IDENTIFYING FPGA IP-CORES BASED ON LOOKUP TABLE CONTENT ANALYSIS

Daniel Ziener, Stefan Aßmus, Jürgen Teich

Department of Computer Science 12
University of Erlangen-Nuremberg
Am Weichselgarten 3; 91058 Erlangen
email: ziener@csfau.de, teich@cs.fau.de

ABSTRACT

In this paper we introduce a new method to identify IP cores in an FPGA by analyzing the content of lookup tables. This techniques can be used to identify registered cores for IP protection against unlicensed usage. We show methods to extract the content of the lookup tables in a design from a binary bitfile of Xilinx Virtex-II and Virtex-II Pro FPGAs. To identify a core, we compare the number of unique functions from lookup tables of the core with the lookup tables extracted from a product with an FPGA from an accused company. Also placement information can be used for increasing the reliability of the result. With these methods, no additional sources or information must be inquired from the accused company. These techniques can be used for netlist and bitfile cores, so a wide spectrum of cores can be identified.

1. INTRODUCTION

In embedded systems, many cores are used and reused, which have been written for other projects or were obtained from other sources. The advantages of the reuse of IP cores (Intellectual Property cores) are enormous. They offer a modular concept, fast development-cycles, and excellent cores are commercially available today from specialized core developers.

IP cores are licensed and distributed like software. One problem of the distribution of IP cores is the lack of security against unlicensed usage. As the cores are provided e.g. as netlist data, they can easily be copied like software. There is only little effort to get the unlicensed core to function. Some core suppliers encrypt their cores and deliver special development tools, which can handle these cores. The disadvantage is that common tools usually cannot handle encrypted cores and that the shipped tools can be cracked.

Another approach is to hide a signature into the core, so called *watermark*, which can be used to proof the original ownership. There exist many concepts and approaches on the issue of implementing a watermark into a core. But most of these concepts are not applicable due to the lack of verification capabilities. A good verification strategy is that the signature (watermark) can be read out only by using the purchased product. No extra files or information must be obtained from the accused company.

In our approach, we use the content of the lookup tables from the FPGA-core which should be protected like the signature. The main functionality of the core is represented by the lookup table content, so these values identify a core uniqely. If the owner of the core would find all lookup table values in an FPGA-design of an accused company, he can be almost sure that his core was used.

This work is organized as follows. In Section 1, an introduction and motivation is given. In Section 2, a short overview of related work for IP-Watermarking is given, and in Section 3 the concept of our approach is presented. Section 4 shows how lookup table contents can be extracted from a given product with a Xilinx FPGA. In Section 5, a method which identifies a core by the lookup table values is introduced. Section 6 shows experimental results and Section 7 concludes the results.

2. RELATED WORK

Hiding a unique signature into user data, such as pictures, video, audio, text, program code, or IP cores is called watermarking. The watermarking of IP cores is different from multimedia watermarking, because the user data, which represents the circuit, must not be altered, since functional correctness must be preserved. Todays watermarking procedures can be categorized into two groups of methods: additive methods and constraint-based methods.

Additive methods have in common that the signature is added to the functional core, for example, by using unused lookup-tables in an FPGA [8]. The constraint-based methods were originally introduced in [4] and restrict the solution space of an optimization algorithm by setting additional constraints which are used to encode the signature.

Some methods for constraint-based watermarking in FPGAs exploit the wire wrap of a scan-chain [7], preserve nets during logic synthesis [6], place constraints for CLBs in odd/even rows [5], or route constraints with unusual routing resources [5].

The major drawback of these approaches are the limitations of the verification possibilities of the watermarked

1-4244-0 312-X/06/$20.00 ©2006 IEEE.

core. With a good watermarking strategy, the verification can be done only with the given product without additional information from the developer of the core.

The introduced approaches at the HDL- and netlist-levels turn out not to be applicable due to the lack of verification possibilities. The problem of watermarking FPGAs is not the coding and insertion of a watermark, rather than the verification with an FPGA embedded in a system.

3. CONCEPT

In our approach, we do not add any signature or watermark. The core itself is remains unchanged, so the functional correctness is given and no additional resources are used. We compare the content of the used lookup tables from the core with the used lookup tables in the FPGA from the product of the accused company. If a high percentage of consistency is detected, the probability that the registered core is used is very high.

The synthesis tool maps the combinatorial logic of an FPGA-design into lookup tables and writes these values into the netlist of the core. After the synthesis step the content of the lookup tables of a core is known, so we can protect cores which are delivered at the netlist level. The protection of cores at the bitfile level is also possible.

After the core is purchased, the customer can combine these cores with other bought or self developed cores. In the following CLB mapping step, it is possible that lookup tables were merged across the core boundaries or were removed by an optimization step. This happens when different cores share logic or when outputs of the core are not used. These lookup tables cannot be found in the FPGA bitfile, but experimental results (see Section 6) show that the percentage of these lookup tables compared to the number of all lookup tables in the core is low.

If a company is accused to use unlicensed cores in a product, the bitfile of the used FPGA can be extracted (see Section 4). After reading out the content and the positions of the lookup tables from the bitfile and comparing them with the lookup table contents form the original core (see Section 5), the ownership of the core can be proven.

4. LOOKUP TABLE CONTENT EXTRACTION

In this section we discuss which possibilities exist to get information about the contents of lookup tables from a product. First, we need to extract the configuration bitfile of the FPGA in the product. On some devices, it is possible to read back the bitfile. This is the easiest way, but it is not always possible, because not all FPGA devices support this or the bitfile creator can disable this feature. In SRAM based FPGAs, the bitfile is stored into a PROM, and during the startup phase the FPGA is configured by loading this bitfile. The communication between the FPGA and the PROM can be recorded by wire tamping and so the bitfile can be reconstructed.

The extraction of the content of the lookup table from a configuration bitfile depends on the FPGA device and the FPGA vendor. For Xilinx FPGAs, there exist two methods: with the Xilinx tool JBits [11] and directly from the bitfile.

JBits can read and manipulate Xilinx bitfiles. With this tool it is easy to extract all used lookup table contents with the slice position. But only some devices, like Virtex, Virtex-E or Virtex-II are supported by JBits.

The other way is to read out the LUT content directly form the bitfile. Here, it must be known at which position in the bitfile the lookup table content is stored. Also, the right interpretation of the values must be known. This information can be obtained by altering the lookup table contents from a well known bitfile with the tool FPGA-Editor from Xilinx and recognize the difference between the old and the new created bitfile. With these informations, it is possible to find the right positions and the right byte representation of lookup table contents.

We have experimented with these observations for Xilinx Virtex-II and Virtex-II Pro bitfiles. These bitfiles are structured in packets. A packet consist of one or more configuration words for a certain configuration register, which controls the configuration process [12]. One large packet contains the configuration data for the circuit, which should be instantiated into the FPGA. This packet is divided into frames, which is the smallest configuration unit. The length of one frame and the number of frames is device dependant.

There exist six frame types: IOB, IOI, CLB, BRAM, BRAM Interconnect, and GCLK. On the left and on the right side of the FPGA exists one column of IOB frames; each IOB column consists of four frames. There are also 22 IOI frames on each side of the FPGA. In the CLB frames, information about the CLBs, routing and the upper and lower IOBs are stored. The FPGA consists of a regular pattern of CLB columns. Each CLB column consists of 22 frames. In two of these frames, the content of the lookup tables are stored. More information can be found in [12].

To access the frames into the configuration memory in the FPGA, each frame has an address. The major address directs the columns, where the minor address directs the frames in a column. If a FPGA is configured completely, this is also the order in which the frames are stored in the configuration packet in the bitfile.

A Xilinx Virtex-II CLB consists of four slices, two horizontal and two vertical. Each slice has two lookup tables, the G- and the F-LUT. The content of the lookup tables for one slice column is stored in one frame, so two frames are used for the lookup table content in a CLB column. In the second frame of a CLB column the lookup table content of the left slice column is stored, whereas in the third frame the content of the right slice column is included.

16 bits are stored in a four input lookup table. These bits are stored together in two bytes but with bit inverted values. The F and G lookup table in a slice is separated by one byte, which is not used for storing lookup table content

informations. So, the lookup table content packet for one slice consists of 5 bytes. First, the G-LUT is stored, but in reverse bit order. Then, the separate byte and the F-LUT is stored. The bit order of the F-LUT is not reversed.

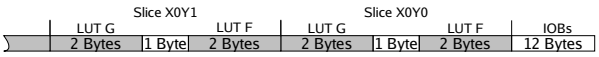

Fig. 1. The positions of the lookup table content in a frame.

These packets which contain the lookup table content are stored successively in the frame for the slices in one column, beginning with the slice with the highest Y coordinate and ending with the slice with the coordinate 0 (see Figure 1). For the upper and lower input/output blocks, which are also stored in the CLB frames, the first and the last 12 bytes in the lookup table content frame is reserved. The frame length FL is:

$$FL = CY \cdot 2\ Slices \cdot 5\ Bytes + 2 \cdot 12\ IOB\ Bytes,$$

where CY is the number of CLB rows in the FPGA.

To calculate the frame address of the lookup table frames, we can use the following formula:

$$F_{cy0} = 1 \cdot F_{GCLK} + 1 \cdot F_{IOB} + 1 \cdot F_{IOI},$$
$$f_{lf}(x) = F_{cy0} + CF_{lut0} + \lfloor x/2 \rfloor \cdot F_{CLB} + x\ mod\ 2,$$

where F_{cy0} denotes the frame number of the first CLB frame and F_{GCLK}, F_{IOB}, F_{IOI} and F_{CLB} identifies the number of frames of a GCLK, IOB, IOI or CLB column. CF_{lut0} denotes the first lookup table content frame in a CLB. x and y are the slice coordinate, and Y the number of slice rows in the FPGA. $f_{lf}(x)$ denotes the frame for lookup tables in slice column x. All addresses begin with zero.

F_{GCLK}, F_{IOB}, F_{IOI}, F_{CLB} and CF_{lut0} are device independent and can be inserted:

$$F_{cy0} = 1 \cdot 4 + 1 \cdot 4 + 1 \cdot 22,$$
$$f_{lf}(x) = F_{cy0} + 1 + \lfloor x/2 \rfloor \cdot 22 + x\ mod\ 2,$$
$$f_{lf}(x) = 31 + \lfloor x/2 \rfloor \cdot 22 + x\ mod\ 2,$$

To calculate the byte address of the lookup tables in the configuration packet of the bitfile, we can use the following formula:

$$Adr_{LUTG}(x,y) = f_{lf}(x) \cdot FL + 12 + ((Y-1) - y) \cdot 5,$$
$$Adr_{LUTF}(x,y) = f_{lf}(x) \cdot FL + 12 + ((Y-1) - y) \cdot 5 + 3.$$

Lookup tables in unused slices have the value 0x0000, whereas unused lookup tables in used slices have the value 0xFFFF. With this information, we are able to extract and decode the lookup table content and the position from used lookup tables in a Xilinx Virtex-II and Virtex-II Pro FPGA.

For other FPGA devices or vendors we expect that similar techniques can be used to find rules of how to extract the lookup tables contents.

5. IDENTIFYING THE CORE

After the extraction of the content of lookup tables in a bitfile, we can compare these values with the information in the netlist. The content of the lookup table can easily be read out from a netlist file. For example, in an EDIF netlist for Xilinx FPGA devices the lookup table content is on the INIT property for the lookup table instances. Unfortunately, the mapping tools adopt these values not necessarily. The mapping tool may merge lookup tables from different cores together, converts one, two or three input lookup tables to a four inputs lookup table and permutes the inputs to achieve a better routing.

If the protected core is used with other cores or user logic, it is possible that the mapping tool merges lookup tables form different cores together. This is done when combinatorial logic or registers can be shared by more than one core. This allows the mapping tool to perform further optimization steps. In these steps, lookup tables can be merged together or lookup table inputs can be switched from one lookup table to another. These lookup tables cannot be found later in the bitfile. So it is possible that on the border of the protected core we cannot find all lookup tables from the netlist in the bitfile. But experimental results (see Section 6) show that the percentage of merged lookup tables is low for the used mapping tool (Xilinx map).

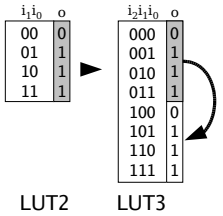

Fig. 2. Converting a two input lookup table into a three input lookup table with unused input i_2.

All lookup tables of an FPGA have n inputs. On most FPGA architectures, lookup tables have four inputs. In a core netlist also lookup tables with less than n inputs may exist. These lookup tables must be mapped into n input lookup tables. If one input is unused, only half of the memory is needed to store the function and the remaining space must be filled. This also depends on the FPGA architecture, but it makes sense to store the function so, that it is regardless if on the unused input a one or a zero is applied. This can be done if the content of a $n-1$ input lookup table is copied into the unused region of the n input lookup table (see Figure 2). This can also be applied if the lookup table has more than one unused input.

The mapping tool can permute the inputs of the lookup tables to achieve a better routing. In most FPGA architectures, the routing resources for lookup table inputs are not equal, and so a permutation of the lookup table inputs can lower the amount of used routing resources. Permutation of the inputs alter significantly the content of a lookup table.

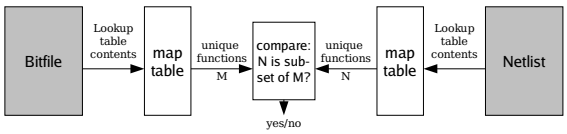

Fig. 3. Before the lookup table contents from the bitfile and the netlist can be compared, they must be mapped into unique functions.

For n inputs, $n!$ permutations exist and up to $n!$ different lookup table values for one unique function. To compare the contents of the lookup table from the netlist and the bitfile, it must be checked if one of these possible different lookup table values for one unique function is equal to the value of the lookup table in the bitfile. This is done by creating a table with all possible values of lookup tables for all unique functions (see Figure 3).

For robustness analysis of our approach, it is necessary to know how many different functions can be realized by an n input lookup table, if it is allowed to permute the inputs. This topic is related to the problems of Boolean matching and equivalence classes [2] [3] [13] [1]. Boolean matching is a technique to check if two Boolean function are equal with respect to transformations (e.g., input permutations, input negotiation, output negotiation). The functions which are equal with respect to these transformations belong to one equivalence class. Our interest is on the equivalence class P which allows to permute the inputs of a function.

Inputs n	equivalence classes P	unique functions f
1	4	2
2	12	10
3	80	78
4	3984	3982
5	37333248 [2]	37333246

Table 1. The number of different equivalence classes P and unique functions.

The number of different values which can be encoded in a n-input lookup table is 2^{2^n}. The number of equivalence classes P is lower. Table 1 shows the number of equivalence classes of functions with different numbers of inputs. These values were achieved by experimental results. Functions which have a constant output do not appear in a netlist, but the mapping tool can create such functions later to generate a VCC or GND signal. Nevertheless, we reduce the number of unique functions by the two constant functions.

Now, in order to decide whether a certain core is used in the FPGA, we define two tuples N and M, that contain the numbers of each unique functions from the core (N) and the whole design (M) that can consist of multiple different cores. Also, the number of different unique functions f is of interest (see Table 1).

$$
\begin{aligned}
N &= (n_1, n_2, n_3, \cdots, n_f), \\
M &= (m_1, m_2, m_3, \cdots, m_f), \\
n &= n_1 + n_2 + n_3 + \cdots + n_f, \\
m &= m_1 + m_2 + m_3 + \cdots + m_f,
\end{aligned}
$$

The number of elements in the tuples is the number of different unique functions f. n_1 means that the function 1 is n_1 times included in the core. n is the number of all functions or lookup tables in the core, and m the number of lookup tables in the whole design. The tuple N can be achieved from the different unique functions of the core netlist and the tuple M can be created from the information of the lookup table content extraction from the bitfile.

If the number of each different unique function in M is higher or equal to N, the core may be possibly included in the design.

$$\forall k \in \{1, \cdots, f\}: \quad m_k \geq n_k$$

If a high percentage of unique functions of M is lower than in N, the core is surely not included.

An important value is the probability p_{fd} that a core is found in a design but which has this core not included. This value should be very low to obtain a high reliability of this method. This value depends on m, n, f, the core N and the probability distribution P_m of the unique functions of the whole design.

$$
\begin{aligned}
P_m &= (p_1, p_2, p_3, \cdots, p_f), \\
p_1 + p_2 + p_3 + \cdots + p_f &= 1,
\end{aligned}
$$

where p_1 is the appearance probability of the function 1 in the design M.

First, we assume that $m = n$. This means that we calculate the probability that the number of each unique function of M and N is equal. All lookup tables of N must be in M with the probability distribution P_m. This can be calculated with the multinomial distribution.

$$p_{fd} = \binom{n}{n_1, n_2, n_3, \cdots, n_f} \cdot p_1^{n_1} \cdot p_2^{n_2} \cdot p_3^{n_3} \cdot \ldots \cdot p_f^{n_f},$$

If $m > n$, than the core N is combined with other cores in the design M. For each possible combination from N with the other cores, the appearance probability must be calculated and summed. The first question is how much combinations of other cores (s) exists. This can be calculated with the formula of combination with repetitions:

$$s = \frac{(f + m - n - 1)!}{(m - n)! \cdot (f - 1)!}$$

Now, all possible combinations of functions in these cores are calculated and stored in a matrix A.

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1f} \\ a_{21} & a_{22} & \ldots & a_{2f} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \ldots & a_{sf} \end{pmatrix},$$

$$\forall k \in \{1, \cdots, s\}: \quad a_{k1} + a_{k2} + \cdots + a_{kf} = m - n$$

For example, for $f = 2$ and $m - n = 3$:

$$A = \begin{pmatrix} 0 & 3 \\ 1 & 2 \\ 2 & 1 \\ 3 & 0 \end{pmatrix}$$

The probability that a core is detected in M with the probability distribution P_m is:

$$p_{fd} = \sum_{i=1}^{s} \binom{m}{n_1 + a_{i1}, n_2 + a_{i2}, \cdots, n_f + a_{if}} \cdot \cdot p_1^{n_1+a_{i1}} \cdot p_2^{n_2+a_{i2}} \cdot \ldots \cdot p_f^{n_f+a_{if}},$$

If we assume that all unique functions in M are uniformly distributed, we can simplify this formula:

$$p_{fd} = \sum_{i=1}^{s} \binom{m}{n_1 + a_{i1}, n_2 + a_{i2}, \cdots, n_f + a_{if}} \cdot \frac{1}{f^m}$$

Unfortunately, only for small values of m, n, and f, this probability can be calculated (see Figure 4 and 5), because of the exponentiated computation complexity. For values of m, n, and f which can appear in realistic applications we can only reference to the experimental results in Section 6.

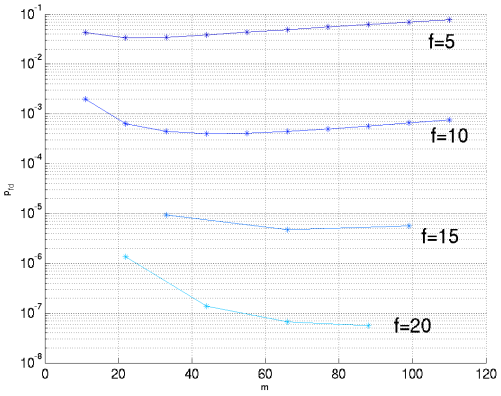

Fig. 4. p_{fd} for different f and m. $n = 0.91 \cdot m$ and N is uniformly distributed.

In order to increase the robustness of our method and lower the risk of a false detection of a core, the positions of the lookup table in a FPGA can be used. The position of a lookup table can be extracted from the bitfile as well as the content (see Section 4). We assume that the lookup tables of a core are placed close together. Elements with direct connection are tried to placed together. The assumption is that lookup tables of a core have more connections with elements inside the core than with elements which do not belong to the core.

If the mean distance between the found lookup tables in the bitfile is low, than the probability that these lookup tables are part of the searched core is higher than when the mean distance is high. In order to calculate this mean distance,

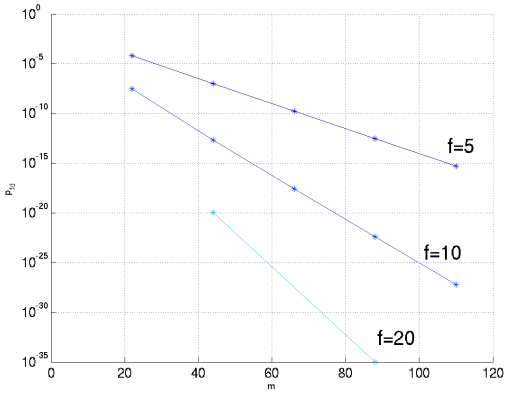

Fig. 5. p_{fd} for different f and m. $n = 0.91 \cdot m$ and N is distributed as follow: $p_{n1} = \frac{1}{2} + \frac{1}{2f}$ and $\forall k \in \{2, \cdots, f\}$: $p_{nk} = \frac{1}{2f}$.

we must know, however which lookup table in the bitfile belongs to the core.

First, we search for functions which only appear in the core, i.e.

$$\forall k \in \{1, \cdots, f\} : \quad m_k = n_k.$$

Lookup tables which implement these functions are surely inside the core. Form these lookup tables we can calculate an estimate of the position of the core center. If the number of these functions is zero, we take all lookup tables which implement functions from the core to calculate the core center. In this case, the core center is inaccurate because also lookup tables which do not belong to the core are considered. For functions which appear in and outside the core, i.e.,

$$\forall k \in \{1, \cdots, f\} : \quad m_k > n_k,$$

we take the n_k lookup tables which are nearest to the calculated core center. Now, the distance to the core center of all lookup tables of the core can be calculated.

In the formula for p_{fd}, the difference between m and n is important. If $m - n$ is small, than also the probability of a false detection p_{fd} is low. We can decrease $m - n$ if we define a bounding box around the core and only consider lookup tables inside the box. The dimensions of the box can be calculated from the positions of the lookup tables from the core inside the bitfile. But to this probability p_{fd} the number of all possible opportunities of the position of this bounding box in the design must be multiplied.

6. EXPERIMENTAL RESULTS

For experimental results, we used a keyboard controller as an example design [10]. This design consists of four cores: *strober*, *producer*, *analyser*, and *fsm*. For all of these cores, the lookup table values and the corresponding unique functions were extracted from the netlists of these cores. The whole design was implemented on a Xilinx Virtex-II FPGA

using the Xilinx tool ISE 6.3i. From the resulting bitfile, all lookup table values were extracted by the method described in Section 4. Now, each core are identified with the methods in Section 5 and the results are shown in Table 2.

Core	n	m	f	found n	distance d
producer	40	450	3982	40	4.225
strober	93	450	3982	93	3.797
fsm	6	450	3982	5	0.883
analyser	379	450	3982	217	5.946

Table 2. Results for identifying cores in a design where the cores are included. The values for mean distance to the core center d are in LUT positions.

The results show that all lookup tables for the core *producer* and *strober* were found. During the implementation, many lookup tables for the core *analyser* were removed and are not found in the bitfile. This can result from unused outputs, or constant inputs. Which lookup tables are affected in these cases can be evaluated by the core developer with reference designs, and so, the results of the identify process can better be interpreted. The mean distance to the calculated core center in all four cases is small, so this and the high percentage of found lookup tables confirmed the assumption that the core is included in the bitfile. To verify the

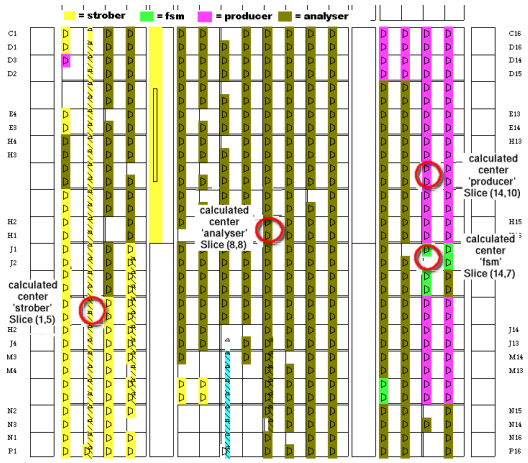

Fig. 6. The calculated core centers compared with the real LUT placements.

calculated core centers, we compare the core centers with the real placement of the slices in the cores. Figure 6 shows that is calculated core center positions correspond with the real positions of the cores.

To evaluate the robustness, we try to find these four cores in a bitfile were these cores are not included. For this case, we implemented a Des56 design [9] in a Xilinx Virtex-II FPGA and extracted all lookup tables. Table 3 shows that the percentage of the found lookup tables are low and the mean distance to the calculated core center is high. These values shows that these cores are not included in the design.

Core	n	m	f	found n	distance d
producer	40	1574	3982	2	5.55
strober	93	1574	3982	44	15.97
fsm	6	1574	3982	3	12.597
analyser	379	1574	3982	69	13.865

Table 3. Results for identifying cores in a design where the cores are not included.

7. CONCLUSIONS

We have presented a new method to identify FPGA cores in FPGA bitfiles. The extraction of lookup table contents of a binary bitfile was demonstrated for Xilinx Virtex-II and Virtex-II Pro devices. The extraction of the lookup table values from the netlist is also shown for Xilinx devices. Possible transformations of the mapping tools and the effect of the robustness of the method were discussed. The experimental results show that is possible to identify a core in the design with a high probability. The identification process is based on two parameters, namely the number of found lookup tables of the core in the design and the mean distance to the core center. However, it must be taken into account that lookup tables of the core are removed by optimization tools, if part of the core are not used because outputs are open or constant values apply to inputs.

8. REFERENCES

[1] D. Debnath and T. Sasao. Fast Boolean Matching Under Permutation Using Representative, 1999.

[2] M. A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill, 1965.

[3] M. Hütter. *Logic Synthesis with Complex Gates*. dissertation, 2003.

[4] Kahng, Lach, Mangione-Smith, Mantik, Markov, Potkonjak, Tucker, Wang, and Wolfe. Constraint-Based Watermarking Techniques for Design IP Protection. *IEEETCAD*, 20, 2001.

[5] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Robust IP Watermarking Methodologies for Physical Design. In *Design Automation Conference*, pages 782–787, 1998.

[6] D. Kirovski, Y.-Y. Hwang, M. Potkonjak, and J. Cong. Intellectual property protection by watermarking combinational logic synthesis solutions. In *proceedings of ICCAD*, pages 194–198, 1998.

[7] D. Kirovski and M. Potkonjak. Intellectual Property Protection Using Watermarking Partial Scan Chains For Sequential Logic Test Generation. In *ICCAD*, 1998.

[8] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Signature Hiding Techniques for FPGA Intellectual Property Protection. In *proceedings of ICCAD*, pages 186–189, 1998.

[9] Opencores.org. Basic DES Crypto Core. overview.

[10] Opencores.org. Keyboardcontroller. overview.

[11] Xilinx Inc. JBits SDK. www.xilinx.com/labs/projects/jbits/.

[12] Xilinx Inc. Virtex-II Platform FPGA User Guide (UG002). 2.0, pages 269–358. Mar. 2005.

[13] Z. Zilic and Z. G. Vranesic. Using BDDs to design ULMs for FPGAs. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, pages 24–30, New York, NY, USA, 1996. ACM Press.