. Examples of the rules for design objects – suffixes connected to hardware, words separated with a capital letter

Exemplary hardware element	Rule	Example
BUFGP	Sufix “BUFGP”	ipsClockBUFGP
IBUF	Sufix “IBUF”	ipsResetIBUF
FD	Suffix “FD”	ILatchFD
RAM16x1	Suffix “RAM16x1”	IMemoryRAM16x1

4 Adapting UML for the Illustration of the VHDL Design

It seems to be very interesting to adopt UML convention for the VHDL projects. As mentioned before, a nature of VHDL and hardware projects is close to an object oriented way of thinking. Thanks to this observations it was possible to use modified UML diagrams for illustration of VHDL projects. This idea is depicted in Fig. 1.

As can be seen, a VHDL entity with its name may be treated as a class name, ports may be resources and finally methods may be replaced with functionality. Additionally, a new part was added, called Generics. This new element of the block presents a possibility to parameterize an entity. This is an analogy to the templates in C++ [9]. Due to this assumptions a form of the UML cell for the class (Fig. 1a) may be replaced with a new form for the entity in VHDL (Fig. 1b).

Of course owning relation may be expressed as well. In the next chapter we present simple example of these ideas.

(a) (b)

Fig. 1. a) Standard UML scheme used for the class notation. b) Adopted notation for the VHDL entity

5 Example of the Design with Proposed Programming Rules

As an example we present design with two independent PRBS (Pseudo Random Bit Sequence) generators. Each generator is built with the same parametrical code – a practical demonstration of the template technique. Fig. 2 depicts hardware structure of this design. As may be seen, inputs to the block are: `ipsClock`, `ipsReset`, `ipsPRBSSelect`, and an output constitutes `opsPRBSOutput`. Inputs `ipsClock`, `ipsReset` are common for both blocks: PRBS9 and PRBS11 generators. Outputs from generators `opsPRBSOutput` are multiplexed and finally output of the selected generator is an output from the whole design, also named `opsPRBSOutput`. It is important to see, that high clarity of the descriptions allows to analyze design quickly. Also information about kind of object is coded into its label. Detailed description of the PRBS generator will be presented in the next section.

5.1 Template Implementation of the PRBS Generator

The PRBS generator has architecture declared as follows:

```
entity EPRBS is
            Generic(
              gsFirstXORInput      : integer      := 9;
              gsSecondXORInput     : integer      := 5;
              gsActiveClockEdge    : std_logic    := '0'
                 );
   Port       (
              ipsClock   :  in std_logic;
              ipsReset   :  in std_logic;
              opsPRBSOut :  out std_logic
                   );
end EPRBS;
```

There are three parameters needed to define the PRBS generator structure. Two of them (`gsFirstXORInput, gsSecondXORInput`) come from definition of the PRBS polynomial and the third (`gsActiveClockEdge`) adopts clock edge of the module to rest of the system. A definition of the PRBS generator entity is presented below.

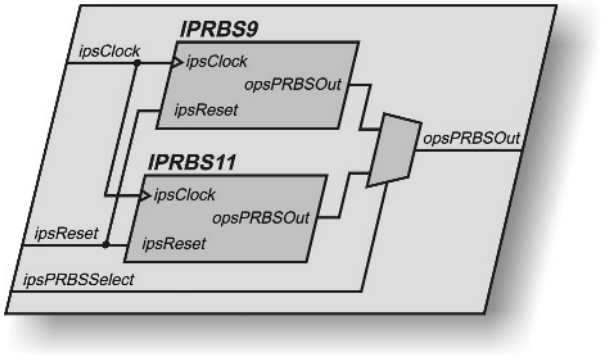

Fig. 2. Hardware structure of the presented example

Please note, that used programming rules immediately allows to identify type of an object. Moreover, any mistakes may be found and corrected faster. Also structure of the generator is more clear than with an ordinary naming convention.

```
architecture EPRBSBehavioral of EPRBS is
  signal svBuffer : std_logic_vector ((gsFirstXORInput) downto 0);

begin
  -- PRBS calculation
  process (ipsReset, ipsClock)
  begin
    if (ipsClock'event and ipsClock = gsActiveClockEdge) then
      svBuffer (0) <= svBuffer (gsFirstXORInput) XNOR svBuffer
                                                 (gsSecondXORInput);
    end if;
  end process;


  -- PRBS output generator
  process (ipsReset, ipsClock)
    begin
      if (ipsReset = '1') then
        opsPRBSOut <= '0';
      elsif (ipsClock'event and ipsClock = gsActiveClockEdge) then
        opsPRBSOut <= svBuffer (gsFirstXORInput);
      end if;
  end process;


  -- Buffer shifter
  process (ipsReset, ipsClock)
    begin
      if (ipsReset = '1') then
        svBuffer (gsFirstXORInput downto 1) <= (others => '0');
      elsif (ipsClock'event and ipsClock = gsActiveClockEdge) then
        svBuffer (gsFirstXORInput downto 1) <=
                          svBuffer ( (gsFirstXORInput -1) downto 0 );
      end if;
  end process;

end EPRBSBehavioral;
```

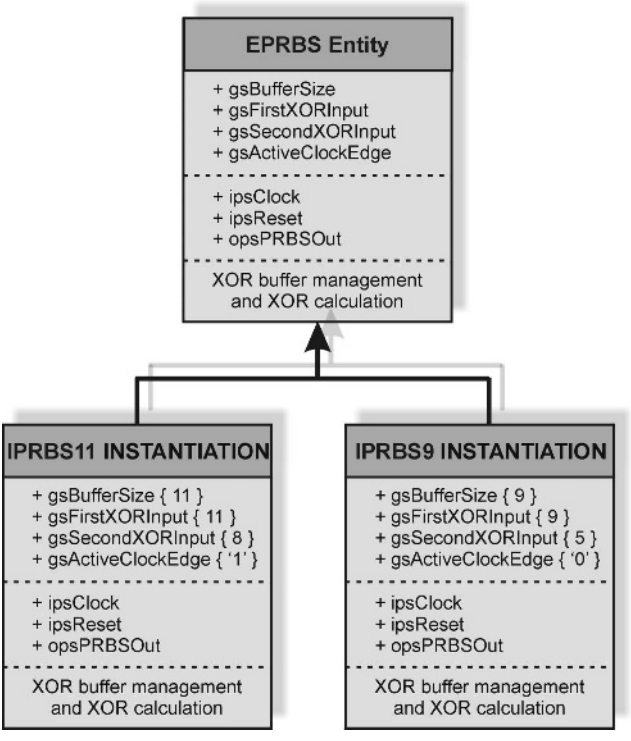

Fig. 3. Illustration of the template instantiation with the modified UML diagram

5.2 Instantiation of the PRBS Generator's Template

Instantiations of the created template are shown below:

```
-- PRBS 9
IPRBS9: ePRBS
Generic map  (
  gsBufferSize          => 9,
  gsFirstXORInput       => 9,
  gsSecondXORInput      => 5,
  gsActiveClockEdge     => '0'
    )
PORT MAP(
  ipsClock => ipsClock,
  ipsReset => ipsReset,
  opsPRBSOut => ssPRBSOut1
);

-- PRBS 11
IPRBS11: ePRBS
Generic map  (
  gsBufferSize          => 11,
  gsFirstXORInput       => 11,
  gsSecondXORInput      => 8,
  gsActiveClockEdge     => '1'
    )
```

```
            PORT MAP(
              ipsClock => ipsClock,
              ipsReset => ipsReset,
              opsPRBSOut => ssPRBSOut2
            );

opsPRBSOut <=   ssPRBSOut1 when ipsPRBSSelect = '0' else
                ssPRBSOut2;
```

Fig. 3 depicts relation between template and PRBS generators, instantiated to the design. Values of all parameters, basic functionality and input / output ports may be easily seen. The proposed programming rules allow to distinguish objects without any troubles.

6 Conclusions

This paper presents a proposal of the new programming rules for the VHDL designs. In opposition to other sources, a detailed description is presented. Also UML diagrams were adopted to make code more clear. Presented ideas were successfully tested with many projects, like STM-1,4,16 framers and mappers (clock speed 155 MHz with 16 bit wide data bus). Thanks to a well balanced and universal approach, the obtained code may be easily resized and reused. For example, an implementation of the error counter and synchronization tracking module for STM – 16 and STM – 4 took 20% of the time needed for STM – 1 (the first module).

References

1. Ameseder T., Pühringer T., Wieland G., Zeinzinger M.:VFIR Coding Conventions 1.0., WWW (2003)
2. Ashenden P.: VHDL Cookbook , White Paper available through WWW.
3. Cohen B.: Coding HDL for Reviewability, MAPLD 2002
4. Cyganek B.:”Programming Paradigms”, White Paper, Krakow (1999)
5. Gord A.: HDL Coding Rules and Guidelines, WWW (2003)
6. IEEE Standard, VHDL, 1987.
7. Meyer S.: Effective C++. Second Edition. Addison-Weseley (1998)
8. Meyer S.: More Effective C++. Addison-Weseley (1995)
9. Stroustrup B.: The C++ Programming Language. Third Edition. Addison-Weseley (1997)
10. Taligent: Taligent Guide to Designing Programs. Addison-Weseley (1995)
11. Xilinx: XST User Guide, Xilinx, USA (2003)