DGECS: Description Generator for Evolved Circuits Synthesis

Fabio Cancare *, Davide B. Bartolini *, Matteo Carminati *, Donatella Sciuto *, and Marco D. Santambrogio+†

* Dipartimento di Elettronica e Informazione, Politecnico di Milano
{cancare, bartolini, mcarminati, sciuto, santambr}@elet.polimi.it

† Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
santambr@mit.edu

Abstract—Evolvable Hardware (EHW) is an approach to the creation of hardware circuits based on a goal-oriented evolutionary process inspired by natural evolution. This approach allows the exploration of a very large design search space, ideally enabling to find solutions that are better in terms of resource requirements, accuracy or timing performance, with respect to traditional design methods. To exploit this approach, it must be possible to port the evolved circuits to custom designs; however, in FPGA-based EHW systems (and, in particular, in the HERA project), the configuration bitstream for an evolved circuit is specific to the evolutionary platform and it cannot be ported to a different architecture.

This paper expands the HERA framework with a tool able to export hardware circuits evolved within the HERA framework to an IP-core reusable in any PLB-based custom design. DGECS (Description Generator for Evolved Circuits Synthesis) permits to export evolved circuits to a VHDL description which can be then synthesized and plugged into a custom PLB architecture. Experimental results provide evidence that DGECS allows to correctly export evolved circuits; moreover, it enables to save resources thanks to the optimizations introduced by the synthesis flow it relies on.

I. INTRODUCTION

The era of digital electronics has witnessed many design techniques and approaches aimed at keeping the pace with the increasing number of transistors on a silicon chip [1]. For decades, this trend in silicon technology has been closely following Moore’s law. In spite of the support of EDA tools, high-level abstraction design techniques, and advancement in IP-core libraries (i.e., libraries made of ready-to-use components), the design-productivity gap is increasing. The same increase can be found in the complexity of electronic systems and in the amount of faults and errors characterizing these systems. These issues are posing a challenging situation in front of the researchers, with the need for a next breakthrough in hardware design techniques.

Inspiration for innovating the hardware design scenario has been found by researchers in an apparently unrelated context. Nature has been able to create (biological) systems far more complex than the (artificial) ones created by the humankind; such systems are the result of an evolutionary process started millions of years ago and still ongoing. The idea of exploiting a similar process for producing hardware circuits and to comply with the requirements of the modern industrial processes (e.g., time to market) represents a promising breakthrough. Starting from this idea, researchers began working on what has been eventually called Evolvable Hardware (EHW) [2]–[6].

Among all the available technologies, FPGA-based Evolvable Hardware systems have emerged as successful solutions for creating math components, controllers, image and audio filters and to deal with fault recovery and system adaptation [7]–[14]. The HERA (Hardware Evolution over Reconfigurable Architectures) project is one of the research projects aimed at creating a FPGA-based evolvable hardware system. Several recent papers [15]–[18] describe the goals and the progresses of the HERA project towards the design and the implementation of such a self-evolvable hardware system.

To allow the HERA framework becoming a real alternative to standard techniques, by creating circuits through evolvable hardware, the evolved circuits must be reusable and portable to the desired design; this step, however, was not possible, as the circuits were bound to the evolutionary platform. The objective of the work described in this paper is to design and implement a tool for extracting and exporting evolved circuits as IP-cores. Prior to this work, this was not easily doable because the configuration bitstream for an evolved circuit (i.e., the solution to a certain problem found through the evolutionary procedure) is specific to the device used for the evolution and a tool allowing to turn this representation to an IP-core was missing. This paper presents DGECS (Description Generator for Evolved Circuits Synthesis), a tool that allows replicating the functionality of an evolved individual into a component described in VHDL and portable to different devices and hardware designs. Moreover, DGECS enables to exploit the optimization mechanisms available in the hardware synthesis tools to simplify the structure of the originally evolved individual (which may be redundant, due to the stochastic characteristics of the evolution process), allowing, whenever possible, to save area even when reusing the exported IP-core over the same platform used for the evolutionary process.

The novel contributions of this work with respect to the previous publications [15]–[18] and to the state of the art are:

- the description of the design and implementation of a tool, namely DGECS, able to extract and export hardware components developed through the Evolvable Hardware design flow. The tool has been implemented in order to work with the HERA framework;
• the validation of the proposed tool against the case studies proposed in the past research papers.

The source code of the DGECS tool and of the HERA framework, as well as the Xilinx EDK projects specific to the device used to gather the experimental results, are publicly available online [19]; the project website also allows to download bitstreams and binaries, providing demos of the HERA framework and the possibility to try the DGECS tool with some evolved circuits.

In the remaining of this paper, Section II provides an overview of the Evolvable Hardware research field and introduces a taxonomy characterizing the various EHW approaches. Section III describes the HERA project, focusing on the aspects closely related to the work proposed in this paper. Section IV presents the DGECS tool and provides its main implementation details. Section V reports some experimental results on components generated using DGECS. Finally, Section VI wraps up the authors conclusions and proposes future research directions.

II. EVOLVABLE HARDWARE AT A GLANCE

The Evolvable Hardware (EHW) paradigm proposes an alternative to traditional circuits design techniques, which follow the various phases of design and testing [20]. This technique is based on an evolutionary process undertaken by populations of circuits, which evolve until a suitable solution is found [5]. EHW can be considered as a scheme for designing hardware components that can change their behavior dynamically and autonomously to adapt to the surrounding environment. This is accomplished thanks to evolutionary strategies aimed at improving the components behavior with respect to a given specification. These evolutionary strategies are stochastic search methods that mimic the process of natural evolution and are implemented as Evolutionary Algorithms (EAs) [21]. As in biological organisms, components are encoded by a genotype, which in turn determines their characteristics and behavior, called phenotype. This approach allows the exploration of a very large design search space, which can ideally enable EHW to find solutions that are better in terms of resource requirements, accuracy or timing performance than those found using traditional design methods [13], [22].

A. Evolvable Hardware: base idea

A typical evolutionary algorithm is based on the concepts of population and generation: each population is formed by a (constant) number of individuals. An individual represents a candidate solution, a circuit that is tested against the desired I/O behavior. Algorithm 1 shows the pseudo code of a standard genetic algorithm; variations of this algorithm are actually implemented in EHW systems.

After a (usually random) initialization of the first generation, each individual is evaluated to see how its behavior compares to the desired one; each individual is assigned a fitness value (i.e., a score), and circuits closer to the desired I/O behavior receive a higher fitness value. After the first evaluation, the evolutionary loop starts and proceeds until either a solution is found or a fixed maximum number of generations is reached.

Within the loop, the first operation is the selection (based on their fitness) of a subset of the individuals that will be used to create the next generation. The selected circuits are manipulated through genetic operators (the most used ones are crossing over - or crossover - and mutation [23]). These operators diversely alter the genotype of the individuals, introducing modifications that allow the exploration of the design space towards a solution. At the end of the evolutionary loop, the whole population is evaluated again, new fitness values are assigned to the new individuals and, if no termination condition is met, the loop starts again.

B. Evolvable Hardware: taxonomy

The first and the most important classification for Evolvable Hardware systems is related to how candidate solutions are evaluated [8]: in extrinsic evolution circuits are evolved using a simulation environment, thus the EA relies on simulated evaluations. Once a suitable solution is found, it is then implemented within the device; in intrinsic evolution: the candidate solution is directly mapped and evaluated on the target device, and the EA relies on actual evaluations.

Intrinsic evolution is more accurate and it allows to create self-Evolvable Hardware systems; this technique can be further improved by implementing the EA as a hardware module (complete intrinsic evolution), allowing to endlessly evolve new circuits, while having the best solution found so far up and running (open-ended intrinsic evolution) [7], [8]. On the other hand, the main drawback of the intrinsic approach is the high time overhead of the reconfiguration process [24].

Digital Evolvable Hardware techniques can also be classified considering the physical level at which evolution is performed [7]: it is possible to distinguish between gate level evolution (or fine-grained EHW) and function level evolution. Fine-grained EHW maximizes the exploration by searching in a large solution space (the space of all the electronic circuits composed of a certain type), providing very efficient solutions to medium-small size problems [10]. On the other hand, function-level EHW generally involves selecting topologies and assembling circuits composed of high-level functional building blocks. Function-level EHW emphasizes the exploitation of functional building blocks, which are known to be necessary and sufficient to solve the problem, thus being more supervised and exploring a smaller solution space.
III. THE HERA APPROACH

The development of the HERA system has already been comprehensively described in literature: from the initial considerations about the FPGA configuration details [15] to the overall system architecture [16]; from the design of an evolutionary technique tailored for FPGA-based circuits [17] to the most recently introduced improvements for achieving a better scalability with respect to the problem size and for boosting the timing performance [18]. The following paragraphs sum up the main characteristics of the framework.

The HERA approach is based on a tight coupling between an evolutionary algorithm and a dynamic reconfigurable, FPGA-based, architecture. Details about the EA are not relevant for this work, but information about the evolutionary system architecture is necessary to understand the design and implementation choices presented in Section IV.

The HERA evolutionary architecture is structured as a bus-based design and it is represented in Figure 1. The main architecture components are [17]:

- the Reconfigurable Area, which contains the candidate solutions undergoing the evolutionary process;
- the PowerPC Processor, which executes the genetic algorithm and generates the information for the dynamic partial reconfiguration of the evolvable region;
- the ICAP Controller, which performs the device internal reconfiguration process;
- the EHW Controller, which provides access to the evolvable region. It allows retrieving and evaluating the outputs of the candidate solutions.

For the scope of this paper, the most relevant component of the HERA architecture is the reconfigurable area. The structure of this area determines the composition of the evolved circuits (which are bound to the device used in the evolutionary process); the DGECS tool exports this device-specific representation to a generic VHDL description. The structure of the candidate solutions was conceived both to be suitable for their evolution through the EA and to fit into the resources of the FPGA (a Xilinx Virtex-4 XC4VFX12) used in the evolutionary platform. Thus, the structure of the evolved circuits is highly dependent on the target FPGA structure. Two different types of candidate solutions are currently supported in the HERA framework:

- the first type is characterized by an 8bit I/O data path and is built upon a hierarchical structure, at the base of which are the LUTs, which determine its behavior;
- the second type provides a wider data path (32bit I/O) and it is built upon the 8bit individual by adding a level at the top of the hierarchy.

Since the 8bit individual is a subset of the 32bit one, the remaining of this Section describes the latter and indicates the differences between the two.

Figure 2(c) shows the elementary block the whole individual is made up of: the slice. Each slice contains two LUTs (called LUTF and LUTG, respectively), one multiplexer (MUXF5) and two FLIP FLOPs. The configuration of the lookup tables (16 bit for each one) is the object of the evolutionary process and represents the genotype the genetic algorithm works on. The multiplexer is in charge of selecting the output of one of the two LUTs in order to be propagated to the FLIP FLOP (only one among the two available ones is actually used) and then to the slice output. The same output is also used as selecting signal for the multiplexer: in this way both combinatorial and sequential circuits can be evolved. By allowing the system to evolve sequential circuits, a wider solution space can be explored. Despite of this, our attention is focused on the evolution of combinational circuits. In order to obtain such a behavior, the configuration of the two lookup tables for each slice needs to be the same. In summary, each slice has four input bits, which are fed to both the LUTF and the LUTG, with the same order, and one output bit.

Slices are grouped into columns – Figure 2(b). Each column contains eight slices; in order to create a module, four columns are needed. The input data path for a column is made up of 8 bits, coming from the previous column, the previous module, or from the overall input. Between each column, signals are statically routed in order to mix the input bits of the next column, while being sure of forwarding all the output bits of the previous column. The signals routing is statically fixed during the architecture synthesis and it is not altered during the evolutionary process; this routing is replicated (as explained in Section IV) by DGECS in the exported IP-core, to maintain the functionality of the evolved circuit.

An 8bit data path characterizes a module; the module represents the first version of the individual, the one with a smaller data path. Going up in the hierarchy, it is possible to obtain an individual with a wider data path (32 bits) simply connecting 16 modules. The wider individual is organized as a $4 \times 4$ matrix of modules, as shown in Figure 2(a). Each column of modules has a 32bit input data path and a 32bit output data path.
After having introduced the HERA framework, the goals of this work can be better defined. DGECS aims at:

- implementing a tool able to read the configuration of an evolved circuit (i.e., the configurations of its LUTs) and to export it as a standalone component, implementing the same logic functionalities of the original individual;
- providing portability (the exported component is independent from the architecture the individual was evolved on), and resource optimization;
- offering the possibility to import in a straightforward way the circuits evolved on devices different from the reference one (Xilinx Virtex-4 XC4VFX12).

IV. DGECS DESIGN AND IMPLEMENTATION

This Section focuses on the design and implementation of DGECS: the tool that allows exporting the circuits evolved with the HERA framework to a portable description. DGECS completes the EHW design flow, enabling to export the device-dependent bitstream of the circuits to a VHDL representation. This flow is represented in Figure 3. Based on a fitness function describing the desired I/O behavior of a circuit, HERA provides an evolved individual characterized by its structure and the configuration of its LUTs. These data represent the input of DGECS, which is able to wrap them up in a PLB-compliant IP-core as a VHDL description. This description can be synthesized, through a synthesis toolchain, to produce a circuit for a target device (possibly different from the one of the evolutionary platform). This flow allows to exploit EHW, through HERA, to produce circuits as PLB-compliant IP-cores; moreover, thanks to the VHDL description produced by DGECS, it exploits the optimizations introduced by the synthesis toolchain to eliminate possible redundancies in the evolved circuit.

DGECS is composed of a static part, which describes the structure of the individual, and of a dynamic part, which automatically generates an IP-core with the same functionality as the evolved circuit.

Fig. 3. EHW design flow with HERA; DGECS fills the lack of a tool to export the evolved circuits to a reusable IP-core, enabling to employ HERA as a real alternative to standard design flows.

A. Individual description implementation

To reach the objective of creating a portable description of any combinatorial evolved circuit, it was necessary to implement the hierarchical structure of an individual (for both the 8bit and 32bit versions). Moreover, one of the motivations for this work was to exploit the optimization techniques applied along the traditional synthesis flow to simplify the structure of the evolved circuit and remove any redundant or unused portions (which may exist due to the stochastic characteristics of the evolution process). To fulfill the work goals, it was decided to create a description of the evolved circuits structure using an hardware description language. This choice allows reproducing the hierarchical structure of the individuals and permits to include the generated description in a portable IP-core that can be synthesized for a number of different devices and used in any custom design. Thus, each component in the hierarchy composing the individual has been implemented as a VHDL entity; the following paragraphs describe the VHDL modules.
1) **Combinatorial Cell:** the elementary block of the evolutionary system is a cell; it can implement a sequential or a combinatorial behavior; we are interested in the second one. In order to obtain such a behavior, the 16 configuration bits of the LUTG and of the LUTF must be the same. In this way a cell behaves simply as a 4bit input/1bit output lookup table, where the less significant bit of the configuration string ($CFG$) behaves simply as a 4bit input/1bit output lookup table, where the less significant bit of the configuration string ($CFG_0$) is triggered by an input of four ones, and the most significant ($CFG_{15}$), conversely, by an input of all zeros. Figure 4 more intuitively explains the functioning of a combinatorial cell.

![Combinatorial Cell](image)

The input bits ordered, from right to left, from the least significant ($I_{7..4}$) to the most significant ($I_0$). The configuration of the LUT (determined by the $CFG_{15..0}$ values) identifies the function of the LUT and, thus, the behavior of the cell and of the output bit $O_0$.

2) **Column:** each column (Figure 5) has an 8bit input data path coming from the previous column, and an 8bit output data path going as input into the next column. Since each of the eight cells contained into the column can have only 4 inputs, the 8 bits are split into two nibbles. The most significant nibble ($I_{7..4}$) represents the input for the cells 1, 3, 5, and 7, while the bits $I_{3..0}$ are the inputs for the remaining cells (0, 2, 4, and 6). The output bit of each cell is considered, in the right order, to create the output of the column, the output bit of the cell 7 being the most significant one and the output bit of the cell 0 the less significant one. This routing pattern is repeated for each column in the system.

3) **Module:** The connections inside a module are straightforward (see Figure 6), since the routing of the signals to the cells is taken care of within each column. Each module contains four columns and has an I/O data path of 8 bits. Note that, for the 8bit version, this component corresponds to the full individual.

4) **Individual:** For what concerns the whole individual, it is worth noting that the connection pattern is different from one column to another, being the same for the second and the fourth column of modules, but different for the third one. This is done to reflect the actual FPGA implementation. The 32 input bits are split into four group of 8 bits and are routed to the first column modules with the following schema: the eight most significant bits (from 0 to 7) are the inputs of the first module in the first column, the bits from 8 to 15 enters the second module, bits from 16 to 23 are the inputs for the third module, last the eight less significant bits (from 24 to 31) enter the last module of the first column.

The output of the first column represents the input of the second column. The first module of the second column has in input the two less significant bits of the previous column cells. In a similar way, the inputs of the second module are the two second less significant bits of the previous column cells, following the same pattern as before. The input bits of the third and of the fourth modules of the second column are made of the two second most significant bits and of the two most significant bits of the output of the previous column cells. The same pattern is followed in routing the connections between the third and the fourth column of modules. The routing of the signals between the second and the third column is similar to the one described in the previous paragraph. Finally, the output of the individual is the concatenation of the 8bit outputs of the last column modules, with the output of $Mod_{30}$ being the most significant bit of the individual output.

One of the issues that were faced during the implementation phase was how to organize the code so that it would become easy to dynamically modify the LUTs configuration according to the one of the evolved individual to export. A solution to this issue comes from the use of VHDL generic signals\(^1\), which have been used to aggregate the configuration of all the LUTs in an individual-wide signal that encodes the truth tables of all the LUTs. This signal is automatically generated by the

\(^1\)The VHDL sources are publicly available online [19] in a git repository and can be freely browsed for more details on the code.
dynamic part of the DGECS tool, which is described in the next Subsection.

B. LUTS configuration setup tool

The use of VHDL and, in particular, of the generic signals to implement the description of the static part of the evolved circuits makes the dynamic generation of a description of a specific individual a relatively simple task. In fact, it is enough to parse the output of the evolutionary algorithm (which contains the bitstream of the evolved circuit) and translate this information extracting the LUT configurations into the individual-wide configuration signal used in the VHDL description. The algorithm to realize this task has been implemented in Python, which ensures easy and quick operations on files and strings management.

To ease the use of the DGECS tool, a graphical interface has been also created, by using the pyQt bindings for the Qt cross-platform toolkit [25]; the main window of the tool is shown in Figure 7.

In the screenshot, it is possible to spot the main functions of the tool; the top part allows selecting the evolved circuit format (the 8bit and 32bit individual evolved on the Virtex-4 EHW system are available and the tool allows adding new individual types by simply extending an Individual class). The bottom part of the window permits to assign a name and a version string to the component to be created and then to export the VHDL description. By using the Generate VHDL button, the tool allows exporting the description of the individual, already embedded in a wrapper that implements a PLB-compliant component that can be plugged to any PLB-based design.

The central part of the window offers tools to manage the LUTs contents, permitting to load this information from a file (which must contain the final output of the EA) and to directly view and edit the LUTs configurations. This last feature is further illustrated with a screenshot shown in Figure 8.

V. EXPERIMENTAL RESULTS

To evaluate the functionality and the capabilities of the DGECS tool, we evolved several benchmark circuits with the HERA framework and then we exported them to different designs using DGECS.

For what concerns the 8bit individual, we evolved four circuits: a 8bit parity generator, a 4bit comparator, a 3bit full adder, and a 65% accurate data mining classifier [17] concerning the Acute Inflammation dataset [26]. Then we used DGECS to get their VHDL description, we synthesized them targeting the same Virtex-4 FPGA and we verified their I/O behavior: it perfectly corresponded to the one of the evolved circuits. This analysis proved that the DGECS tool is actually able to reliably export evolved circuits while maintaining their functionality. This experiment also allowed gathering some data on the resources requirements of the exported circuits; these results are reported in Table I. The Table also reports the occupation results of the HERA 8bit individual, that indicate the area that would be needed if the evolved circuits
requirements of the three circuits are listed in Table III.

The resources used in the synthesis flow focuses on area optimizations. We then verified and a Xilinx Spartan-6 XC6SLX100). Also in this case, the target circuits (a Xilinx Virtex-5 XC5VLX110T and a Xilinx Spartan-6 XC6SLX100). Also in this case, the synthesis flow focuses on area optimizations. We then verified and reused on devices different from the one used in the EHW system. To evaluate this aspect, the 32bit circuits used in the previous experiment were packed in IP-cores and synthesized on the same FPGA, and the IP-core used during the evolution process, showed a reduction in area usage. In a second experiment meant to prove the portability of the components created with DGECS, we synthesized the 8bit circuits considered in the first experiment targeting a Xilinx Virtex-5 chip and a Xilinx Spartan-6 chip. Experimental results shown that the circuits behavior is correctly maintained.

Another goal of this project was to bring portability to the evolved circuits by creating a description that could be reused within the same hard macro used for evolving them. This experiment is meant to show how exporting the evolved circuits with DGECS enables to benefit from both the structural simplification (the cells, being purely combinatorial, are simpler in the exported description) and the synthesis flow area optimization, yielding reduced resource requirements.

The experiment has been repeated using the 32bit individual and evolving more complex benchmark circuits: a 12bit parity generator, a 8bit full adder, a 10 bit comparator, a 100% accurate data mining classifier concerning the Acute Inflammation dataset [26] and a 99.9% accurate data mining classifier concerning the Cars Evaluation dataset [26]. Also in this case, we successfully performed the validation of their I/O behavior. Table II summarizes the gathered resource requirements.

The data reported in Table I and Table II show that the circuits behavior is correctly maintained. The work presented in this paper consists in the design and implementation of a tool allowing the creation of standalone components that provide the same functionalities of the circuits evolved through the HERA framework. The tool eases the use of such components on devices different from the one they were evolved on. The produced software, implemented in Python using the Qt graphic libraries, needs the device type and the individual structure to be selected first. Then, it allows describing the behavior of the circuit from scratch (directly writing the content of the lookup tables) or to import a file containing such information and modify it with an easy-to-use graphical interface.

To validate the tool, we performed two experiments. We first evolved combinatorial circuits (parity generators, comparators, full adders, and simple classifiers) using both the 8bit and the 32bit HERA individual on a Xilinx Virtex-4 evaluation board. For each of these circuits, we saved the content of the lookup tables or to import a file containing such information and modify it with an easy-to-use graphical interface.

Table II summarizes the gathered resource requirements.

Another goal of this project was to bring portability to the evolved circuits by creating a description that could be reused on devices different from the one used in the EHW system. To evaluate this aspect, the 32bit circuits used in the previous experiment were packed in IP-cores and synthesized targeting different FPGAs (a Xilinx Virtex-5 XC5VLX110T and a Xilinx Spartan-6 XC6SLX100). Also in this case, the synthesis flow focuses on area optimizations. We then verified these IP-cores against all the combinatorial inputs and their behavior corresponded to the expected one. The resources requirements of the three circuits are listed in Table III.

VI. CONCLUSIONS

The work presented in this paper consists in the design and implementation of a tool allowing the creation of standalone components that provide the same functionalities of the circuits evolved through the HERA framework. The tool eases the use of such components on devices different from the one they were evolved on. The produced software, implemented in Python using the Qt graphic libraries, needs the device type and the individual structure to be selected first. Then, it allows describing the behavior of the circuit from scratch (directly writing the content of the lookup tables) or to import a file containing such information and modify it with an easy-to-use graphical interface.

To validate the tool, we performed two experiments. We first evolved combinatorial circuits (parity generators, comparators, full adders, and simple classifiers) using both the 8bit and the 32bit HERA individual on a Xilinx Virtex-4 evaluation board. For each of these circuits, we saved the content of the LUTs and we then imported it into DGECS. A comparison between the resource requirements of the exported circuits, re-synthesized on the same FPGA, and the IP-core used during the evolution process, showed a reduction in area usage. In a second experiment meant to prove the portability of the components created with DGECS, we synthesized the 8bit circuits considered in the first experiment targeting a Xilinx Virtex-5 chip and a Xilinx Spartan-6 chip. Experimental results shown that the circuits behavior is correctly maintained.

It is possible to envision many future developments for the research line this paper belongs to; in addition to the resource occupation analysis already described, a timing analysis could be performed, trying to understand whether the standalone components allow, in general, to work at higher clock

---

**TABLE I**

FPGA resources needed by the DGECS-synthesized evolved circuits with respect to the HERA 8bit individual; to allow a comparison, the DGECS description has been synthesized on the same Virtex-4 FPGA used in the EHW system.

<table>
<thead>
<tr>
<th>Description</th>
<th>8bit HERA</th>
<th>8bit par. gen.</th>
<th>4bit comp.</th>
<th>3bit FA</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td># Slices</td>
<td>113</td>
<td>62 &lt; -45%</td>
<td>49 &lt; -57%</td>
<td>62 &lt; -45%</td>
<td>55 &lt; -51%</td>
</tr>
<tr>
<td># Slice FF</td>
<td>151</td>
<td>37 &lt; -75%</td>
<td>37 &lt; -75%</td>
<td>37 &lt; -75%</td>
<td>37 &lt; -75%</td>
</tr>
<tr>
<td># 4LUTS</td>
<td>180</td>
<td>84 &lt; -53%</td>
<td>59 &lt; -67%</td>
<td>85 &lt; -53%</td>
<td>72 &lt; -60%</td>
</tr>
</tbody>
</table>

Average resource savings: -61%

**TABLE II**

FPGA resources needed by the DGECS-synthesized evolved circuits with respect to the HERA 32bit individual; to allow a comparison, the DGECS description has been synthesized on the same Virtex-4 FPGA used in the EHW system.

<table>
<thead>
<tr>
<th>Description</th>
<th>32bit HERA</th>
<th>12bit par. gen.</th>
<th>8bit FA</th>
<th>10bit comp.</th>
<th>Infl. Class.</th>
<th>Cars Class.</th>
</tr>
</thead>
<tbody>
<tr>
<td># Slices</td>
<td>740</td>
<td>251 &lt; -66%</td>
<td>307 &lt; -59%</td>
<td>361 &lt; -51%</td>
<td>146 &lt; -80%</td>
<td>380 &lt; -49%</td>
</tr>
<tr>
<td># Slice FF</td>
<td>677</td>
<td>67 &lt; -90%</td>
<td>67 &lt; -90%</td>
<td>67 &lt; -90%</td>
<td>67 &lt; -90%</td>
<td>67 &lt; -90%</td>
</tr>
<tr>
<td># 4LUTS</td>
<td>1168</td>
<td>413 &lt; -65%</td>
<td>509 &lt; -56%</td>
<td>604 &lt; -48%</td>
<td>223 &lt; -81%</td>
<td>630 &lt; -46%</td>
</tr>
</tbody>
</table>

Average resource savings: -70%
cies. Another interesting point to be further investigated is the comparison between the features of the obtained components with the ones of specialized cores.

### REFERENCES


---

**TABLE III**

FPGA resources needed by the 32bit components synthesized on different devices (A XILINX VIRTEX-5 XC5VLX110T and a XILINX SPARTAN-6 XC6SLX100)

<table>
<thead>
<tr>
<th></th>
<th>12bit par. gen.</th>
<th>8bit FA</th>
<th>10bit comp.</th>
<th>Infl. Class.</th>
<th>Cars Class.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtex5</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Slices</td>
<td>153</td>
<td>208</td>
<td>233</td>
<td>95</td>
<td>195</td>
</tr>
<tr>
<td># Slice FF</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td># 6LUTS</td>
<td>330</td>
<td>431</td>
<td>552</td>
<td>157</td>
<td>551</td>
</tr>
<tr>
<td><strong>Spartan6</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Slices</td>
<td>86</td>
<td>115</td>
<td>141</td>
<td>58</td>
<td>139</td>
</tr>
<tr>
<td># Slice FF</td>
<td>76</td>
<td>76</td>
<td>76</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td># 6LUTS</td>
<td>267</td>
<td>357</td>
<td>455</td>
<td>142</td>
<td>414</td>
</tr>
</tbody>
</table>