Soft IP Protection: Watermarking HDL Codes

Lin Yuan, Pushkin R. Pari, and Gang Qu

Department of Electrical and Computer Engineering
and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742
{yuanl,pushkin,gangqu@eng.umd.edu}

Abstract. Intellectual property (IP) reuse based design is one of the most promising techniques to close the so-called design productivity gap. To facilitate better IP reuse, it is desirable to have IPs exchanged in the soft form such as hardware description language (HDL) source codes. However, soft IPs have higher protection requirements than hard IPs and most existing IP protection techniques are not applicable to soft IPs. In this paper, we describe the basic requirements, make the necessary assumptions, and propose several practical schemes for HDL code protection.

We protect the HDL codes by hiding author's signature also called as watermarking, similar to the idea for hard IP and multimedia data protection. But the new challenge is how to embed watermark into HDL source codes, which must be properly documented and synthesizable for reuse. We leverage the unique feature of Verilog HDL design to develop several watermarking techniques. These techniques can protect both new and existing Verilog designs. We watermark SCU-RTL & ISCAS benchmark Verilog circuits, as well as a MP3 decoder. Both original and watermarked designs are implemented on ASICs & FPGAs. The results show that the proposed techniques survive the commercial synthesis tools and cause little design overhead in terms of area/resources, delay and power.

1 Introduction

Design reuse and reuse-based design have become increasingly important and are widely considered as the most efficient way to close the design productivity gap between silicon capacity and designer's ability to integrate circuits onto silicon [27]. For reuse to be successful, the reusable building blocks, also known as macros, cores, intellectual properties (IPs), or virtual components, must be easily accessible and integrable. Several industry organizations such as the San Jose-based "Virtual Socket Interface Alliance", the "design and reuse" in Europe, and "IP highway" in Japan have already started building libraries and tools that can be shared by designers all over the world. More importantly, they are working on the specification of various IP design standards for IP integration. But how to guarantee IP provider's IP rights and royalties remains one of the major obstacles for design reuse.

J. Fridrich (Ed.): IH 2004, LNCS 3200, pp. 224–238, 2004.

IP exchange and reuse normally takes the forms of hard, firm, or soft. Hard IPs, delivered as GDSII files, are optimized for power, size, or performance. Soft IPs are delivered in the form of synthesizable HDL codes. Firm IPs, such as placement of RTL blocks or fully placed netlist, are a compromise between hard and soft IPs [24]. From security point of view, hard IPs are the safest because they are hard to be reverse engineered or modified. But this one-fits-all solution does not give IP users any flexibility other than the built-in configuration options. Soft IPs, on the other hand, are preferred by IP users due to their flexibility of being integrated with other IPs without much physical constraints. On some occasions, IP provider may also prefer releasing soft IPs to leave customer-dependent optimization process to the users. Not surprisingly, it has been recognized that the IP market will be dominated by soft IPs [10]. However, the flexibility makes soft IPs hard to trace and therefore difficult to prevent IP infringements from happening. IP providers are taking a high risk in releasing their IPs in the soft form without protecting their HDL codes with techniques that are effective, robust, low-complexity, and low-cost. Unfortunately, such techniques or tools are not available and their development is challenging.

Most existing VLSI design IP protection mechanisms, such as physical tagging, digital watermarking and fingerprinting, target the protection of hard/firm IPs. Traditional software obfuscating and watermarking methods are not applicable to HDL code either. In this paper, we 1) analyze the challenges in HDL code protection; 2) describe the basic requirements and necessary assumptions; 3) develop the first set of Verilog source code protection methods. Our approaches can be easily integrated with the design process to protect a new design. They can also be applied to protect existing designs, which give IP providers the option of releasing the (protected) source code for their hard/firm IPs that are already in the IP market to make them more competitive.

We propose three watermarking techniques to protect Verilog source code. The first method takes advantage of the don't-care conditions inherently existing in the modules by enforcing them to have specific values corresponding to designer's signature. A separate test module can be easily constructed to retrieve such information. The second one utilizes the fact that many logic units can be implemented in different ways. Instead of using one fixed structure, we build multiple functionally identical modules with different implementations in the same Verilog code. We then selectively instantiate these duplicated modules for information hiding. The third technique splits the implementation of one module into two phases in such a way that designer's signature will be mixed with the module's input and output information. We implement and test the proposed protection schemes on SCU-RTL and ISCAS benchmark circuits using Synopsys' design analyzer and Xilinx FPGA CAD tool. The results show that our watermark survives the synthesis and optimization tools. We measure the area/resources, delay, and power of the designs before and after watermarking, and find that our methods introduce little overhead in these three key aspects.

2 Previous Work

HDL codes describe VLSI design IPs in the style and structure similar to general C/C++ programs. Hence, it is natural to investigate whether the existing design IP protection techniques and software watermarking and obfuscating methods can be extended for HDL code protection.

2.1 VLSI Design IP Protections

According to the IP protection white paper released by VSIA, there are three approaches to secure an IP: **deterrent approach** like patents, copyrights, and trade secrets; **protection** via licensing agreements or encryption; **detection mechanism** such as physical tagging, digital watermarking and fingerprinting [23]. Legal enforcement (copyright, licensing agreement, etc.) can be used to protect HDL codes. But it is always hard to enforce such protection, particularly for the flexible soft IPs. Encryption can be used for soft IP protection [21,22]. But it makes IP reuse inconvenient and there are security holes from which the un-encrypted IP information may leak. Recently, Kahng et al. [8] established principles for constraint-based watermarking techniques in the protection of VLSI design IPs [16,17].

The protection is achieved by tracing unauthorized reuse and making untraceable unauthorized reuse as difficult as re-designing the IP from scratch. The essence of their approach is to introduce watermark-related additional constraints into the input of a black-box design tool such that the design will be rather unique and the embedded watermark can be revealed as proof of authorship. This approach is generic and has been applied to various stages of the VLSI design process, from behavioral and logic synthesis to standard cell place and route algorithms, to FPGA designs [7,8,9,10,11,14].

It is possible, but never easy, to extend the idea of constraint-based watermarking directly into the context of HDL code protection. RT-level HDL source codes normally describe a design in a program-like manner. The constraints are the abstract description of the system's functionality. One can introduce new constraints as watermark. However, any additional constraint at the top abstract level description usually can be easily identified and thus removed or modified. Another concern is the design overhead incurred by adding constraints at this level. If we add constraints at such early stage, it may have large impact to the design quality.

2.2 Software Watermarking and Obfuscating

Watermarking, tamper proofing, and obfuscating are the typical source code protection methods to prevent software piracy [3,2,6,15,19,20]. Watermarking is a technique that embeds a secret message into the program to discourage IP theft by enabling the establishment of IP ownership [5,12]. Tamper-proofing technique protects software from being tampered by making the software with

any unauthorized modification into a nonfunctional code. Obfuscating method makes the program "unintelligible" while preserving its correct functionality.

Obfuscating and tamper-proofing techniques are not suitable for HDL code protection. First, they make programs less readable and harder (if not impossible) to modify, which are all against the incentive to release soft IPs for better design reuse. Secondly, the continuous push for HDL design standards reduces the power of such protections.

Software watermarking methods embed a structure or a function into the program such that it can be reliably located and extracted even after the program has been translated, optimized, and obfuscated. Existing software watermarking schemes are either static or dynamic [4]. Static schemes embed watermark only in the executable and are vulnerable to many attacks. HDL program does not have any executables, so this approach cannot be applied. Dynamic watermark is constructed at run time and stored in dynamic state of the program. The quality of a watermarking scheme depends on how well it stands up to different types of attacks and how successfully the watermark can be retrieved.

To sum up, HDL source codes are soft IPs in the form of program. They have more reuse value than hard/firm IPs because of their flexibility and easy accessibility. However, existing hard/firm IP protection techniques cannot be directly used to prevent designers from losing control of their IPs once HDL source codes are released. On the other hand, HDL code is different from other programming languages like C/C++ and Java [1,13,18]. Current software protection is not applicable for HDL code protection due to the following two reasons: 1) design reuse methodology requires HDL code to be developed and documented following industrial standards; 2) there are no executables associated with HDL programs.

3 HDL Code Watermarking Techniques

3.1 Goals and Assumptions

The sole objective of HDL code watermarking is to hide designer's digital information into the HDL source code for ownership protection. However, a good HDL watermarking technique (at RT-level) must also meet the following goals: (1) Strong proof of authorship. (2) Low design overhead. (3) Survivability from re-synthesis. (4) Resilience. (5) Preserve IP's I/O interface.

To achieve the above goals, we make two necessary assumptions:

Documentation Assumption: designer must document the HDL modules properly and give sufficiently detailed information on each reusable module's input, output, and functionality. However, other details on how each module is implemented are not required.

This assumption has been widely accepted in the HDL design community. It is critical for HDL code watermarking. Without this assumption, designers will be forced to document everything including their watermark implementation. This makes watermark visible and further increases the difficulty of watermarking.

Verification Assumption: all HDL design should follow the hierarchical modular fashion and complicated gate-level HDL code should not be mixed with RT-level description.

We focus on the protection of soft IP at HDL source code level and consequently we restrict the watermark verification problem to be within the context of HDL code as well. The verification assumption prohibits the following attack: obtain the gate-level HDL codes for certain modules from the netlist and use them to replace their equivalent modules. We mention that our proposed techniques are robust against the attack of replacing a single RT-level module by its equivalent gate-level code. However, if the attacker constructs a new module by flattening and combining several modules in gate-level code, then the problem of identifying a watermark is equivalent to sub-circuit isomorphism which is NP-hard. The verification assumption requires hierarchical modular design and discourages designs with only a single module. This is not vital to our approach, but rather a common practice in large real life designs. Attackers can verify the functionality of each module, but they cannot afford to extract all the modules from the HDL source code and re-synthesize them and their combinations. In fact, such attempt is more expensive than redesign from the top-level description given in the documentation.

Next, we will use Verilog as the framework to illustrate three HDL code watermarking approaches. We mention that since Verilog share many common features as hierarchical modular design fashion, code documentation and reused-based design with other HDL languages, VHDL for example, we can easily extend the proposed watermarking techniques for the protection of general HDL code.

3.2 Verilog Watermarking Approaches

Module Watermarking: In this method, we extend the concept of constraint-based watermarking to the Verilog design of a module. A module takes certain input signals and produces output based on the logical functionality to be implemented. The input-output relationship, known as truth table, constrains the module's implementation. To embed additional constraints, we take advantage of the don't care conditions inherently existing in the module. Consider the design of an encoder that converts radix-4 numbers into binary. The useful inputs for this module are 0001, 0010, 0100 and 1000, which produce outputs 00, 01, 10 and 11 respectively. The other twelve combinations of the 4-bit input are don't care conditions as they will not occur in the circuit. Now we show how to embed into the design of this encoder a 15-bit stream $b_{14}b_{13}\ldots b_1b_0$=100010010000010 ('DA' in binary with the last bit as the parity bit). First we order the 12 don't cares in ascending order and make a cyclic list: 0000,0011,0101,,1111. Then we repetitively pick don't cares one- by-one and assign them specific output values to embed the above bit-stream following the algorithm in Figure 1.

More specifically, we take 3, the value of $\lfloor log_2 12 \rfloor$, bits $b_2b_1b_0 = 010$ from the given bit-stream (line 4 and 5). This gives us the binary 2 and we thus select the third (0-offset) don't care 0101 from the list of don't cares. Next we assign a specific value, 00=b_4b_3, to this input (line 7) and delete it from the list

of don't cares (line 10). Now there are 11 don't cares left and we restart from the top don't care 0110 which is the one after 0101. We repeat this process and assign don't care 1011 output 00=b_9b_8 and 1101 output 10=$b_{14}b_{13}$, where the two don't cares are selected based on $b_7b_6b_5$ and $b_{12}b_{11}b_{10}$. As a result, we implement this encoder based on the watermarked truth table (Figure 2(b)) instead of the original truth table (Figure 2(a)).

Input: cyclic list of n don't cares L, number of output bits m, and bit-stream b_i to be embedded.
Output: list of selected don't cares and their assigned output W.

Algorithm:

1. i = 0; // start with the last bit b0 of the bit-stream
2. $j = 0$; //start with the top don't care in list L
3. do
4. $s = \lfloor log_2 n \rfloor$; //bits to pick don't care
5. $(d)_{10} = (b_{s+i-1} \ldots b_{i+1}b_i)_2$;
6. $i = s + i$; // update the position in the bit-stream
7. add {the (d+j) (mod n) don't care, its assigned output $b_{m+i-1} \ldots b_{i+1}b_i$} to the output list W;
8. i = m + i;
9. j = (d+j+1) mod n;//update the top don't care in list L
10. delete the (d+j) (mod n) don't care from L;
11. n = n - 1; // delete the selected don't care from L
12. while (bit-stream embedding not done)

Fig. 1. Pseudo-code for module watermarking with don't cares.

(a)

INPUT	OUT
1000	00
0100	01
0010	10
0001	11

(b)

INPUT	OUT
1000	00
0100	01
0010	10
0001	11
0110	00
1011	00
0000	10

Fig. 2. (a) Original truth table. (b) Watermarked truth table.

To retrieve the bit-stream, we can conveniently write test module forcing these selected don't cares to be the inputs of the watermarked module. We then use Verilog simulation tools to re-establish the bit-stream from the mapping between the don't cares and their corresponding outputs.

Now we briefly analyze this technique. First, watermark's strength, P_c, can be defined as the probability that a random truth table implementation for the original assigns same values to our selected don't cares. Small P_c indicates strong watermark. Let n be the total number of don't cares in the module, k be the number of don't cares that we choose to embed our watermark and m be the number of output bits. P_c can be roughly estimated as:

$$P_c = \frac{k!.(n-k)!}{n}.(\frac{1}{2})^{m.k} \quad (1)$$

$$k = |b|/(\lfloor log_2 n \rfloor + m) \quad (2)$$

However, to embed b bits information, we must choose at least don't care conditions and give them specific values accordingly. This may introduce design overhead. For example, the original encoder in Figure 2(a) can be implemented by two OR gates and four literals, but the watermarked one needs one NOR gate, one OR gate, one AND gate, and a total of five literals. To reduce this overhead, we break long watermark into multiple short ones and embed them into different modules.

Finally, we mention that this method is robust and it is unique for circuit design protection. To remove or alter the watermark, one need to change the values that we have assigned to the selective don't cares. But, in the final circuit implementation of the design, every input pattern will produce a deterministic output. One cannot distinguish whether this output is the original requirement, or comes from the watermark, or simply a value assigned to a don't care condition during the logic synthesis and minimization.

Module Duplication: Despite the possible overhead, the module watermarking method can be easily implemented before the start of Verilog coding. However, it cannot be applied to protect an existing module in Verilog unless we know all original don't cares and redesign the module. The second approach avoids this problem by duplicating some modules and selectively instantiating either the original module or one of its 'duplicates' to hide information.

In Verilog code, there usually exist basic functional modules that are instantiated multiple times by the top-level module or other modules in a higher hierarchy. The functionality of these modules normally can be implemented in different ways. We thus build duplicates for these modules with a 'different look'. That is, they all perform the same function as the original one but synthesis tools will not identify them as identical copies and therefore they will stay in the final design. The simplest way is by assigning different values to don't cares in the original module every time we duplicate it. In this way, the synthesis tool will not delete the duplicates to optimize the design. In the absence of don't cares, normally we can find alternative implementation for the module. Consider an '1101' pattern detector example which receives one bit input data during each clock cycle. It sets output to be '1' whenever it detects a consecutive inputs pattern '1101'; otherwise, the output is always '0'. This module is implemented in Verilog in two different ways. Module detector_0 in Figure 3(a) uses finite state

machine while module detector_1 in Figure 3(b) uses a shift register. However, these two modules are functionally equivalent.

Suppose the detector module has been instantiated multiple times in another module P. Now, with the presence of duplicates, we will have option of which module to instantiate. Instead of instantiating one randomly, we can embed information behind the selection. For example, one scheme is to pick the original Verilog module detector_0 for bit '0' and instantiate module detector_1 for a bit '1'. This is shown in Figure 4 below:

Verifying the signature involves a simple equivalence checking of the module and its duplicates, as well as the evidence of their instantiations. This method provides a strong protection for the Verilog IP as it is highly unusual for an optimized design to have two or more modules that are functionality equivalent. The implementation challenge of this method, however, is how to disguise the duplicates to survive from synthesis. In practice, we have discovered a set of tricks to successfully fool the synthesis tools. One of them, for example, is to assign different values to the same don't care for the original module and its duplicate. The method is robust against attackers who attempt to remove the duplicates or change the module instantiations. Attackers face the (sub-) circuit verification problem and they need to detect the duplicates that have already survived the synthesis optimization tools! Frequent module instantiations in large hierarchical Verilog designs not only provide us a large space for signature embedding, but also make the watermark resilient as we can duplicate modules throughout the design. Another advantage is that it is applicable to existing Verilog codes.

Module Splitting: This approach is usually applied to fairly large modules. It basically splits a large module into several smaller modules. As shown in Figure 5, we use two modules: $\mathcal{A}(X_1, Y_1, Z_1)$ and $\mathcal{B}(X_2, Y_2, Z_2)$, to implement a single-module $\mathcal{M}(X, Y, Z)$, where X is the set of inputs, Y is the set of outputs and Z are optional test outputs. Module splitting is performed as follows:

- First, the watermarking module $\mathcal{A}$ takes input $X_1 \subset X$ and produces 1) part of the functional outputs in $Y_1 \subset Y$, 2) part of the optional test outputs in $Z_1 \subset Z$, and 3) the intermediate watermarking outputs W. W is defined according to our signature on specific input pattern of X_1.
- Next, the correction module $\mathcal{B}$ takes inputs $X_2 \subset W \cup X$ and produces the rest of the required outputs in Y_2 and Z_2. That is, $Y_2 = Y - Y_1$ and $Z_2 = Z - Z_1$.

The above module splitting method is functionally correct because the two modules $\mathcal{A}$ and $\mathcal{B}$ combined to generate signals Y and Z, same as the signals generated by $\mathcal{M}$. To verify the signature, one only needs to feed module $\mathcal{A}$ the input pattern that we define our watermarking signal W, which will be observable from $\mathcal{A}$'s output. To make the watermark robust against both synthesis tools and attackers, we use watermarking signal W from A and as few as possible inputs from X as input for module $\mathcal{B}$. In such way, the watermarking signal W becomes part of the design. Otherwise, they will most likely be removed immediately by

```
module detector_0 (clk, reset, dataIn, out);
      input clk, reset, dataIn;
      output out;
      reg       out;
      reg      [1:0] currentState, nextState;
      always @(dataIn or currentState) begin
            case (currentState)
             2'b00: begin
                           nextState = (dataIn == 1) ? 2'b01 : 2'b00;
                           out = 0;end
             2'b01: begin
                           nextState = (dataIn == 1) ? 2'b10 : 2'b00;
                           out = 0;end
             2'b10: begin
                           nextState = (dataIn == 0) ? 2'b11 : 2'b10;
                           out = 0;end
             2'b11: begin
                           nextState = 2'b00;
                           out = (dataIn == 1);end
            endcase
      end
      always@(posedge clk) begin
        if(~reset) begin
            currentState <= 2'b00;
            out <= 0;
        end
        else currentState <= nextState;
      end
endmodule
```

(a) FSM implementation

```
module detector_1 (clk,reset,dataIn,out);
        input dataIn,clk,reset;
        output out;
        reg out;
        reg [3:0] pattern;
        always@(posedge clk) begin
              if( reset) begin
                    pattern = 0;
                    out = 0;end
              else begin
                    pattern[0]=pattern[1];
                    pattern[1]=pattern[2];
                    pattern[2]=pattern[3];
                    pattern[3]=dataIn;
                    if(pattern==4'b1101) out=1;
                    else out=0;
               end
        end
endmodule
```

(b) Shift register implementation

Fig. 3. '1101' Patern Detector

```
module P;
        reg clk, reset;
        reg data1,data2,data3;
        wire out1, out2, out3;
        detector_0 d1(clk, reset, data1, out1);// signature bit 0
        detector_1 d2(clk, reset, data2, out2);// signature bit 1
        detector_0 d3(clk, reset, data3, out3);// signature bit 0
        ...          ...          ...
endmodule
```

Fig. 4. Instantiation of module detector

optimization tools. The strength of the watermark relies on the rarity of implementing module $\mathcal{M}$ by constraining the intermediate signal W. Although it is secure as watermark is integrated into the design, we mention that this method may considerably increase design complexity particularly for the second module and will be vulnerable to attacks if the verification assumption is not made.

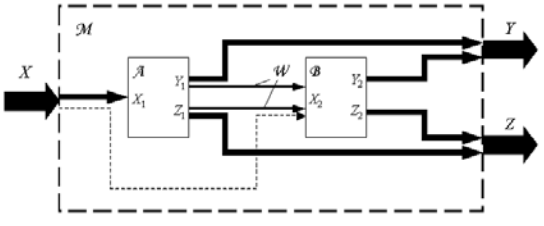


Fig. 5. The idea of module splitting

4 Experimental Results

We apply the proposed techniques to benchmark Verilog circuits and demonstrate that they meet the watermarking objectives. Verilog designs include circuits such as controllers, adders, multipliers, comparators, DSP cores, ALUs (from SCU-RTL and ISCAS benchmarks [26,27], and a MP3 decoder [25]. The MP3 design and SCU-RTL benchmarks [28] are original designs while the RT-level ISCAS Verilog codes are obtained from netlists by reverse engineering [25]. The SCU-RTL and the MP3 benchmarks are perfect examples for the module duplication technique because of the multiple single module instantiations or function calls. The first module watermarking method can also be applied to these designs if the original detailed functional description of each module is available. However, they are not good for the module splitting method because the modules are small. For the ISCAS benchmarks, because they are reversed engineered, we cannot identify the original don't cares and they have only a few

modules, almost all of which are instantiated only once. Consequently, both module watermarking and duplication are not applicable for these benchmarks. But we can use module splitting technique to protect these moderate sized modules with known functionality. We are currently developing a set of Verilog designs to test the first module watermarking method, which we are unable to test over these two existing benchmarks due to the unavailability of their original detailed design specifications. We optimize each original design by Synopsys' design analyzer and then map them to the CLASS library. After that, we collect the following design metrics: area, power, and delay through the design analyzer report. Next, we apply the proposed Verilog watermarking techniques to these designs and repeat the above design process for the watermarked design. As we have described, SCU-RTL benchmark is watermarked by module duplication, and ISCAS circuits by module splitting.

After optimization, we can clearly identify from the schematic view in the Synopsys design analyzer window, both the duplicated modules in the module duplication method and the watermark module in the module splitting method. This insures that our watermarks survive the synthesis tools. Figure 6 gives the gate-level views of ISCAS 74181 circuit (a 4-bit ALU) before and after watermarking, where a 9-letter message (corresponding to author's affiliation, hidden for anonymous review) in ASCII is embedded by splitting the CLA module, which has 3 inputs and 4 outputs, into two modules. We document these two modules in the same way as other original modules. To test the watermark's resilience at both the Verilog code level and the gate level, we showed the watermarked Verilog codes with documentation to a group of our colleagues together with Figure6. None of them could tell which one was the original.

Benchmark Circuits		Original	Watermarked	Overhead
FIR (2264 gates, 16 bits embedded)	Area (λ^2)	4083	4557	11.6 %
	Power (μ W)	34.49	35.33	2.4 %
	Delay (ns)	48.7	48.7	0 %
IIR (15790 gates, 15 bits embedded)	Area (λ^2)	16419	16431	0.07 %
	Power (μ W)	35.33	35.06	-0.76 %
	Delay (ns)	49.15	49.15	0 %
IDCT (17341 gates, 16 bits embedded)	Area (λ^2)	20755	21271	2.5 %
	Power (μ W)	23.31	23.5	0.8 %
	Delay (ns)	49.2	49.2	0 %
MP3 (>20000 gates, 20 bits embedded)	Area (λ^2)	16955	17297	4.9 %
	Power (μ W)	67.49	70.82	2.0 %
	Delay (ns)	49.15	49.15	0 %

Table 1. Watermarking SCU-RTL & MP3 Verilog benchmark circuit.

Table 1 reports the design overhead on SCU-RTL benchmarks by module duplication. As expected, there is little area overhead due to the duplicated modules. However, the average area overhead is about 4.76% (and this percentage is mainly caused by the small FIR Design). The watermarked design does

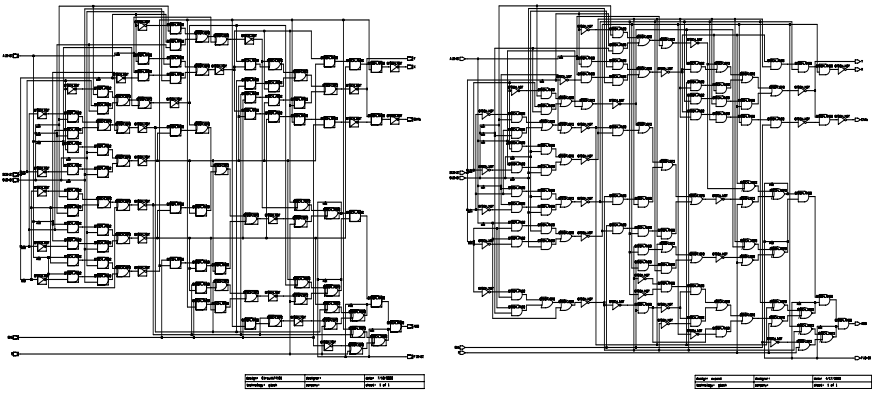
not introduce any additional delay and consumes only 1% more energy on an average than the original design.

Benchmark Circuits		Original	Watermarked	Overhead
74181 (61 gates, 56 bits embedded)	Area (λ^2)	86	94	9.3 %
	Power (μ W)	102.41	111.84	9.2 %
	Delay (ns)	9.26	10.38	12.1 %
C432 (160 gates, 56 bits embedded)	Area (λ^2)	176	192	9.1 %
	Power (μ W)	230.87	249.89	8.2 %
	Delay (ns)	20.44	19.63	-4.0%
C499 (202 gates, 56 bits embedded)	Area (λ^2)	400	410	2.5 %
	Power (μ W)	14.75	11.71	2.5 %
	Delay (ns)	14.75	11.71	-20.6 %
c1908 (880 gates, 56 bits embedded)	Area (λ^2)	574	598	4.1 %
	Power (μ W)	581.43	612.47	5.3 %
	Delay (ns)	21.82	22.54	3.3 %
C7552 (61 gates, 56 bits embedded)	Area (λ^2)	4489	4525	0.8 %
	Power (μ W)	5778.1	5808.5	0.5 %
	Delay (ns)	65.57	65.57	0 %

Table 2. Watermarking on ISCAS benchmark circuits.

Table 2 reports the results of watermarking ISCAS benchmarks by module splitting. In this technique, we enforce the watermark into design's functionality. In general, this should cause design overhead. For example, we see that both average area and power overhead are slightly over 5 %. Interestingly, the circuit delay may decrease after watermarking. This might be possible, for example, if we split a module that has a signal on the critical path, this signal may be generated by the simpler watermarking module and thus reduce the delay. From tables 1 and 2, we can see that large design overhead often occurs for small designs (FIR, 74181, and C432). Although it is premature to claim, given the limited set of experiments, we anticipate that all design overhead will decrease for large designs and eventually become negligible for real life designs.

FPGA designs occupy a significant part of the integrated circuit market these days, and our watermarking techniques are easily applicable to them as well. We implement the original and watermarked Verilog benchmarks on certain Xilinx Virtex-II devices. Most of the Verilog code written for ASIC implementation can be synthesized by FPGA synthesis tools with little or no modifications. Some Verilog code had some technology dependent features, such as instantiating some gates from a particular library that could not be mapped to the FPGA devices using our FPGA synthesis tool. Therefore a subset of all the designs was chosen and synthesized using the Xilinx ISE5.1. (This is the reason why in Table 3, the gate counts for the same benchmark increase.) The synthesizable Verilog benchmarks include: the IIR circuit from the SCU-RTL benchmark suite and 74181, C432 and C499 circuits from the ISCAS benchmark suite. These designs are mapped to Xilinx Virtex-II devices and the implementation results show

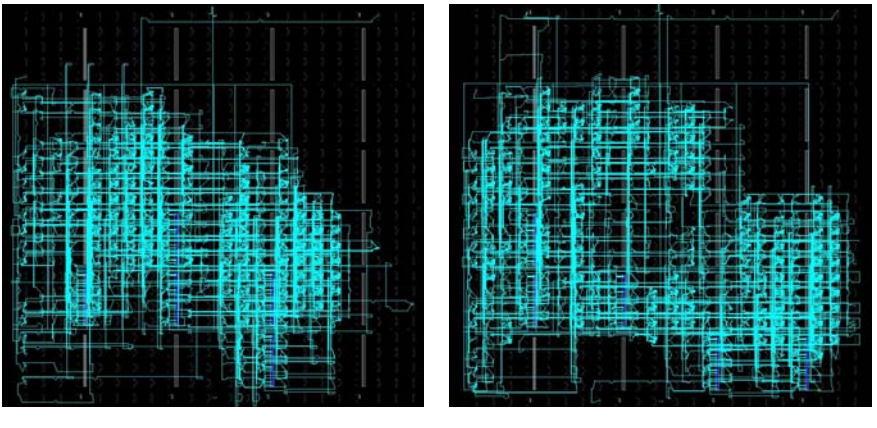


(a) Original (b) Watermarked

Fig. 6. Gate-level view of circuit 74181

that the embedded watermarks survive synthesis and optimization performed by the design tool. Figure 7 displays the fully placed and routed original and watermarked IIR design on the Xilinx Virtex II FPGA. We have embedded a 15-bit signature in the watermarked Verilog source code. However, it is not easy to locate the watermark by reverse engineering at the chip-level as there is no information available at that level, specific to the watermark.

In FPGAs, the two main design criteria are speed and resource utilization in terms of the number of slices and LUTs used. Table 3 reports the Maximum combinational path delay and resource utilization in both original and watermarked designs generated by the Xilinx tool.



(a) Original (b) Watermarked

Fig. 7. Floor plan view of IIR targeted to Xilinx Virtex-II.

Benchmark Circuits		Original	Watermarked	Overhead
IIR (27572 gates,	♯ Slices	286	329	18.71 %
15 bits embedded	Max. Path Delay (ns)	11.1	11.1	0 %
C432 (420 gates,	♯Slices	40	44	10.1 %
56 bits embedded	Max. Path Delay (ns)	29.791	29.137	-2.2 %
C499 (696 gates,	♯Slices	67	74	10.45 %
56 bits embedded	Max. Path Delay (ns)	16.326	17.908	9.69 %
74181 (132 gates,	♯Slices	13	13	0 %
56 bits embedded	Max. Path Delay (ns)	13.71	13.71	0 %

Table 3. Benchmarks targeted to Xilinx Virtex-II FPGA.

5 Conclusions

We propose the first set of non-traditional protection mechanisms for soft IPs (HDL codes). These codes describe circuits at the software level and therefore their protection has different requirements and challenges, from those for hard/firm VLSI IP or software protection. We use Verilog as the framework and leverage Verilog's unique role between hardware and software to embed the watermark message into the source code for protection. We evaluate the strength, resilience, and design overhead of these watermarking techniques both analytically and by simulation over benchmark Verilog circuits available in the public domain. We demonstrate the applicability of these techniques for FPGA and ASIC designs and evaluate the overhead. The proposed techniques can be used to protect both new and existing Verilog designs as well as VHDL designs. We are currently collecting and building more Verilog and VHDL circuits to test our approach. We are also planning to develop CAD tools for HDL protection.

References

1. G. Arboit, "A Method for Watermarking Java Programs via Opaque Predicates (Extended Abstract)", The Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002.
2. C. Collberg and G. Myles and A. Huntwork, "SandMark — A Tool for Software Protection Research", IEEE Magazine of Security and Privacy, vol. 1, aug, 2003.
3. C.S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation – tools for software protection", IEEE Transactions on Software Engineering, Vol. 8, **8**, 2002.
4. C.S. Collberg, C. Thomborson, "Software Watermarking Models and Dynamic Embeddings," ACM Symposium on Principles of Programming Languages, Jan 1999.
5. P. Cousot and R. Cousot, "An Abstract Interpretation-Based Framework for Software Watermarking", ACM Principles of Programming Languages, 2004.
6. R.L. Davidson and N. Myhrvold, "Method and System for Generating and Auditing a Signature for a Computer Program", US Patent 5,559,884, Assignee: Microsoft Corporation,1996.

7. I. Hong and M. Potkonjak. "Behavioral Synthesis Techniques for Intellectual Property Protection", 36th ACM/IEEE Design Automation Conference Proceedings, pp. 849-854, June 1999.
8. A.B. Kahng, et al.. "Watermarking Techniques for Intellectual Property Protection", 35th ACM/IEEE Design Automation Conference Proceedings, pp. 776-781, June 1998.
9. D. Kirovski, Y. Hwang, M. Potkonjak, and J. Cong. "Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions," IEEE/ACM International Conference on Computer Aided Design, pp. 194-198, November 1998.
10. M. Keating and P. Bricaud. "Reuse Methodology Manual, For System-On-A-Chip Designs," Second Edition, 1999.
11. J. Lach, W.H. Mangione-Smith, and M. Potkonjak. "FPGA Fingerprinting Techniques for Protecting Intellectual Property," Proceedings of the IEEE 1998 Custom Integrated Circuits Conference, pp. 299-302, May 1998.
12. G. Myles and C. Collberg, "Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks", International Conference on Information Security and Cryptology, 2003.
13. A. Monden and H. Iida and K. Matsumoto and K. Inoue and K. Torii, "A practical method for watermarking Java programs", 24th Computer Software and Applications Conference, 2000.
14. A.L. Oliveira. "Robust Techniques for Watermarking Sequential Circuit Designs," 36th ACM/IEEE Design Automation Conference Proceedings, pp. 837-842, June 1999.
15. J. Palsberg and S. Krishnaswamy and M. Kwon and D. Ma and Q. Shao and Y. Zhang, " Experience with Software Watermarking", Proceedings of ACSAC'00, 16th Annual Computer Security Applications Conference, pp. 308-316, 2000.
16. G. Qu and M. Potkonjak, "Analysis of watermarking techniques for graph coloring problem", Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design, pp. 190–193, 1998.
17. G. Qu and M. Potkonjak, "Fingerprinting intellectual property using constraint-addition", Design Automation Conference, pp. 587-592, 2000.
18. J. P. Stern and G. Hachez and F. Koeune and Jean-Jacques Quisquater, "Robust Object watermarking: Application to code", Information Hiding, pp. 368-378, 1999.
19. R. Venkatesan and V. Vazirani and S. Sinha, "A Graph Theoretic Approach to Software Watermarking", 4th International Information Hiding Workshop, Pittsburgh, PA, april, 2001.
20. R. Venkatesan and V. Vazirani, "A Technique For Producing, Through Watermarking, Highly Tamper-Resistant Executable Code And Resulting "Watermarked" Code So Formed", International Patent WO 01/69355 A1, 2001.
21. Altera Corporation. San Jose, California. http://www.altera.com/
22. Xilinx Inc. San Jose, California. http://www.xilinx.com
23. Virtual Socket Interface Alliance. "Intellectual Property Protection White Paper: Schemes, Alternatives and Discussion Version 1.0," September 2000.
24. Virtual Socket Interface Alliance. "Architecture Document Version 1.0," March 1997.
25. http://www.ece.cmu.edu/∼ee545/s02/10/fpga_1813.txt
26. http://www.eecs.umich.edu/∼jhayes/iscas.
27. International Technology Roadmap for Semiconductors. http://public.itrs.net/Files/2001ITRS/
28. http://www.engr.scu.edu/mourad/benchmark/RTL-Bench.html