PROCEEDINGS OF SPIE

Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX

Harold Szu
Liyi Dai
Editor

27–29 April 2011
Orlando, Florida, United States

Sponsored and Published by
SPIE

Volume 8058

Proceedings of SPIE, 0277-786X, v. 8058

Intellectual Property Protection (IPP) using Obfuscation in C, VHDL, and Verilog Coding

Uwe Meyer-Baese*[a], Encarni Castillo[b], Guillermo Botella[a], L. Parrilla[b], and Antonio Garcia[b]
[a]Department of E&C Eng., Florida State University, Tallahassee, FL, USA 32310
[b]Dpto. de Electrónica y Tecnología de Computadores., Univ. of Granada, 18071 Granada, SPAIN

ABSTRACT

One of the big challenges in the design of embedded systems today is how to combine design reuse and intellectual property protection (IPP). Strong IP schemes such as hardware dongle or layout watermarking usually have a very limited design reuse for different FPGA/ASIC design platforms. Some techniques also do not fit well with protection of software in embedded microprocessors. Another approach to IPP that allows an easy design reuse and has low costs but a somehow reduced security is code "obfuscation." Obfuscation is a method to hide the design concept, or program algorithm included in the C or HDL source by using one or more transformations of the original code. Obfuscation methods include, for instance, renaming identifiers, removing comments or formatting of the code. More sophisticated obfuscation methods include data splitting or merging, and control flow changes. This paper shows strength and weakness of method obfuscating C, VHDL and Verilog code.

Keywords: FPGA, IPP, Obfuscation, VHDL, Verilog, ANSI-C

1. INTRODUCTION

The rapid advance in the design and development of digital integrated circuits, combined with the hard competition in the electronics market, is leading to a substantial change in the design strategies allowing the optimization of company resources. These strategies are based on reusable modules, so called Intellectual Property cores (IP cores), or Virtual Components (VC)[1,2]. These concepts help to reduce development time and costs of up to 70% according to VSI Alliance[3]. Thus, IP-based design has become a major tool within the IC industry. These new design strategies provide precious competitive advantages due to their reduced development time. However, sharing IP cores poses significant security concerns, one of the main being the intellectual property protection of those shared modules[4]. IP modules require author and owner claiming mechanisms to ensure that content will not be illegally used or redistributed by customers who break license agreements.

As motivation consider that the U.S. Chamber of Commerce has published estimates that IP theft costs U.S. companies between $200 to $250 billion a year, as well as 750,000 jobs. The World Customs Organization estimated that pirated and counterfeited goods make up $600 billion annually.

The Intellectual Property Protection (IPP) usually can be split into three major challenges[5], namely

- PIRACY is the illegal copying or resale of goods.
- REVERSE ENGINEERING is the understanding of the idea, algorithm, and code.
- TAMPERING is the change, modifying or extraction of IP cores.

The VSI Alliance[4] has proposed the use of three methods for proper protection of IP cores:

- DETERRENT APPROACHES try to stop illegal distribution by using patents, license agreements, copyrights, trade secrets, and obfuscation. Obfuscations are transformations of the source code to make reverse engineering difficult and can use lexical, control, data or anti-disassembly transformations.

*umeyerbaese@fsu.edu; phone 850-410-6220; fax 850-410-6479

Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX,
edited by Harold Szu, Liyi Dai, Proc. of SPIE Vol. 8058, 80581F · © 2011 SPIE
CCC code: 0277-786X/11/$18 · doi: 10.1117/12.884142

- PROTECTION APPROACHES prevent the unauthorized usage of the IP physically by license agreements, encryption, hardware dongle, and tamper-proofing, i.e., malfunction when modification is detected.
- DETECTION APPROACHES detect and trace both legal and illegal usages of the designs by means of digital signatures, such as digital fingerprinting and digital watermarking. Watermarks can be static (IC layout, IPP@HDL, or embedded figure in IC masks), or dynamic (e.g., Easter egg). Fingerprinting is used to add a unique customer ID to identify the user at fault when IP is illegally used.

Much consideration has been given on watermarking[6-16] and protection approaches in the past, see Fig. 1. Another major focus has been the encryption[17-19] of software in the XOM approach via a crypto controller inserted between program memory and processor[20-23]. For embedded systems, however, we have to consider that a factor of 100 more embedded processors systems with hard cost constraints are produced than powerful general purpose processors (GPP). Embedded processors are designed by small teams with hard time-to-market constraints[24-28]. XOM can not be used with standard embedded system configurations provided by Xilinx or Altera. Little effort so far has been the obfuscation of HDL or C source code. This paper reports on obfuscation methods for HDL and C code and their cost penalties.

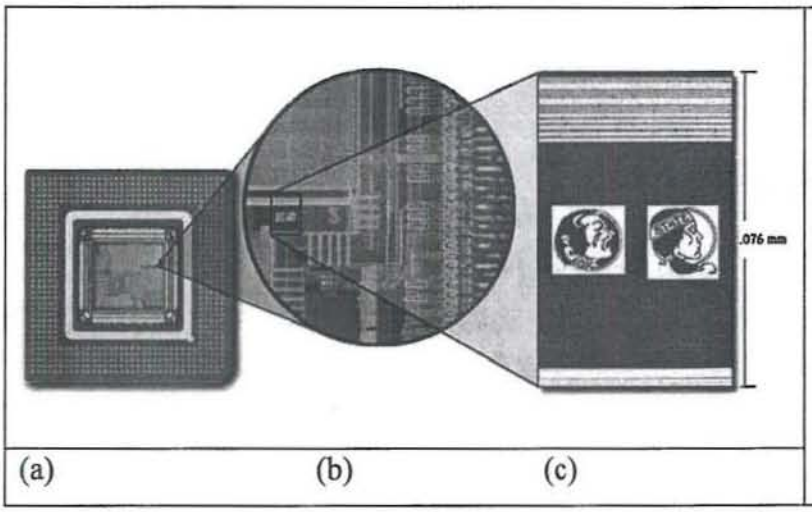


Figure 1. Florida State logo[6] embedded in layout of the Silicon Graphics MIPS R12000 microprocessor [http://micro.magnet.fsu.edu/creatures/technical/packaging.html]. (a) The microprocessor chip in its ceramic package (b) Area on the chip where the logos reside with 500x magnification, (c) 2000x magnification (© 2002 IEEE[6]).

2. SYSTEM DESIGN WITH IP CORES

With the increased demand of time-to-market and the large complexity of today's designs, the reuse of IP block has become and integral design principle. The designer struggles with producing enough gates per day such a multi-million system on a chip (SOC) design is completed on time. Today most new designs with low to medium volumes are based on FPGA technology[29-33] and classic watermarking techniques (see Fig. 1) are no longer viable. For an IP provider the balancing between blocks that can easily be reused and a secure intellectual property protection (IPP) becomes a challenging task. IPs are usually provided[33] in one of the following 3 forms:

1. A **SOFT CORE** is a behavioral description of a component, which needs to be synthesized with ASIC/FPGA vendor tools. The block is typically provided in a hardware description language (HDL) like VHDL or Verilog, which allows easy modification by the user, even new features can be added or deleted.
2. A **PARAMETERIZED CORE** is a structural description of a component. The parameters of the design can be changed before synthesis using a GUI, but the HDL is usually not visible. The majority of cores provided by Altera and Xilinx come in this type of core. Third party IP provider will need substantial investment to make their IP block available in this format.
3. A **HARD CORE** is a physical description, provided in any of a variety of physical layout formats like EDIF. The cores are usually optimized for a specific device (family), when hard realtime constrains are required, like

for instance a PCI bus interface. The reused of the block is very limited and not often used in embedded system design.

For a third party IP provider working on the customization of embedded systems it is not economical to provide hard or parameterized cores, since this is done usually in small teams with hard time-to-market constrains. Obfuscation is then most often the only viable option to protect the provided soft core IP.

3. IPP METHODS USED FOR OBFUSCATION

In todays embedded system design the main idea, algorithms or coding style is a valuable IP and gaining knowledge of someone else work is called *Reverse Engineering*. Now *Obfuscation* is considered a method that transforms the original source into a different kind of representation that:

- Preserves the functionality of the program
- Make the Reverse Engineering difficult
- Make the code more complicated
- Make it unattractive, from a business standpoint, to engage in Reverse Engineering.

Having enough time and money such code transformations usually can still be broken, i.e., Reverse Engineered. In fact it has been shown by Barak[34] at al. that a complete secure obfuscation is not possible, but de-obfuscation will be very time and resource consuming. Obfuscation does not intend to make it impossible to gain knowledge about the IP in question, but it should be more time and cost consuming than to design a new IP block. Not all obfuscation transformations are indeed useful. Changing a two dimensional row/column access to a column/row access for instance would chance the original code and therefore can be claimed to be obfuscation, however from the standpoint of Reverse Engineering the IP is not more difficult to understand. Let us briefly describe some popular metrics as defined by Collberg[35] et al. that classify different method.

3.1 Obfuscation metric, score, and measurements

A real world obfuscator usually will make a sequence of transformation not all at once. In general it would be preferable if each of these transformations can be evaluated separately. Sometimes a combination of several transformations together will be even more beneficial. The obfuscation tools can be evaluated by the following 4 metrics as suggested by Collberg[35-37] at al. However, keep in mind that most evaluations are based on the human cognition ability, and the grading of the transformations is therefore not always a very precise science.

1. **Obscurity or Potency**: Is a measurement of how expensive it is to undo this transformation, or in other words, how difficult it is for a *human* to understand the code after transformation. As *score* low, medium, and high was suggested. As *measurements* it has been suggested that program length, number of predicates, data flow or nesting complexity may be used.
2. **Resilience:** Measures how expensive it is to build an *automatic inverse transformation*. In contrast to obscurity which depends mainly on the human recognition, here the tool development (time) is key. The best transformation can not be reversed and we call this one-way. An example of the one-way would be to remove the comment in the code. Other suggested *score* include *full, strong, weak*, and *trivial.* The reordering of the array mentioned above would be an example of a weak or trivial transformation.
3. **Stealth:** Measures how well the transformation blends into the general coding style. Many transformations add dead code in form of opaque predicates to the code. If we have a program with lot of arithmetic then a statement like
   `IF IS_PRIME(2^512-1) THEN...`
   would blend in just fine. But if the code has only Boolean logic and no arithmetic, e.g., a PCI bus interface, then such a code sequence can easily be identified as an opaque predicate and removed from the code.
4. **Cost:** The question here is how costly is the transformation. Obviously, the transform takes some time and resources. However, transform time and resources used are less a concern. More important is the question if the transformed code still has the same performance, i.e., in HDL we would demand that size (A), speed (T), power, the AT product, or AT^2 product should not change much. For software the main concern would be to preserve

the runtime. An increase in (source) code size, or compile time, on the other hand would be of less concern. As *score* dear, costly, cheap and free has been suggested by Collberg[35] at al.

This four metrics are, by nature of their definition, subject to human recognition capabilities and the skills of the programmer who is doing the Reverse Engineering. In a controlled experiment at least for the identifier obfuscation (discussed next) it could be shown that indeed reverse engineering is complicated through obfuscation[38].

Typically four major classes of transform methods have been defined by Collberg[35] at al. which are lexical, control, data and preventive transformations. Let us discuss in the following some typical obfuscation methods that are not trivial and can be used in obfuscating VHDL, Verilog and C programs.

3.2 Lexical Transformation

The most popular and very useful obfuscation methods rely on the lexical transformation of the code. The simplest step is removing all comments and the formatting from the code. Adding confusing comments has also been suggested[39]. The next step would be to substitute the identifiers with hard-to-read identifiers that no longer reflect the functionality, such as loop counter, enable or clock signals. Fortunately, HDLs and C have similar requirement for their standard identifiers as the Table 1 below shows[40-42]. Escape type identifiers are not considered since they are not often used in practice.

Table 1. Information on identifier requirements (α=[a-zA-Z]; *N*=[0–9]).

	VDHL	Verilog	C
First character	α	α _	α _
Last character	α *N*	α *N* _ $	α *N* _
Other characters	α *N* _	α *N* _ $	α *N* _
LRM required length	∞	≥1024	∞
Altera Quartus support	≥ 2048	≥2048	
Xilinx ISE support	≤ 128	≤ 128	
GCC support			≥2048

The Language Reference Manuals (LRMs) for the three languages do not restrict the maximum length of the identifier. Only Verilog LRM states that at least 1024 characters should be supported. Table 1 reports the support of identifiers by Altera Quartus up to a length of 2048. For Xilinx ISE length 128 worked without problems. The GCC compiler, popular in many embedded processors, was confirmed to work with length 2048 identifiers. A set of lexical obfuscation tools including examples has been posted on the web[47].

A typical C code example for applying all three transformations is shown in Figure 2.

```
for (LOOP=1;LOOP<LMAX;LOOP++){
    k2 = N; dw = 1;
    for (l = 1; l <= S; l++) {  /* Loop Stages */
      k1 = k2; k2 >>= 1;
      w = 0;                    /* Angle count */
```

(a)

```
for(llllllll = 1; llllllll< llllllll; llllllll++){ llllllll = llllllll;
llllllll = 1; for( llllllll = 1; llllllll <= llllllll; llllllll++){
llllllll = llllllll; llllllll >>= 1; llllllll = 0;
```

(b)

Figure 2. Lexical transformation example: (a) Original FFT C-code. (b) Obfuscated code: remove comment, substitute identifier, and remove formatting.

Comments and formatting is removed and identifiers are replaced by length-8, hard-to-read type '1' (one) and 'l' (small L) identifiers. Potency of the layout transformation can be considered to be in the range from low to high based on the skills of the programmer. Cost is free, and transformation is stealthy, since no new language constructs are introduced. Comment removal and scrambling identifiers is one-way, however we do not agree with the formatting assessment in by

Collberg[35] et al.: The format can be mostly recovered, assuming the original code was also formatted with a tool such as `indent` program available in UNIX for C programs.

3.3 Control Transformation

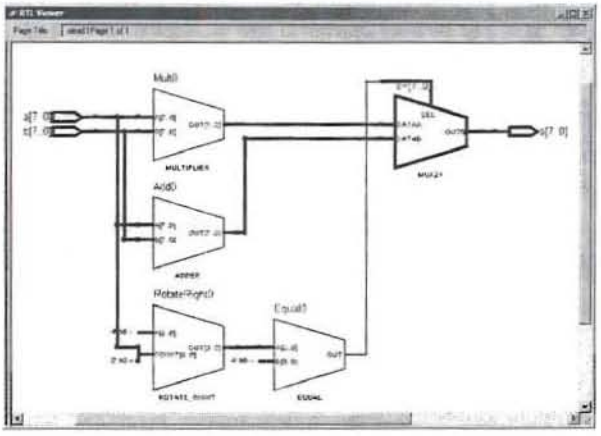
The lexical transformations discussed in the last section are the most popular methods used in available first generation tools, since it is almost guaranteed to be free of cost and still can substantially hinder reverse engineering. The next popular method is the control flow transformation.

The insertion of dead or irrelevant code will have a positive effect on the potency and resilience, and the cost is usually cheap or free. Many suggested transformations rely on *opaque predicate*[36]. The predicates can be always true (P^T), always false (P^F), or sometimes true ($P^?$). In the first two methods only the valid path has the desired code, the other path is never taken, hence the name "dead code." In the last case ($P^?$) two code segments need to be developed that have the same result, (e.g. `x<=2*y;` or `x<=y+y`) since both paths may be taken. This is in general harder to do since a code transformation has to be found that is substantially different, but still has the same output result.

The opaque predicates can be added to IF conditions, LOOP conditions, or by adding redundant operations. Fig. 3 shows a P^T example using Altera Quartus VHDL tools. The $k*L$-times right rotation (ROR) operation of an length-L vector will indeed always results in the same vector, and hence the opaque predicate will always be true. The *potency* of the method shown in Fig. 3 is medium to high given the fact that both code length and number of conditions in the code increases.

```
-- This Is the ALTERA Quartus
-- P^T dead.vhd example
ENTITY dead IS
PORT(a,b: IN INTEGER RANGE 0 TO 2**8-1;
     s: OUT INTEGER RANGE 0 TO 2**8-1);
END;
ARCHITECTURE test OF dead IS
  BEGIN
  -- Free after synthesis;
  -- RTL viewer shows multiplier
  -- s <= a + b; original code
  s <= a + b WHEN "0110" ROR a*4 = "0110"
                         ELSE a * b;
END test;
```

(a)



(b)

Total logic elements	8 / 33,216 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)

(c)

Figure 3. Dead code P^T example: (a) VHDL code. The condition of a rotation $k*L$ of a length L vector will always reproduce the original, i.e., always true. (b) Initial RTL view. (c) Compiler Report.

From the initial RTL viewer diagram in Fig 3(b) we see that this predicate is well *resilient* to the automatic de-obfuscation with the RTL viewer. The cost of this transformation is zero, since after compilation the synthesis result shows the use of just one 8-bit adder, the array multiplier is removed from the netlist. Since the added code and predicate use the same coding style as the original code we can also argue that the transformation is *stealthy*.

The next control flow example is a change in control aggregation by using outlining of statements. Standard arithmetic or Boolean operations are replaced by equivalent function calls. Since the function names are also obfuscated, the code becomes harder to read. Here function names are obfuscated via hard to read '0' (zero) and 'O' (capital o) type identifiers, and placed in a library. Fig. 4 shows an example using Xilinx ISE tools of outlining four operations. Other

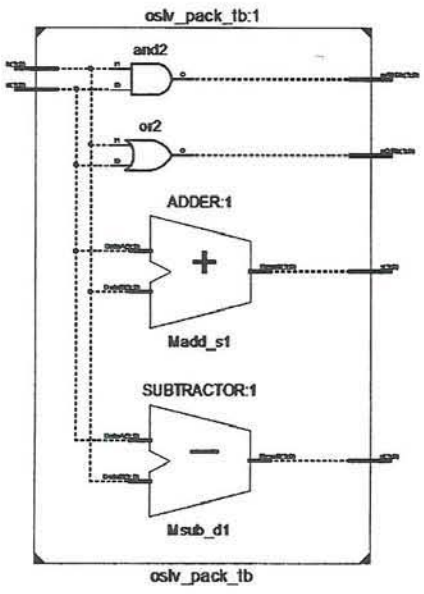
than the claim in the literature[44] our results show that the cost (if outlining is done properly) is free. Potency is medium, resilience is weak, but together with function name obfuscation can be made strong. Transformation is stealthy.

```
-- The Xilinx ISE 12.3 original code
ENTITY olsv_pack_tb IS
PORT (a, b : IN
            STD_LOGIC_VECTOR(3 DOWNTO 0);
      r1, r2, r3, r4: OUT
            STD_LOGIC_VECTOR(3 DOWNTO 0));
END;
ARCHITECTURE test OF olsv_pack_tb IS
  BEGIN
  -- Original code
  r1 <= a + b;
  r2 <= a - b;
  r3 <= a AND b;
  r4 <= a OR b;
END test;
```

(a)

```
-- This the Xilinx ISE 12.3 example
ENTITY olsv_pack_tb IS
PORT (a, b : IN
            STD_LOGIC_VECTOR(3 DOWNTO 0);
      r1, r2, r3, r4: OUT
            STD_LOGIC_VECTOR(3 DOWNTO 0));
END;
ARCHITECTURE test OF olsv_pack_tb IS
SIGNAL  l111, l111, l111, l111, l111, l111
        : STD_LOGIC_VECTOR (3 DOWNTO  0);
  BEGIN
  -- Outline code and ID obfuscation
  l111 <= a_in; l111 <= b_in;
  r1 <= l111; r2 <= l111;
  r3 <= l111; r4 <= l111;
  l111 <=  O000(l111=>l111,l111=>l111);
  l111 <=  O000(l111=>l111,l111=>l111);
  l111 <=  O000(l111=>l111,l111=>l111);
  l111 <=  O000(l111=>l111,l111=>l111);
END test;
```

(b)



(c)

Device Utilization Summary		
Logic Utilization	**Used**	**Available**
Number of 4 input LUTs	12	3,840
Number of occupied Slices	6	1,920
Number of Slices containing only related logic	6	6
Number of Slices containing unrelated logic	0	6
Total Number of 4 input LUTs	12	3,840
Number of bonded IOBs	16	173
Average Fanout of Non-Clock Nets	3.00	

(d)

Figure 4. Outline Xilinx ISE code: (a) VHDL code. (b) Code using outline of operations into a function. (c) RTL view picture of the 4 operations. (d) Compiler Report shows same used Resource (12 LUTs) for both versions.

3.4 Data Transformations

The data transformation methods usually require a high coding effort than control and dead code insertion. Only a few JAVA tools are capable of this type of transformation. Different methods have been suggested in the literature for data transformation. Many rely on array transformations, but have weak resilience. The *change of encoding* is a method that

is stronger and can be used in HDL and C coding. It is discussed in the example from Fig. 5. A Boolean type is split up into several integer representations. In the example the integer 0 to 3 are used to represent the Boolean type.

Binary	Integer= $2b_1+b_2$
0	0
1	1
1	2
0	3

(a)

Right \ Left	0	1	2	3
0	3	3	0	3
1	0	2	1	0
2	0	1	2	0
3	0	3	3	3

IAND

(b)

Right \ Left	0	1	2	3
0	3	1	2	0
1	1	3	0	2
2	2	0	3	1
3	0	1	2	3

IEXOR

(c)

```
...
ENTITY splitting IS
PORT (a, b : IN  BIT;
      c_org, s_org,
      s, c : OUT BIT);
END;
ARCHITECTURE test OF splitting IS
  SIGNAL  a1, a2, b1, b2, ci, si, c1, c2, s1, s2 : INTEGER;
BEGIN
     s_org <= a XOR b; -- Original code
     c_org <= a AND b;
     -- Code using splitting
     a1 <= 1; a2 <= 0 when a='1' else 1; -- Map Boolean a to integer
     b1 <= 0; b2 <= 1 when b='1' else 0; -- Map Boolean b to integer
     si <= IEXOR(a1*2+a2,2*b1+b2);       -- Apply integer EXOR
     s1 <= si/2; s2 <= si-2*(si/2);      -- Compute bits
     s  <= VAL(s1,s2);                   -- Map integer to Boolean s
     ci <= IAND(a1*2+a2,2*b1+b2);        -- Apply integer AND
     c1 <= ci/2; c2 <= ci-2*(ci/2);      -- Compute bits
     c  <= '0' when c1=c2 else '1';      -- Map integer to Boolean c
END test;
```

(d)

Figure 5. Splitting a Boolean into 4 integers. (a) Transformation. (b) Coding of the integer AND operation. (c) Coding of the integer EXOR operation. (d) VHDL code for a half-adder circuit showing first the original code and then the transformed code.

Then Boolean operations are then defined via the integer representation or table operation, as shown in Fig.5(b) for the **AND** and in Fig. 5(c) for the **EXOR** operations. The integer **EXOR** function (placed in an external library) for two bits can be, for instance, implemented in VHDL as follows:

```
FUNCTION IEXOR(l,r: INTEGER) return INTEGER IS
 VARIABLE result : INTEGER;
BEGIN
  result := 0;
  IF (l=0 AND r=1) OR (l=1 AND (r=0 OR r=3)) OR (l=3 AND r=2) THEN result:=1; END IF;
  IF (l=0 AND r=2) OR (l=2 AND (r=0 OR r=3)) OR (l=3 AND r=1) THEN result:=2; END IF;
  IF (l = r) THEN result := 3; END IF;
  RETURN result;
END;
```

For the inverse operation the table in Fig. 5(a) is used, this time from right-to-left.

The RTL viewer shows in Fig. 6(a) the substantial amount of gates used to implement the half-adder circuit that in the original code consist of just one **EXOR** and one **AND** gate. The simulation results in Fig. 6(b) do match for original and splitting designs. Potency of the transform is high, and resilience strong to one-way since the mappings in use are one-to-many and many-to-one and the RTL viewer can hardly optimize this circuit. The transform is stealthy and has low cost. The method can be further improved if a mapping with 8 or 16 integers are used and the integer Boolean operations are (randomly) defined at runtime. However, the cost may no longer be free.

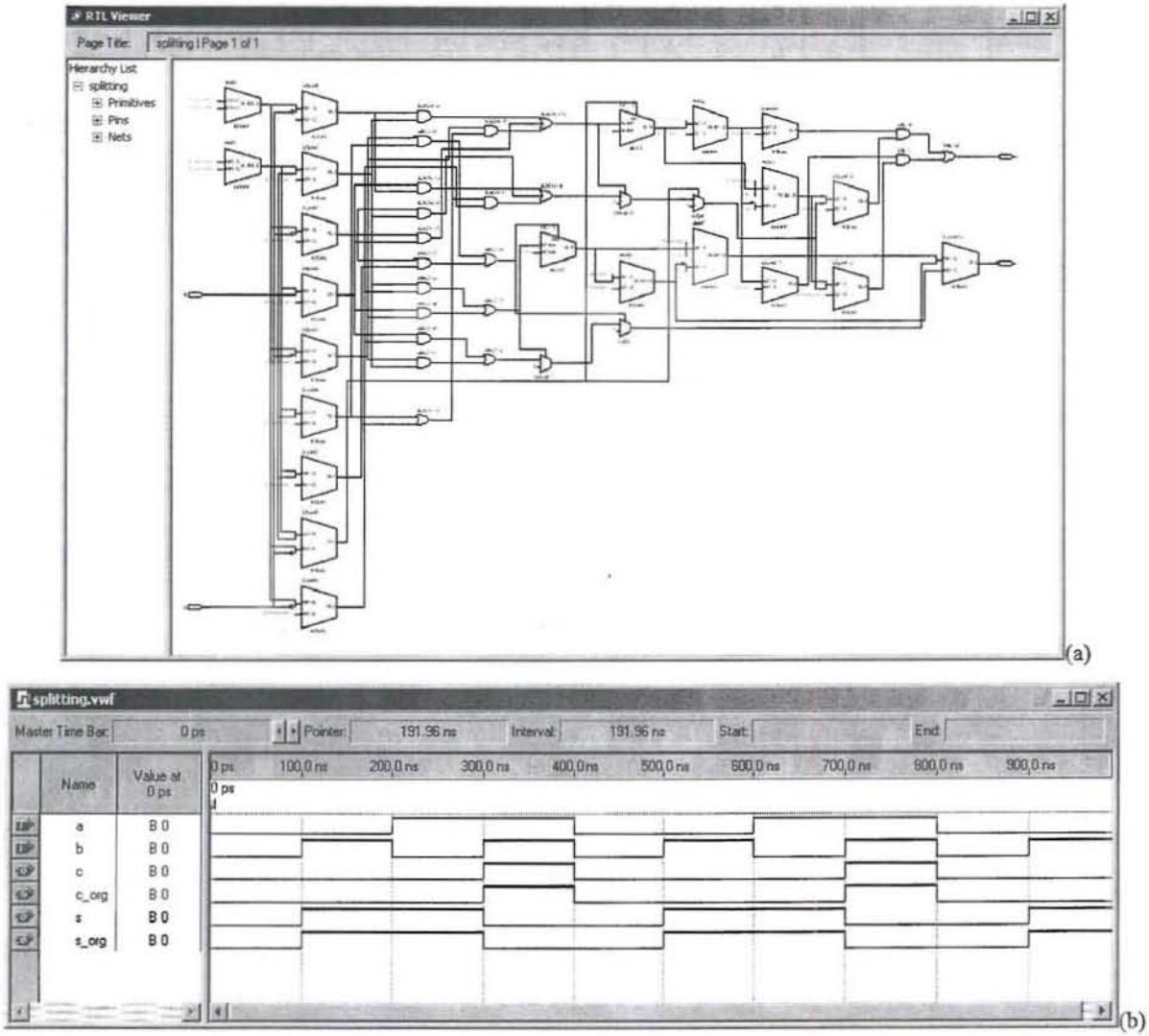


Figure 6. Splitting a Boolean into 4 integers Synthesis and Simulation results. (a) RTL viewer output. (b) Simulation comparing the original and the transformed code.

The following example will demonstrate a data transformation method. It was found to be free of cost in JAVA[35], but not so in HDL. The method is based on merging several arithmetic operations. In the example the original code has the following 4 arithmetic instructions: `u=a+5; v=b+11; x=c*u; y=d*v;` all in 8-bit arithmetic. Now we merge the 8-bit instructions into 16-bit instructions. Fig. 7 shows the resulting Verilog code. Within the `always` statement the HDL analysis is sequential just as in a regular C program.

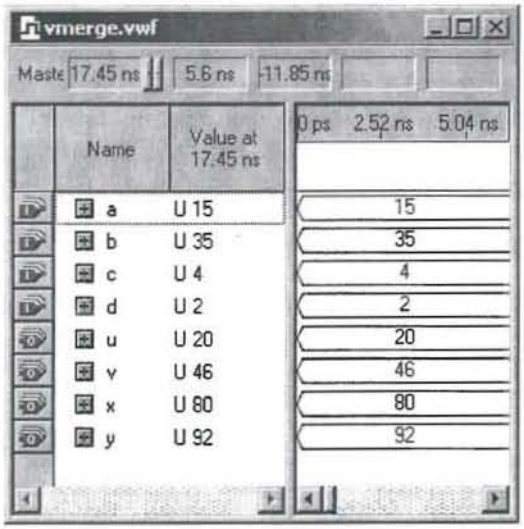
Fig. 7(c) shows that the results are correct. However, from the comparison of the resources we see that the merging method requires additional multipliers and LEs adder resources. To compute the expressions (c-1) and (d-1) 58 LEs are required for the merge compared to 17 LEs in the original. The products now require two 16x16-bit multipliers compared to the two 8x8-bit one of the original code. A software implementation will be dominated by the multiplier time; the additional adder time would not matter much.

```
module vmerge
 (input unsigned [7:0] a, b, c, d,
  output unsigned [7:0]  u, v,
  output unsigned [7:0]  x, y);

// Original 17 LEs and
// two 9x9 bits multipliers
// set a=15, b=35, c=4, d=2;
// in VWF simulator
 assign u = a + 5; // u=20
 assign v = b + 11; // v=46
 assign x = c * u; // x=80
 assign y = d * v; // y=92

endmodule
```

(a)

```
module vmerge
 (input  unsigned [7:0] a, b, c, d,
  output reg unsigned [7:0]  u, v,
  output reg unsigned [7:0]  x, y);

  reg unsigned [15:0] z;
  parameter k = 2821; // i.e.,k = (11 * 256) + 5;
// Needs 58 LEs and four 9x9 bits multipliers
  always@(*)
  begin
    // task: u=a+5; v=b+11; x=c*u; y=d*v;
    z = a + b * 256; // One double length word
    z = z + k;       // Add just one constant
    u = z - (z /256) * 256; // LSBs
    v = z / 256;            // MSBs
    z = z + (c-1) * u;  // Merge 1. multiply
    z = z + ((d-1) * 256) * v; // Merge 2. multi.
    x = z - (z/256) * 256; // MSBs
    y = z >> 8;            // LSBs
  end
endmodule
```

(b)



(c)

```
Top-level Entity Name : vmerge
Family : Cyclone II
Total logic elements : 58 / 17
Total registers : 0
Total pins : 64
Total virtual pins : 0
Total memory bits : 0
Embedded Multiplier 9-bit elements : 4 / 2
Total PLLs : 0
```

(d)

Figure 7. Merge obfuscation Verilog Example. Task at hand are the following 4 arithmetic operation: u=a+5;v=b+11;x=c*u;y=d*v; Data are merged to single long register variable z. (a) Original Verilog code. (b) Merged code. Note that within the always block the computation is sequential. (c) Simulation result for both should match. (d) Synthesis results with/without merge.

3.5 Preventive Transformations

Preventive transformations are used with the goal to crash a particular de-obfuscation program. The newest third generation Java obfuscator like JBCO from the Sable[45-46] group focuses on these preventive transforms. As an example consider adding predicates with side effect. Here two predicates work together. Removing one and not both predicates

only would result in a malfunction. Other method includes using difficult theorems like `IF IS_PRIME(2^512-1) THEN`... Also, complicated pointer/array transformations used in predicates have been shown to be difficult for the de-obfuscators to solve. However, pointer structures are not supported in HDLs and other difficult predicates need to be found.

Since currently no de-obfuscator for HDL or C exists at time of writing, designing HDL preventive transforms for de-obfuscators will be a concern for the future.

4. ACKNOWLEDGEMENT

The authors would like to thanks Altera and Xilinx for the provided hardware and software under the University programs. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors. The help of the following consultants and program manager is acknowledged: B. Esposito, S. Brown, R. Maroccia, M.Phipps (Altera) and J. Weintraub, A. Acevedo, C. Sepulveda, C. Dick (Xilinx). U. Meyer-Baese gratefully acknowledges support for this work from the Humboldt research foundation. Products and company names used in this article may be trademarks of their respective owners.

5. CONCLUSION AND FUTURE STUDY

This paper discusses obfuscation methods for C, VHDL and Verilog languages. Commercial first generation tools that perform lexical transformations and include watermarking capability are available for VHDL from VISENGI. Semantic Designs (http://www.semdesigns.com/products/obfuscators/) offers first generation tools for VHDL, Verilog, and C languages and others. A second generation obfuscator KRYPTON was available from the French[43] company LEDA, but has been discontinued. The only available open source HDL obfuscator so far has been the Verilog obfuscator from EDA utilities, see http://www.eda-utilities.com/vo.htm. Many obfuscation tools are available for JAVA, see http://www.dmoz.org.

Three first generation open source lexical obfuscators for C, VHDL and Verilog languages have been developed and posted online[47]. In this paper advanced control and data obfuscation methods, as used in second generation obfuscators, have been discussed in term of potency, resilience, stealth and cost. It turned out that HDL transforms and JAVA obfuscation do not always have equal quality measurement results.

Future study will include the development of strong opaque predicates that RTL viewer in HDL and C compiler cannot break but are still cost free. Also, HDLs offer additional language features that allow developing obfuscation methods not available with normal C or Java code. One example is the use of concurrent code. Here in HDLs the statements are evaluated concurrently, i.e., the ordering of the statements does not matter. Combined with a wire permutation, code with high potency and cost free can be developed.

REFERENCES

[1] Keating, M. and Bricaud, P., [Reuse Methodology Manual for System-on-a-Chip Designs], Kluwer Academic Publishers, (1998).
[2] Amos, D. Lesea, A. and Richter, R., [FPGA-Based Prototyping Methodology Manual] free ebook http://www.synopsys.com/Systems/FPGABasedPrototyping/FPMM/Pages/default.aspx(last accessed 3-22-2011)
[3] VSI Alliance, "The Value and Management of Intellectual Assets," White Paper, V1.0. (June, 2002).
[4] VSI Alliance, "Intellectual Property Protection: Schemes, Alternatives, and Discussion," White Paper, V1.1. August, (2002).
[5] Naumovich, G. and Memon, N. "Preventing Piracy, Reverse Engineering and Tampering," IEEE Computer, 64-71 (2003).
[6] Goldstein, H., "The secret art of chip graffiti" IEEE Spectrum, 39(3), 50-55 (2002).
[7] Cox, I., Miller, M. and Bloom, J., [Digital Watermarking: Principles & Practice], Morgan Kaufmann, (2001).
[8] Kahng, A.B., Lach, J., Mangione-Smith, W. H., Mantik, S., Markov, I. L., Potkonjak, M., Tucker, P., Wang H. and Wolfe, G., "Watermarking Techniques for Intellectual Property Protection," Proc. of the Design Automation Conference, 776-81 (1998).

[9] Lach, J., Mangione-Smith, W.H., Potkonjak, M., "Fingerprinting Techniques for Field Programmable Gate Array Intellectual Property Protection," IEEE Transactions on Computer-Aided Design, 20(10), 1253-61 (2001).
[10] Charbon, E. and Torunoglu, I., "Watermarking techniques for electronic circuit design," *in* Lecture Notes on Computer Science, Vol. 2613, 147-169, (2003).
[11] Castillo, E., Meyer-Baese, U., Parrilla, L., García A. and Lloris, A., "Watermarking Strategies for RNS-based System Intellectual Property Protection," Proc. of 2005 IEEE Workshop on Signal Processing Systems SiPS'05, Athens, 160-165 (2005).
[12] Castillo, E., Parrilla, L., García A., Lloris A. and Meyer-Baese, U., "IPP Watermarking Technique for IP Core Protection on FPL Devices," Proc. of 16th International Conference on Field Programmable Logic and Applications FPL'2006, 487-492 (2006).
[13] Castillo, E., Parrilla, L., García A., Lloris A. and Meyer-Baese, U., "Intellectual property protection of IP cores through high-level watermarking," Proc. SPIE Int. Soc. Opt. Eng., Vol. 6576 (2007).
[14] Castillo, E., Meyer-Baese, U., Parrilla, L., García A. and Lloris, A., "IPP@HDL: Efficient Intellectual Property Protection Scheme for IP Cores," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15(5), 578-591 (2007).
[15] Castillo, E., Meyer-Baese, U., Parrilla, L., García, A. and Lloris, A.,"New advances for automated IP soft-core watermarking," Proc. SPIE Int. Soc. Opt. Eng., 7343-0L-1-12 (2009).
[16] Parrilla, L., Castillo, E., Meyer-Baese, U., Botella, G., García, A., González, D., Todorovich, E., Boemo, E. and Lloris, A., "Watermarking strategies for IP protection of micro-processor cores," Proc. SPIE Int. Soc. Opt. Eng., 77030L1-10 (2010).
[17] Schneier, B., [Applied Cryptography], John Wiley & Sons, New York (1996).
[18] Menezes, A., van Oorschot, P., and Vanstone S., [Handbook of Applied Cryptography], CRC Press, Boca Raton, FL (1996).
[19] Welschenbach, M. [Cryptography in C and C++], Apress, Berkley, CA (2001).
[20] Lie, D., Thekkath, C., Mitchell, M., Licoln, P., Boneh, D., Mitchell, J. and Horowitz, M., "Architecture Support for Copy and Tamper Resistant Software, ACM ASPLOS-IX, 168-177 (2000).
[21] Yang, J., Zhang, Y. and Gao, L., "Fast Secure Processor for Inhibiting Software Piracy and Tampering," Proceedings of the 36th International Symposium on Microarchitecture, 1-10 (2003).
[22] Yang, J., Gao, L. and Zhang, Y., "Improving Memory Encryption Performance in Secure Processors," IEEE Transactions on Computers, 54(5), 630-640 (2005).
[23] Mahar, A., Athanas, P., Craven, S., Edminson, J., and Graf, J., "Design and Characterization of a Hardware Encryption Management Unit for Secure Computing Platforms," Proceedings 39th Conference on System Sciences, Hawaii, Vol. 10, 251b (2006).
[24] Meyer-Baese, U., Sunkara, D., Castillo, E. and García, A., "Custom Instruction Set Nios-based OFDM Processor for FPGAs," Proc. SPIE Int. Soc. Opt. Eng., 6248U-1-10 (2006).
[25] Zurawski, R., ed., "A Novel Methodology for the Design of Application-Specific Instruction-Set Processors," in Embedded System Handbook, CRC Press, Boca Raton, FL (2006).
[26] Vera, A., Meyer-Baese U. and Pattichis, M., "An FPGA based rapid prototyping platform for wavelet coprocessors," Proc. SPIE Int. Soc. Opt. Eng., 657615-1-10 (2007).
[27] Meyer-Baese, U., A. Vera, A., Rao, S., Lenk, K., Pattichis, M., "FPGA Wavelet Processor Design using Language for Instruction-set Architectures (LISA)," Proc. SPIE Int. Soc. Opt. Eng., 65760U-1-12 (2007).
[28] Meyer-Baese, U., Botella, G., Castillo, E. and García, A.,"Nios II hardware acceleration of the epsilon quadratic sieve algorithm," Proc. SPIE Int. Soc. Opt. Eng., 77030M-1-10 (2010).
[29] Meyer-Baese, U., Vera, A., Meyer-Baese, A., Pattichis, M. and Perry, R., "Discrete Wavelet Transform FPGA Design using MatLab/Simulink," Proc. SPIE Int. Soc. Opt. Eng., 624703-1-10 (2006).
[30] Meyer-Baese U., Natarajan, H., Castillo, E. and García. A., "Faster than the FFT: The chirp-z RAG-n Discrete Fast Fourier Transform," Frequenz, Vol. 60, 147-151 (2006).
[31] Meyer-Bäse, U., Natarajan, H. and Dempster, A., "Fast Discrete Fourier Transform Computations Using the Reduced Adder Graph Technique," EURASIP Journal on Advances in Signal Processing, Vol. 2007, Article ID 67360, 8 pages (2007).
[32] Meyer-Baese, U., Vera, A., Meyer-Baese, A., M. Pattichis, M., R. Perry, R., "Smart Altera Firmware for DSP with FPGAs," Proc. SPIE Int. Soc. Opt. Eng., 65760T-1-11 (2007).
[33] Meyer-Baese, U., [Digital Signal Processing with Field Programmable Gate Arrays], 3nd ed., Springer-Verlag, Berlin (2007).

[34] Barak, B., O. Goldreich, O.,Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., "On the (Im)possibility of Obfuscating Programs," Advance in Cryptology – Crypto 2001, Proc. LNCS 2139, Springer-Verlag, 1-18 (2001).
[35] Collberg, C., Thomborson, C. and Low, D., "A Taxonomy of Obfuscating Transformations," Technical Report #148, Dept. CS, Uni. Of Auckland, New Zealand (1997).
[36] Collberg, C., Thomborson, C. and D. Low D., "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," Proceedings Principles Of Programming Languages, POPL'98, 184-196 (1998).
[37] Collberg C. and Thomborson, C., "Watermarking, Tamper-Proofing and Obfuscation – Tools for Software Protection," IEEE Transactions on Software Engineering, 28(8), 735-746 (2002).
[38] Ceccato, M. Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano M. and Tonella, P., "The Effectiveness of Source Code Obfuscation: an Experimental Assessment," IEEE 17th International Conference on Program Comprehension, 178 – 187 (2009).
[39] Lifshits, Y., "Lecture Notes: Program Obfuscation and Cryptography," Invited Course at Tartu University in Spring 2006.
[40] ISO/ANSI Programming languages –C, http://www.open-std.org/jtc1/sc22/wg14/, Sec. 6.4.2 Identifiers, p.44 (last accessed 3-22-2011)
[41] VHDL 1076-2008 Language Reference Manual, IEEE Explore, Section 15: Lexical elements
[42] Verilog® hardware description language, IEEE Standard 1364-2005, Ch3: Lexical conventions, p. 14
[43] K. O'Brien K. and S. Maginot, S., "Non-Reversible VHDL Source-Source Encryption" Proceedings of the conference on European design automation, Grenoble, France, 480-485 (1994).
[44] Brzozowski M. and Yarmolik, V., "Obfuscation as Intellectual Rights Protection in VHDL Language" Proceeding Int. Conf. on Computer Information Systems and Industrial Management Applications, p. 1-4 (2007).
[45] Naeem, N., Batchelder, M. and Hendren, L., "Metrics for Measuring the Effectiveness of Decompilers and Obfuscators," Sable Technical Report no. 2006-4, School of CS, McGill University (2006).
[46] Batchelder, M. and Hendren, L., "Obfuscating Java: the most pain for the least gain," Sable Technical Report no. 2006-5, School of CS, McGill University (2006).
[47] Meyer-Baese, U., "128 Length Obfuscator Tool Set" (online) http://www.eng.fsu.edu/~umb/o4.htm (last accessed 3-22-2011)