2010 23rd International Conference on VLSI Design

RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation

Rajat Subhra Chakraborty and Swarup Bhunia
Dept. of EECS, Case Western Reserve University
Cleveland, OH-44106, USA
{rsc22, skb21}@case.edu

Abstract

Recent trends of hardware intellectual property (IP) piracy and reverse engineering pose major business and security concerns to an IP-based system-on-chip (SoC) design flow. In this paper, we propose a Register Transfer Level (RTL) hardware IP protection technique based on low-overhead key-based obfuscation of control and data flow. The basic idea is to transform the RTL core into control and data flow graph (CDFG) and then integrate a well-obfuscated finite state machine (FSM) of special structure, referred as "Mode-Control FSM", into the CDFG in a manner that normal functional behavior is enabled only after application of a specific input sequence. We provide formal analysis of the effectiveness of the proposed approach and present a simple metric to quantify the level of obfuscation. We also present an integrated design flow that implements the proposed obfuscation at low computational overhead. Simulation results for two open-source IP cores show that high levels of security is achievable at nominal area and power overheads under delay constraint.

1. Introduction

Hardware intellectual property (IP) cores form an integral part of modern multi-million gate SoC designs. However, recent trends in IP-piracy and reverse-engineering efforts have raised serious concerns in the IP vendor community [1,2]. A major fraction of commercial IPs come in the form of synthesizable RTL descriptions in Hardware Description Languages (HDLs), due to their better portability. Protection of these *soft IPs* from piracy and reverse-engineering, is extremely challenging due to their better clarity and intelligibility.

Hardware IP protection has been investigated earlier in diverse contexts. Previous work on this topic can be broadly classified into two main categories: (1) *Authentication* based and (2) *Obfuscation* based. Efficient insertion of hard-to-remove "digital watermark" or *authentication signature* in IPs has been widely investigated [1]. The watermark is usually one or more special input-output response pairs, which do not arise during the normal course of operation. The watermarking approaches, however, are effective as *passive* defence measures, which only help to prove the ownership of the IP in case of litigation without preventing the piracy from happening in the first place.

In *obfuscation* based IP protection, the IP vendor usually affects the human readability of the HDL code [6,9], or relies on cryptographic techniques to encrypt the source code [7], such that the resultant code becomes significantly more difficult to reverse-engineer. However, the above techniques do not modify the functionality of an IP core and thus often cannot prevent it from being stolen by a hacker and used as a "black-box" RTL module. In [7,8], the HDL source code is encrypted and the IP vendor provides the key to de-crypt the code to only its legitimate customers. However, each of the above techniques forces the use of a particular design platform, a situation that may not be acceptable to many SoC designers who seek the flexibility of multiple tools from diverse vendors in the design flow. In [2], we proposed an approach to obfuscate the functionality of gate-level *firm* IP cores through modification of the state transition graph (STG) that requires a specific input sequence to enable normal functional behavior. Such an obfuscation approach provides an active defence to IP infringement. However, extension of this approach to RTL involves major challenges, primarily due to the difficulty of obfuscating the STG modification in a highly comprehensible RTL code. In [3], we proposed a decompilation-based RTL obfuscation approach, which requires converting the RTL design into gate-level netlist. However, decompilation can remove some preferred RTL constructs and hence can be undesirable at some cases. Software obfuscation to prevent reverse-engineering has also been studied extensively [4,5]. However, software approaches cannot be directly applicable to hardware IPs since software obfuscation is per-

1063-9667/10 $26.00 © 2010 IEEE
DOI 10.1109/VLSI.Design.2010.54

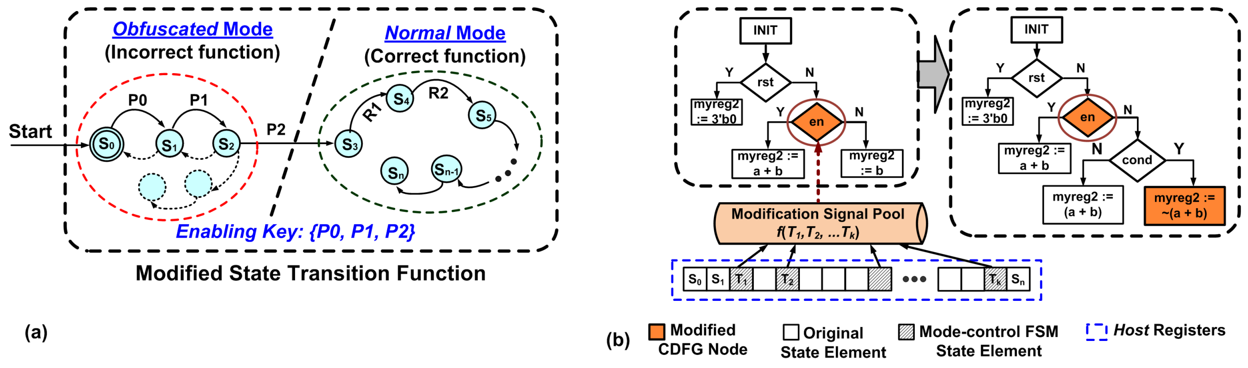


Figure 1. The general functional obfuscation scheme by state transition function modification [2]: a) modified state transition function; and b) change in internal node structure using modification cells.

formed with very different constraints (e.g. code size, run time as opposed to power and performance in hardware) and it typically cannot prevent black-box usage of a code.

In this paper, we propose a low-overhead methodology for implementing key-based obfuscation in RTL IP core. It does not require converting RTL into gate level netlist. The main idea is to efficiently integrate a *Mode-Control FSM* into the design through judicious modification of control and data flow structures, such that the design works in two modes: 1) *obfuscated mode* and 2) *normal mode*. The design produces undesired output values in obfuscated mode, while normal functional behavior is enabled when it moves to normal mode on application of a pre-defined input sequence. The *Mode-Control FSM* is integrated inside the CDFG of the IP in a way that makes it extremely hard to isolate from the original IP. The FSM is realized in the RTL by expanding a judiciously selected set of registers, which we refer to as *host registers* and modifying their assignment conditions and values. Once the FSM is integrated, both control and data flow statements are conditioned based on the mode control signals derived from this FSM.

We derive a metric that estimates the level of obfuscation in terms of the difficulty of reverse engineering the obfuscated RTL. Effectiveness of the proposed approach is verified with simulation results for two open-source IP cores. The results show that the proposed approach is computationally efficient and provides high level of obfuscation at low design overhead, while preserving the portability of the RTL code. Although we consider Verilog RTL in our analysis and simulation, the technique is applicable to other HDLs and to both SoC and FPGA platforms.

2 Methodology

The general scheme of functional obfuscation [2] is shown in Fig. 1. Here, a mode-control FSM is inserted into a design that forces it to operate in two distinct modes: (a) the *obfuscated mode* when the circuit functionality is drastically different from its prescribed functionality, and (b) the *normal mode* when the circuit operates in its normal mode. On power-up, the circuit is initialized in the *obfuscated mode*, and on the application of a particular sequence at the primary input, which we refer to as the *initialization key sequence*, the mode-control FSM goes through a sequence of state transitions that brings the system to the *normal mode*. This is shown in Fig. 1(a), where the application of the sequence $P0 \rightarrow P1 \rightarrow P2$ takes the circuit to the *normal mode*. The circuit is modified at selected control and data flow nodes using *modification signals* which are formed as Boolean functions of the mode-control FSM states. These *modification signals* affect the logic values at select nodes in the *obfuscated mode*, preventing the circuit to perform its normal functionality. The mode-control FSM performs some *dummy state transitions* in both the *obfuscated mode* and the *normal mode*. These transitions are inserted so that the FSM is not "stuck-at" at a particular state in either mode, thus making it difficult for an adversary to identify the state elements implementing this FSM through structural analysis.

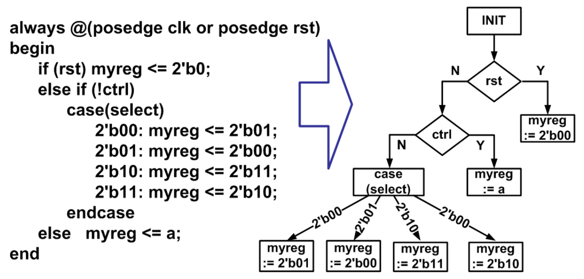

```
always @(posedge clk or posedge rst)
begin
    if (rst) myreg <= 2'b0;
    else if (!ctrl)
        case(select)
            2'b00: myreg <= 2'b01;
            2'b01: myreg <= 2'b00;
            2'b10: myreg <= 2'b11;
            2'b11: myreg <= 2'b10;
        endcase
    else  myreg <= a;
end
```

Figure 2. Transformation of a block of RTL code into CDFG.

The security against piracy and reverse engineering stems from the following features: (a) it is not possible to make the hardware IP functional until the *initialization key sequence*, which also acts as the proof of ownership of the IP, is known and (b) it is extremely difficult to derive the key through functional simulations as described in [2]. The main challenge, however, is to hide the mode-control FSM and design modifications in the RTL. The proposed obfuscation scheme comprises of four major steps described in the rest of the section.

2.1 Parsing the RTL and building CDFG

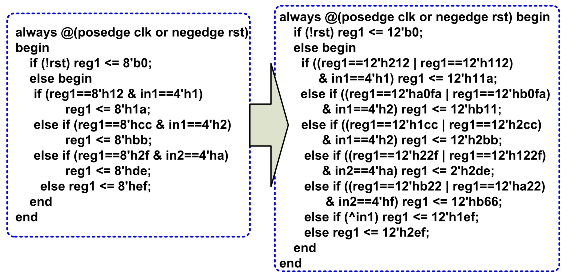
In this step, the given RTL is parsed and concurrent blocks of RTL code are transformed into a CDFG data structure. Fig. 2 shows the transformation of an "always @()" block of a Verilog code to its corresponding CDFG. Next, small CDFGs are merged (whenever possible) to build larger combined CDFGs. For example, all CDFGs corresponding to non-blocking assignments to clocked registers can be combined together without any change of the functionality. This procedure creates larger CDFGs with substantially more number of nodes than the constituent CDFGs, which helps to obfuscate the *hosted* mode-control FSM better.

2.2 "Hosting" the mode-control FSM

Instead of having a stand-alone mode-control FSM as in [2], the state elements of the mode-control FSM can be hosted in existing registers in the design to increase the level of obfuscation. This way, the FSM becomes an integral part of the design, instead of controlling the circuit as a structurally isolated element. An example is shown in Fig. 3, where the 8-bit register *reg1*, referred as the "host register", has been expanded to 12-bits to host the mode-control FSM in its left 4-bits. When these 4-bits are set at values $4'h1$ or $4'h2$, the circuit is in its *normal* mode, while the circuit is in its *obfuscated mode* when they are at $4'ha$ or $4'hb$. Note that extra RTL statements have been added to make the circuit functionally equivalent in the *normal mode*. The obfuscation can be improved by distributing the mode-control FSM state elements in a non-contiguous manner inside one or more registers.

2.3 Modifying CDFG branches

After the FSM has been hosted in a set of selected *host registers*, several CDFG nodes are modified using the control signals generated from this FSM. The nodes with large fanout cones are preferentially selected for modification, since this ensures maximum change in functional behavior at minimal design overhead. Three example modifications of the CDFGs and the corresponding RTL statements



```
always @(posedge clk or negedge rst)
begin
    if (!rst) reg1 <= 8'b0;
    else begin
     if (reg1==8'h12 & in1==4'h1)
            reg1 <= 8'h1a;
     else if (reg1==8'hcc & in1==4'h2)
            reg1 <= 8'hbb;
     else if (reg1==8'h2f & in2==4'ha)
            reg1 <= 8'hde;
      else reg1 <= 8'hef;
    end
end
```

```
always @(posedge clk or negedge rst) begin
    if (!rst) reg1 <= 12'b0;
    else begin
      if ((reg1==12'h212 | reg1==12'h112)
          & in1==4'h1) reg1 <= 12'h11a;
      else if ((reg1==12'ha0fa | reg1==12'hb0fa)
           & in1==4'h2) reg1 <= 12'hb11;
      else if ((reg1==12'h1cc | reg1==12'h2cc)
           & in1==4'h2) reg1 <= 12'h2bb;
       else if ((reg1==12'h22f | reg1==12'h122f)
           & in2==4'ha) reg1 <= 2'h2de;
       else if ((reg1==12'hb22 | reg1==12'ha22)
            & in2==4'hf) reg1 <= 12'hb66;
       else if (^in1) reg1 <= 12'h1ef;
       else reg1 <= 12'h2ef;
    end
end
```

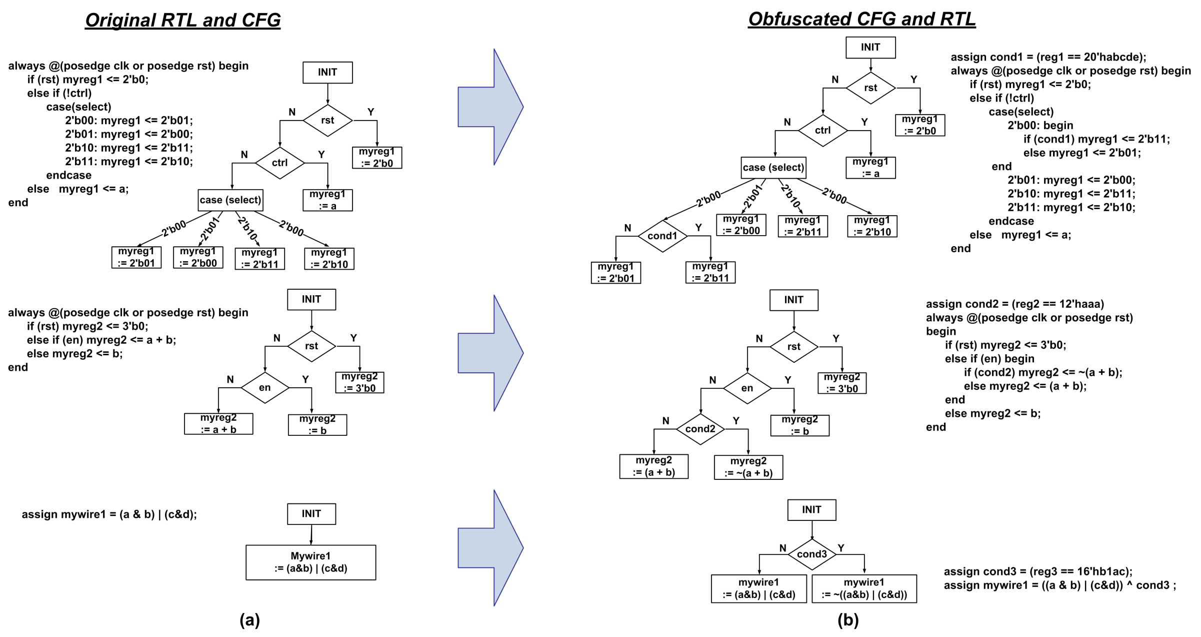
Figure 3. Example of a register hosting part of the mode-control FSM.

are shown in Fig. 4. The registers *reg1*, *reg2* and *reg3* are the *host registers*. Three "case()", "if()" and "assign" statements in Fig. 4(a) are modified by the mode-control signals *cond1*, *cond2* and *cond3*, respectively. These signals evaluate to logic-1 only in the *obfuscation mode* because the conditions *reg1*=$20'habcde$, *reg2*=$12'haaa$ and *reg3*=$16'hb1ac$ correspond to states which are only reachable in the *obfuscation mode*. Fig. 4(b) shows the modified CDFGs and the corresponding CDFG statements.

Besides changing the control-flow of the circuit, functionality is also modified by introducing additional datapath components. However, such changes are done in a manner that ensures sharing of the additional resources during synthesis. This is important since datapath components usually incur large hardware overhead. An example is shown in Fig. 5, where the signal *out* originally computes $(a+b)\times(a-b)$. However, after modification of the RTL, it computes $(a+b)$ in the *obfuscated mode*, allowing the adder to be shared in the two modes and the outputs of the multiplier and the adder to be multiplexed.

2.4 Generating obfuscated RTL

After the modifications have been preformed on the CDFG, the obfuscated RTL is generated from the modified CDFGs, by traversing each of them in a *depth-first* manner. Fig. 6(a) shows an example RTL and Fig. 6(b) shows its obfuscated version. A 4-bit FSM has been *hosted* in registers *int_reg1* and *int_reg2*. The conditions *int_reg1*[13:12]=$2'b00$, *int_reg1*[13:12]=$2'b01$, *int_reg2*[13:12]=$2'b00$ and *int_reg1*[13:12]=$2'b10$ occur only in the obfuscated mode. The initialization sequence is *in1*=$12'h654$ $\rightarrow$ *in2*=$12'h222$ $\rightarrow$ *in1*=$12'h333$ $\rightarrow$ *in2*=$12'hacc$ $\rightarrow$ *in1*=$12'h9ab$. Note the presence of *dummy state transitions* and out-of-order state transition RTL statements. The outputs *res1* and *res2* have been modified by two different *modification signals*. Instead of allowing the inputs to appear directly in the sensitivity list of the "if()" statements, it is possible to derive internal signals (similar to the ones shown in Fig. 4(b)) with complex Boolean ex-



```
always @(posedge clk or posedge rst) begin
    if (rst) myreg1 <= 2'b0;
    else if (!ctrl)
        case(select)
            2'b00: myreg1 <= 2'b01;
            2'b01: myreg1 <= 2'b00;
            2'b10: myreg1 <= 2'b11;
            2'b11: myreg1 <= 2'b10;
        endcase
    else   myreg1 <= a;
end
```

```
always @(posedge clk or posedge rst) begin
    if (rst) myreg2 <= 3'b0;
    else if (en) myreg2 <= a + b;
    else myreg2 <= b;
end
```

```
assign mywire1 = (a & b) | (c&d);
```

```
assign cond1 = (reg1 == 20'habcde);
always @(posedge clk or posedge rst) begin
    if (rst) myreg1 <= 2'b0;
    else if (!ctrl)
        case(select)
            2'b00: begin
                if (cond1) myreg1 <= 2'b11;
                else myreg1 <= 2'b01;
         end
            2'b01: myreg1 <= 2'b00;
            2'b10: myreg1 <= 2'b11;
            2'b11: myreg1 <= 2'b10;
        endcase
    else   myreg1 <= a;
end
```

```
assign cond2 = (reg2 == 12'haaa)
always @(posedge clk or posedge rst)
begin
    if (rst) myreg2 <= 3'b0;
    else if (en) begin
        if (cond2) myreg2 <= ~(a + b);
        else myreg2 <= (a + b);
    end
    else myreg2 <= b;
end
```

```
assign cond3 = (reg3 == 16'hb1ac);
assign mywire1 = ((a & b) | (c&d)) ^ cond3 ;
```

Figure 4. Examples of control-flow obfuscation: (a) original RTL, CDFG; (b) obfuscated RTL, CDFG.

pressions which are used to perform the modifications. The output *res1* has been modified following the datapath modification approach using resource sharing.

3 Obfuscation Efficiency

In this section, we derive a metric to quantify the effectiveness of the proposed obfuscation technique. We then propose a method to improve the obfuscation efficiency under overhead constraint.

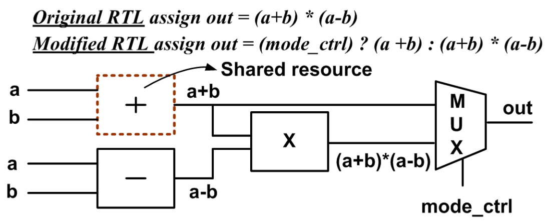
3.1 Measures of obfuscation efficiency

Since obfuscation of RTL IP serves similar objective as software IP obfuscation, we borrow ideas from software obfuscation to estimate the success of the proposed obfuscation scheme. In software engineering, the success of obfuscation is measured by the the following characteristics of the obfuscated code [5]:

- *Potency*: the complexity in comprehending the obfuscated program compared to the unobfuscated one.
- *Resilience*: difficulty faced by an automatic obfuscator in breaking the obfuscation.
- *Stealth*: how well the obfuscated code blends in with the rest of the program, and
- *Cost*: how much computational overhead it adds to the obfuscated program.

In our scheme, the *potency* is estimated by the structural and functional differences between the obfuscated and the original circuit. We estimate the success of the functional/structural modification by the percentage of nodes failing verification when the original and the obfuscated designs are subjected to formal verification. This is because the modifications in the control and data flow of the circuit has a direct impact on the Boolean functionality of the internal nodes in the circuit, which in turn modifies the *Reduced Ordered Binary Decision Diagram* (ROBDD) representation of the logic network representing the nodes [10]. Since formal verification is mostly based on the comparison of the ROBDDs of two corresponding nodes in two designs, a higher percentage of verification failures indicates that the obfuscated design differs substantially from the original design in both functionality and structure.

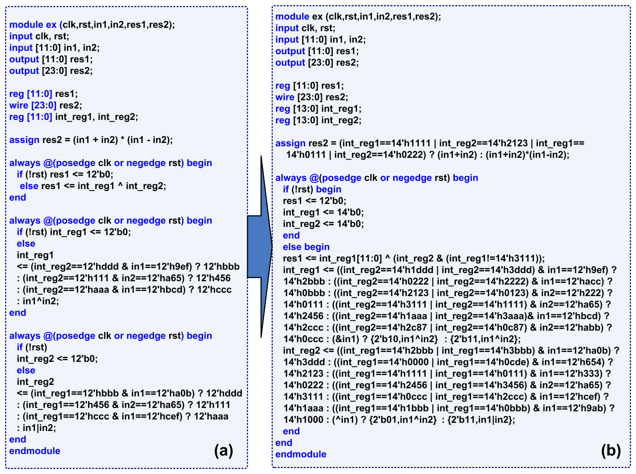
The *resilience* and *stealth* of our obfuscation scheme is estimated by the degree of difficulty faced by a hacker in discovering the *hosted* mode-control FSM and the modification signals. Consider a case where n mode-control FSM



Original RTL *assign out = (a+b) * (a-b)*

Modified RTL *assign out = (mode_ctrl) ? (a +b) : (a+b) * (a-b)*

Figure 5. Example of datapath obfuscation allowing resource sharing.



```
module ex (clk,rst,in1,in2,res1,res2);
input clk, rst;
input [11:0] in1, in2;
output [11:0] res1;
output [23:0] res2;

reg [11:0] res1;
wire [23:0] res2;
reg [11:0] int_reg1, int_reg2;

assign res2 = (in1 + in2) * (in1 - in2);

always @(posedge clk or negedge rst) begin
  if (!rst) res1 <= 12'b0;
   else res1 <= int_reg1 ^ int_reg2;
end

always @(posedge clk or negedge rst) begin
  if (!rst) int_reg1 <= 12'b0;
  else
  int_reg1
  <= (int_reg2==12'hddd & in1==12'h9ef) ? 12'hbbb
  : (int_reg2==12'h111 & in2==12'ha65) ? 12'h456
  : (int_reg2==12'haaa & in1==12'hbcd) ? 12'hccc
  : in1^in2;
end

always @(posedge clk or negedge rst) begin
  if (!rst)
  int_reg2 <= 12'b0;
  else
  int_reg2
  <= (int_reg1==12'hbbb & in1==12'ha0b) ? 12'hddd
  : (int_reg1==12'h456 & in2==12'ha65) ? 12'h111
  : (int_reg1==12'hccc & in1==12'hcef) ? 12'haaa
  : in1|in2;
end
endmodule
```

```
module ex (clk,rst,in1,in2,res1,res2);
input clk, rst;
input [11:0] in1, in2;
output [11:0] res1;
output [23:0] res2;

reg [11:0] res1;
wire [23:0] res2;
reg [13:0] int_reg1;
reg [13:0] int_reg2;

assign res2 = (int_reg1==14'h1111 | int_reg2==14'h2123 | int_reg1==
   14'h0111 | int_reg2==14'h0222) ? (in1+in2) : (in1+in2)*(in1-in2);

always @(posedge clk or negedge rst) begin
  if (!rst) begin
  res1 <= 12'b0;
  int_reg1 <= 14'b0;
  int_reg2 <= 14'b0;
  end
  else begin
  res1 <= int_reg1[11:0] ^ (int_reg2 & (int_reg1!=14'h3111));
  int_reg1 <= ((int_reg2==14'h1ddd | int_reg2==14'h3ddd) & in1==12'h9ef) ?
  14'h2bbb : ((int_reg2==14'h0222 | int_reg2==14'h2222) & in1==12'hacc) ?
  14'h0bbb : ((int_reg2==14'h2123 | int_reg2==14'h0123) & in2==12'h222) ?
  14'h0111 : ((int_reg2==14'h3111 | int_reg2==14'h1111) & in2==12'ha65) ?
  14'h2456 : ((int_reg2==14'h1aaa | int_reg2==14'h3aaa)& in1==12'hbcd) ?
  14'h2ccc : ((int_reg2==14'h2c87 | int_reg2==14'h0c87) & in2==12'habb) ?
  14'h0ccc : (&in1) ? {2'b10,in1^in2}  : {2'b11,in1^in2};
  int_reg2 <= ((int_reg1==14'h2bbb | int_reg1==14'h3bbb) & in1==12'ha0b) ?
  14'h3ddd : ((int_reg1==14'h0000 | int_reg1==14'h0cde) & in1==12'h654) ?
  14'h2123 : ((int_reg1==14'h1111 | int_reg1==14'h0111) & in1==12'h333) ?
  14'h0222 : ((int_reg1==14'h2456 | int_reg1==14'h3456) & in2==12'ha65) ?
  14'h3111 : ((int_reg1==14'h0ccc | int_reg1==14'h2ccc) & in1==12'hcef) ?
  14'h1aaa : ((int_reg1==14'h1bbb | int_reg1==14'h0bbb) & in1==12'h9ab) ?
  14'h1000 : (^in1) ? {2'b01,in1^in2}  : {2'b11,in1|in2};
  end
end
endmodule
```

Figure 6. Example of RTL obfuscation: (a) original RTL; (b) obfuscated RTL.

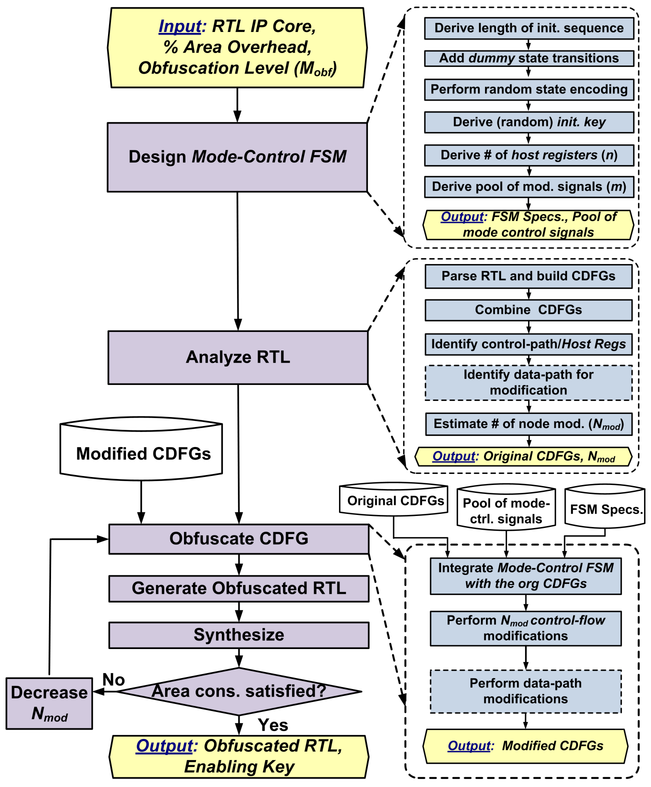

Input: RTL IP Core, % Area Overhead, Obfuscation Level (M_{obf})
Design Mode-Control FSM
Derive length of init. sequence
Add dummy state transitions
Perform random state encoding
Derive (random) init. key
Derive # of host registers (n)
Derive pool of mod. signals (m)
Output: FSM Specs., Pool of mode control signals
Analyze RTL
Parse RTL and build CDFGs
Combine CDFGs
Identify control-path/Host Regs
Identify data-path for modification
Estimate # of node mod. (N_{mod})
Output: Original CDFGs, N_{mod}
Modified CDFGs
Original CDFGs
Pool of mode-ctrl. signals
FSM Specs.
Obfuscate CDFG
Generate Obfuscated RTL
Synthesize
Area cons. satisfied?
No
Yes
Decrease N_{mod}
Output: Obfuscated RTL, Enabling Key
Integrate Mode-Control FSM with the org CDFGs
Perform N_{mod} control-flow modifications
Perform data-path modifications
Output: Modified CDFGs

Figure 7. Steps of the proposed RTL obfuscation methodology.

state-transition statements have been hosted in a RTL with N blocking/non-blocking assignment statements. However, the hacker does not know a-priori how many registers host the mode-control FSM. Then, the hacker must correctly figure out the hosted FSM state transition statements from one out of $\sum_{k=1}^{n}\binom{N}{k}$ possibilities. Again, each of these choices for a given k has $k!$ associated ways to arrange the state transitions (so that the *initialization key sequence* is applied in the correct order). Hence, the hacker must correctly identify one out of $\sum_{k=1}^{n}\left(\binom{N}{k}\cdot k!\right)$ possibilities. The other feature that needs to be deciphered by the hacker are the mode control signals. Let M be the total number of blocking, non-blocking and dataflow assignments in the RTL, and let m be the size of the modification signal pool. Then, the hacker must choose m signals correctly out of M signals, which is one out of $\binom{M}{m}$ choices. Combining these two security features, we propose the following metric to estimate the *resilience* and *stealth* of the obfuscated design:

$$M_{obf} = \frac{1}{\sum_{k=1}^{n}\left(\binom{N}{k}\cdot k!\right)\cdot\binom{M}{m}} \qquad (1)$$

A lower value of M_{obf} indicates a greater obfuscation efficiency. As an example, consider a RTL with values $N = 30$, $M = 100$, in which a FSM with parameter $n = 3$ is hosted, and let $m = 20$. Then, $M_{obf} = 7.39\times10^{-26}$. In other words, the probability of the hacker reverse-engineering the complete scheme is about 1 in 10^{27}. In practice, the values of n and M would be much higher in most cases, making M_{obf} smaller and thus tougher for the hacker to reverse-engineer the obfuscation scheme.

The *cost* of our obfuscation scheme is measured by the increase in compilation time and design overheads of the obfuscated design compared to the original design, as discussed in Section 4.

4 Results

Fig. 7 shows the steps of the proposed design obfuscation methodology. The input to the flow is the original RTL, the desired obfuscation level represented by the obfuscation metric (M_{obf}), and the maximum allowable area overhead. It starts with the design of the mode-control FSM based on the target M_{obf}. The output of this step are the specifications of the FSM which include its state transition graph, the state encoding, the pool of modification signals, and the *initialization key sequence*. Random state encoding and a random *initialization key sequence* are generated to increase the security. The steps described in Section 2 are then performed. The obfuscated RTL is synthesized, and if the area overhead exceeds the overhead constraint, the number of modifications (N_{mod}) is decreased by a pre-defined

Table 1. Obfuscation Efficiency and Design Overhead (at iso-delay) for a Set of Open-source IP Cores (for 5% Target Area Overhead)

IP Cores	Sub-Modules	# of Mods.	Obfuscation Efficiency		Design Overhead		Run time (s)
			Failing Verif. Pts. (%)	M_{obf}	Area (%)	Power (%)	
FPU	post_norm	20	98.16	1.56e-37	8.22	9.14	27
	pre_norm	20	94.16	1.24e-33	9.39	9.79	25
	pre_norm_fmul	20	90.00	7.36e-24	8.30	9.69	20
	except	10	100.00	6.85e-24	7.56	8.73	14
TCPU	control_wopc	20	92.79	7.53e-43	8.74	8.97	29
	mem	10	97.62	2.06e-20	8.29	9.76	15
	alu	10	97.62	9.82e-16	9.59	9.88	15

step and the process is repeated until the target area overhead is satisfied.

The proposed flow illustrated in Fig. 7 was implemented in TCL language and effectiveness of the obfuscation approach was studied for two Verilog IP cores - a single precision IEEE-754 compliant floating-point unit ("FPU"), and a 12-bit RISC CPU ("TCPU"), both obtained from [11]. In each case, the mode-control FSM was implemented using 4 state elements, with a *initialization key sequence* of length 3. The FSM was *hosted* in three RTL statements ($n = 3$). The size of the pool of modification signals was taken as ten ($m = 10$). Logic synthesis was carried out by Synopsys *Design Compiler*, using a ARM 90-nm standard cell library. Formal verification to estimate the difference between the original and the obfuscated design was carried out using Synopsys *Formality*. All simulations were carried out on a HP Linux workstation with a 1.5GHz processor and 2GB main memory.

Table 1 shows the obfuscation efficiency and the associated design overheads and the run-times, with the number of modifications adjusted for a 10% target area overhead. All results presented are at iso-delay. Note that all the modules had an area overhead less than the target area overhead of 10%. The functional obfuscation efficiency was estimated by the number of verification failures between the original and the obfuscated designs, while the semantic obfuscation efficiency was estimated by the metric M_{obf} as defined in Section 3. The increase in the compilation time of the obfuscated RTL with respect to the original RTL was negligible. The proposed flow was also very efficient in terms of run time, as seen from the table.

5 Conclusion

We have presented a practical RTL hardware IP protection technique based on a key-based obfuscation scheme. We integrate a small FSM of special structure into the design which affects its control and data flow based on the FSM states. The technique works on the CDFG representation of a design and can be easily automated. The proposed obfuscation technique provides an active defence mechanism that can prevent IP infringement at different stages of integrated circuits design and fabrication flow, thus protecting the IP vendors' interest, while incurring low design and computational overhead. Future work will include improving the obfuscation level and design overhead by extracting and using unreachable states of a RTL design.

References

[1] E. Castillo, *et al.* IPP@HDL: efficient intellectual property protection scheme for IP cores. *IEEE Trans. on VLSI*, 15(5):578-590, 2007.

[2] R.S. Chakraborty and S. Bhunia. Hardware protection and authentication through netlist level obfuscation. *Intl. Conf. on CAD*, 2008.

[3] R.S. Chakraborty and S. Bhunia. Security Through Obscurity: An Approach for Protecting Register Transfer Level Hardware IP. *HOST*, 2009.

[4] X. Zhuang, *et al.* Hardware assisted control flow obfuscation for embedded processors. *Intl. Conf. on Compilers, Arch. and Synth. for Embedded Systems*, 2004.

[5] C. Collberg, C. Thomborson and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. *Symp. on Principles of Programming Languages*, 1998.

[6] Thicket^{TM} family of source code obfuscators. [Online]. Available: `http://www.semdesigns.com`.

[7] T. Batra. Methodology for protection and licensing of HDL IP. [Online]. Available: `http://www.us.design-reuse.com/news/?id=12745\&print=yes`.

[8] R. Goering. Synplicity initiative eases IP evaluation for FPGAs. [Online]. Available: `http://www.scdsource.com/article.php?id=170`.

[9] M. Brzozowski and V.N. Yarmolik. Obfuscation as intellectual rights protection in VHDL Language. *Intl. Conf. on Comp. Info. Syst. and Indus. Management App.*, 2007.

[10] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on VLSI*, 35(8):677-691, 1986.

[11] [Online]. Available: `http://www.opencores.org`.