Obfuscation as Intellectual Rights Protection in VHDL Language

Maciej Brzozowski, Vyacheslav N. Yarmolik
Department of Computer Science Bialystok Technical University
Wiejska 45A Street, 15-351 Bialystok, Poland
{brzozowski,yarmolik}@ii.pb.bialystok.pl

Abstract

Software is more and more frequently distributed in form of source code. Unprotected code is easy to alter and build in others projects. The answer for attacks against intellectual rights is obfuscation, a process that makes software unintelligible but still functional. In this paper we review several generic techniques of obfuscation VHDL (Very High Speed Integrated Circuit Hardware Description Language) code and present a set of them for software protection. We are going to describe some related experimental work.

1. Introduction

Subject of obfuscation is so young so it is hard to tell about its history. The first articles date from second half of nineties. The precursor of subject was Christian Collberg. He wrote the first publication connected with obfuscation . In several articles, which co-author he is, Collberg presents techniques of code obfuscation and divide them into four groups by the target application - layout, data, control and preventive transformtions. A lot of articles were written but no one concerned obfuscation of VHDL code. It is very easy to change code of other language like JAVA or C# because size and execute time of transformed program is not so significant. In VHDL these metrics are the most important for a programmer.

2. Definition of obfuscation

For people who are not familiar with the subject of obfuscation, it is intentional action conducting to modify the software code in such way in order to become difficult to understand.

Gal Hachez in [7] changed definition of obfuscation created by Collberg in [4] and also used in [6] to:

Transform a program P into another program P" harder to reverse engineer with the same observable behaviour. If P fails to terminate or terminates with an error, then P' fails to terminate or terminates with an error. Otherwise, P' must terminate and produce the same output as P.

We should remember, that every program P' will possible to reconstruct to P", which will have very similar structure to P. Obviously, such operation will absorb much time and costs, but always it will be possible [1]. Therefore problem of obfuscation is not a protection before decompilation of program, but makes it very difficult.

3. VHDL Source Code

We subject analysis the simple codes of VHDL language -floating point multiplication unit (Figure 1).

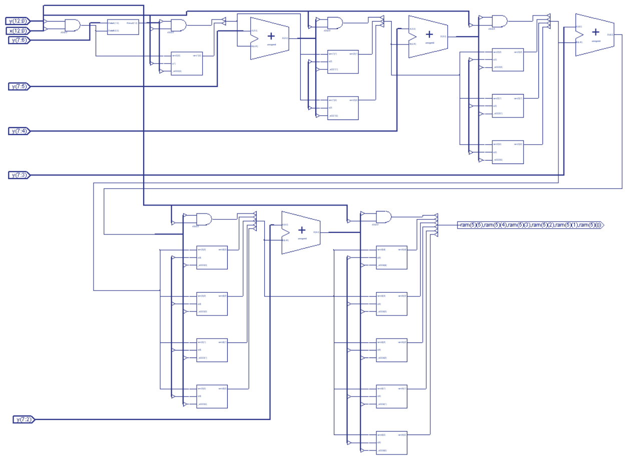

Figure 1. RTL Scheme of floating point multiplication unit (Register Transfer Level)

We analysed four metrics: offset, number of slices, number of slice flip flops, number of 4 input LUTs. Base project was build for Xilinx Spartan2 xc2s100 and before obfuscation metrics had value:

Offset:	29.988ns	
Number of Slices:	42 out of 1200	3%
Number of Slice Flip Flops:	13 out of 2400	0%
Number of 4 input LUTs:	77 out of 2400	3%

4. Layout obfuscation

Layout obfuscation relies on changing (removing) information in the source code that does not effect operation of program. It is often called free because does not influence on size of program neither on its speed of operation after transformation. There is one-way transformation because once deleted information can not be restored. Amount of helpful information decreases for analysing person. This technique contains removing comments, descriptions and changing names of variables which suggest what is it for.

```
signal exponent_1 : std_logic_vector(Exponent-1 downto 0);
                              ⇓
signal O010100101:std_logic_vector(l01010101-1 downto 0);
```

Figure 2. Scrambling identifiers in VHDL code

In Figure 2 identifiers were changed exponent_1 to O010100101. Symbol ⇒ means transformation of source program.

Scrambling identifiers is one and only publicly method available by Semantic Designs. Xilinx Inc. renders accessible components indemnify above-mentioned method. Times of operations and amount of occupied blocks remain always without changes.

5. Control obfuscation

The objective of this transformation is alter control flow rather than computing part of code. Some of them come from compiler optymalization theory.

5.1. Outline statements

This type of obfuscation transforms parts of code into separate methods - in our case into functions and components.

In Figure 3 was outline part of code into component and function. Part of code was modified which was responsible for multiply mantissas of two floating point arguments. The part of code was transferred in project

```
                      change: new_component port map (a, b, y);
y<=a+b;      ⇒                         or
                              y <= function_add (a, b);
```

Figure 3. Outline statements in VHDL code

and situated as a component.

Offset:	31.366ns	
Number of Slices:	47 out of 1200	3%
Number of Slice Flip Flops:	13 out of 2400	0%
Number of 4 input LUTs:	84 out of 2400	3%

Outline the statements is one of the obfuscation simplest methods. In analysed program time and number of logical cells were changed.

5.2. Ordering

Ordering depends on changing the order of each statements. In VHDL such operation has not influence on working program if the transforms of code apply to the blocks of parallel processing

```
y <= a + z;          z <= b + c;
               ⇒
z <= b + c;          y <= a + z;
```

Figure 4. Ordering change in VHDL code

It is not important if in code first will be addition a+z or b+c (Figure 4) because they both execute parallel. A user or a tool can change the order of each lines of code without influence on his functioning.

It is unimportant when in source code first will be enumeration of mantissa or exponent - enumeration of both operations results will execute parallel.

5.3. Loop transformation

Programmers use loops to increase legibility of written code. If we know bounds of loop we might unroll it.

```
for i in 1 to M generate
    y(i)(i downto 0)<=y(M-1 downto M-1-i)
            +y(i-1)(i-1 downto 0)
    when x(i)='1' else '0'& y(i- 1)(i-1 downto 0);
end generate;
                          ⇓
y(1)(1 downto 0) <= y(M-1 downto M-2)
    + y(0)(0 downto 0) when x(1)='1' else '0' & y(0);
y(2)(2 downto 0) <= y(M-1 downto M-3)
     + y(1)(1 downto 0) when x(2)='1' else ...
```

Figure 5. Unroll loop in VHDL code

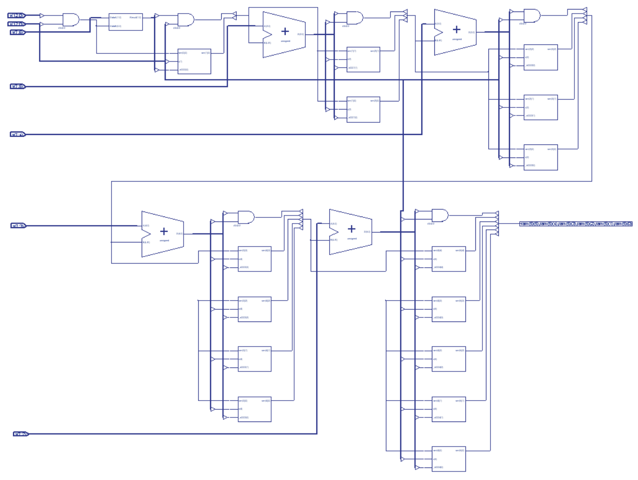

Figure 6. . Scheme of floating point multiplication unit after unroll loop

Offset:	29.988ns	
Number of Slices:	42 out of 1200	3%
Number of Slice Flip Flops:	13 out of 2400	0%
Number of 4 input LUTs:	78 out of 2400	3%

This method used alone is not strong. Rolled loop can be easy rerolled. But in a connection with other tehniques e.g. *ordering* it gives more better results.

Extension of loop increases only the size of source code. The time of operation does not change in relation to project before transforms. The number of used LUT cells grows up slightly.

6. Conversion of parallel processing to sequential

There is transformation which does not appear in other programming languages. In view of its specification VHDL is based on parallel processing instruction. Almost every code written in VHDL may be converted to sequential processing (Figure 7).

```
y_next <= a + b;
                                    process (clock)
process (clock)                     begin
begin                               if rising_edge ...
if rising_edge(clock) then    ⇒         y <= a + b;
    y <= y_next;                    end if;
end if;                             end process;
end process;
```

Figure 7. Conversion of parallel processing to sequential

Mentioned above transform is not difficult to realize. All instructions of serial processing in our project were closed in process - sequential processing. Analysed parameters did not change further.

7. Removing many levels of registers

Originally VHDL is parallel processing language. Programmers can not use variables outside the process so they build multilevel signals.

```
type tablie is array (Mantissa-1 downto 0) of
    std_logic_vector (Mantissa-1 downto 0);
signal ram : tablie;
...
    for i in 1 to Mantissa-1 generate
        ram(i) ¡= ram(i-1) + line_in(i);
    end generate;
                        ⇓
process (clock)
variable ram_variable:std_logic_vector(...);
...
for i in 1 to Mantissa-1 loop
    ram_variable := ram_variable + line_in(i);
end loop;
...
end process;
```

Figure 8. Removing many levels of registers in VHDL code

Offset:	20.563ns	
Number of Slices:	71 out of 1200	5%
Number of Slice Flip Flops:	20 out of 2400	0%
Number of 4 input LUTs:	132 out of 2400	5%

Above transform requires deeper analysis of the project code. The group of signals ram (Figure 9) was replaced by the variable ram_variable. Such transform caused the increase of used LUT cells and decreased time of signal propagation.

8. Conclusion

To conclude obfuscation can be used in many programming languages. It applies to high level languages such as C# or Java and low-level language such as Assembler. In many cases (concernning obfuscation of higher level) these operations are connected with increasing in size of code or lengthening working of program keeping his full functionality. One of the largest limits in using obfuscation of program is time of his execution after transformation. In applications, which task is to communicate with

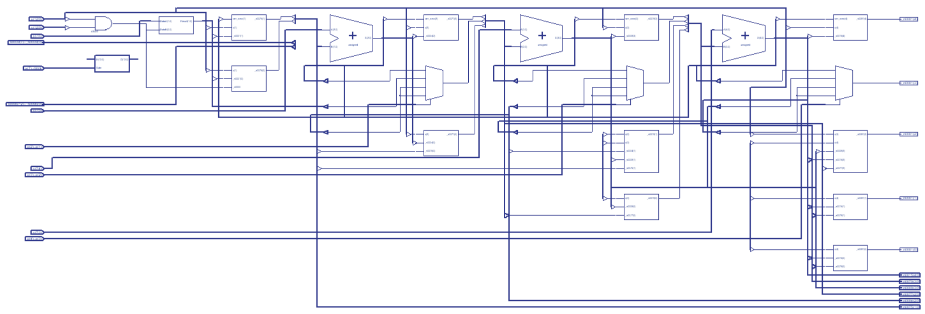

Figure 9. RTL Scheme of floating point multiplication unit after remove multilevel signals

a user, it does not matter if program executes in "extreme" short time. User always can wait for "one moment longer" on execution of task.

However, protecting the code, which brings nothing new or brings little innovative solutions is pointless. Obfuscation of code affects in the largest extents of algorithms and innovative solutions applied in software. It joins with increase of code "complicatedly". The usage of techniques will probably influence (without layout obfuscation) on speed of program operation or on his capacity. It is necessary to consider here time limits of task realization - effects maybe decrease the quality of software (e.g. decrease of signal sampling) or complete his inability to proper working (the algorithm does not execute in proper time).

The passed tests show, that in contrast to other programming languages, the obfuscation of VHDL language does not cause so far changes in program (project) executing time. It is due to the characteristics of language. For example in target architectures (e.g. FPGA, PLD and different) it does not distinguish in such stages as in architecture x86 of registers sixteen or thirty two bits. Both the run time (the propagation of signal) and the size of project after compilation (occupied area on programmable unit) underwent small changes.

All tests were passed on the same compiler with identical settings, programming environment - Xilinx ISE 7.1, target unit Xilinx Spartan2 xc2s100. The unit operating time increased maximally about 5 per cent.

As a result of obfuscation caming into being a project working identically as original one however the occupied area on chip and time of reaction underwent small changes. A user can choose transforms answering requirements of created project.

The simplest transform to use is scrambling identifiers. This method has no limitations in use and its usage does not change parameters of project in no way. Protected in this way code stops being readable for attackers. All presented methods may be use successfully to protect the intellectual property of projects written in VHDL language.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:1–14, 2001.

[2] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, pages 311–324, 1999.

[3] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protecti on. Technical Report TR00-03, Thursday, 10 2000.

[4] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[5] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, pages 184–196, 1998.

[6] C. S. Collberg, C. D. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38, 1998.

[7] G. Hachez. A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards. Universite Catholique de Louvain, March 2003.

[8] J. S. Johnson N. F., Duric Z. *Information hiding - steganography and watermarking - attacks and countermeasures*. Kluwer Academic Publishers, Norwell, 2001.

[9] P. F. A. P. Katzenbeisser S. *Information hiding - techniques for steganography and digital watermarking*. Artech House, Norwood, 2000.

[10] D. Low. Java control flow obfuscation, 1998.

[11] J. Nagra, C. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. In M. J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[12] T. Sahoo and C. Collberg. Software watermarking in the frequency domain: Implementation, 2004.

[13] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

Work was funded with Praca Statutowa S/WJ/3/03 and Praca Wlasna.