SN Computer Science (2022) 3:188
https://doi.org/10.1007/s42979-022-01045-9

ORIGINAL RESEARCH

Grammatical Evolution of Complex Digital Circuits in SystemVerilog

Michael Tetteh[1] · Douglas Mota Dias[1,2] · Conor Ryan[1]

Received: 22 July 2021 / Accepted: 21 January 2022 / Published online: 12 March 2022

Abstract

The evolution of complex circuits remains a challenge for the Evolvable Hardware field in spite much effort. There are two major issues: the amount of testing required and the low evolvability of representation structures to handle complex circuitry, at least partially due to the destructive effects of genetic operators. A 64-bit × 64-bit add-shift multiplier circuit modelled at register-transfer level in SystemVerilog would require approximately 33,200 gates when synthesized using Yosys Open SYnthesis Suite tool. This enormous gate count makes evolving such a circuit at the gate-level difficult. We use Grammatical Evolution (GE) and SystemVerilog, a hardware description language (HDL), to evolve fully functional parameterized Adder, Multiplier, Selective Parity and Up–Down Counter circuits at a more abstract level other than gate level—register transfer level. Parameterized modules have the additional benefit of not requiring a re-run of evolutionary experiments if multiple instances with different input sizes are required. For example, a 64-bit × 64-bit and 128-bit × 128-bit multipliers etc., can be instantiated from a fully evolved functional and parameterized N-bit × N-bit multiplier. The Adder (6.4×), Multiplier (10.7×) and Selective Parity (6.7×) circuits are substantially larger than the current state of the art for evolutionary approaches. We are able to scale so dramatically because of the use of a HDL, which permits us to operate at a register-transfer level. Furthermore, we adopt a well known technique for reducing testing from digital circuit design known as corner case testing. Skilled circuit designers rely on this to avoid time-consuming exhaustive testing. We demonstrate a simple way to identify and use corner cases for evolutionary testing and show that it enables the generation of massively complex circuits. All circuits were successfully evolved without resorting to the use of any standard decomposition methods, due to our ability to use programming constructs and operators available in SystemVerilog.

Keywords Grammatical evolution · Evolvable hardware · Evolutionary design of conventional circuits · Hardware description language (HDL) · Verilog · SystemVerilog · Register transfer level (RTL) · Combinational circuits · Sequential circuits · Corner case testing

This article is part of the topical collection “Genetic Programming” guest edited by Aniko Ekart, Ting Hu and Nuno Lourenço.

The authors are supported by Research Grant 16/IA/4605 from the Science Foundation Ireland and by Lero, the Irish Software Engineering Research Centre. The second author is partially financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brasil (CAPES)—Finance Code 001, and FAPERJ.

✉ Michael Tetteh
michael.tetteh@ul.ie

Douglas Mota Dias
douglas.motadias@ul.ie; douglas.dias@uerj.br

Conor Ryan
conor.ryan@ul.ie

[1] University of Limerick, Limerick, Ireland

[2] Rio de Janeiro State University, Rio de Janeiro, Brazil

Introduction

Researchers began to explore the feasibility of the application of evolutionary algorithms (EAs) to circuit design tasks back in the early 1990s [2–4], which gave birth to the evolvable hardware (EHW) field. The choice of EAs for circuit evolution is motivated by the fact that EAs were demonstrated to be better and more creative at exploring the circuit design search space than humans [5].

EHW is made up two main sub domains: adaptive hardware and evolutionary design of conventional digital circuits [6]. The latter, adaptive hardware, refers to the continuous and autonomous reconfiguration or adaptation of evolved hardware to conform to changing operational requirements or conditions of its deployed environment over its lifespan [6, 7]. On the other hand, the latter deals with the design of conventional digital circuits. EAs such as Genetic Algorithm, Genetic Programming, Cartesian Genetic Programming (CGP), Grammatical Evolution (GE) etc. have been applied to circuit design tasks.

The ultimate objective of EHW is the development of next generation hardware that is capable of self-reconfiguration through evolution [6, 8]. However, a major and basic requirement of an adaptive hardware system is the ability to evolve its circuitry. A major reason for the extreme importance of adaptive systems is their suitability for use in environments characterized by extreme conditions (such as extreme temperatures, high radiations etc.), which are unsafe and inaccessible to humans to operate. A typical application area is space exploration [9–11].

In spite of the immense progress made in the EHW field since its inception, issues of scalability have long hampered its progress. This is because existing methods mostly design circuits at the gate-level using EAs. A major benefit of gate-level evolution is its capability to evolve optimized novel circuit designs (i.e. circuits using fewer gates) compared to human designed circuits. For example, a 3-bit $\times$ 3-bit and a 4-bit $\times$ 4-bit multiplier evolved in [12] were 23.3% and 10.9% more efficient in terms of the number of two input logic gates used respectively. However, the drawbacks of gate-level evolution are: inability to scale to evolve complex circuits [13], designs are less interpretable, difficult and take more time to verify etc. These drawbacks account for the two major issues confronting EHW.

The two major issues are: Scalability of Representation and Scalability of Fitness Evaluation [14]. As circuit inputs increases, longer chromosomes are required to represent circuits. This results in large search spaces which are difficult, slow and expensive to search [14], reducing the chances of EAs finding novel circuit designs. Furthermore, due to the destructive nature of genetic operators, evolving circuits of such complexity becomes intractable. This is termed as Scalability of Representation. Also, increasing circuit inputs increases the number of test cases [15]. As a result, exhaustive testing is only feasible for modestly complex circuits — circuits with a few number of inputs and reasonable number of gates that can be exhaustively tested. This phenomenon is termed Scalability of Fitness Evaluation.

Conventionally, hardware engineers design circuits either through the use of schematic diagrams or Hardware Description Languages (HDLs). Schematic diagrams involve the use of graphical representations of circuit components to design circuits. Schematic diagrams reveal the functionality and connections between the various components used to realize the circuit functionality. HDLs are programming languages used for the behavioural description of circuits. The two dominant HDLs in industry today are Verilog/SystemVerilog (SV) and Very High-Speed Integrated Circuit Hardware Description Language (VHDL).

HDLs are well suited and ideal for large and complex circuits designs compared to schematic diagrams [16, 17]. The use of HDLs is the *de facto* standard in the design of digital circuits in industry due to a number of reasons: first, real world digital circuit designs are usually intractable to design at the gate-level; second, more abstract designs (such as RTL) are more interpretable, require relatively less time to vary/adapt and easier to verify [20]; third, they enable design module reuse through parameterization and support for mixed-style design in a single module; fourth, HDL designs are portable across different hardware because they can be synthesized for several vendor-specific Field Programmable Gate Arrays (FPGAs) [16] , unlike other approaches which evolve circuits directly on a specific FPGA, such as Xilinx XC6200 series, by evolving bitstreams that configure the FPGA [21]. These approaches stalled after Xilinx Corporation discontinued the production of such FPGAs due to cost and security reasons [21].

HDLs support digital circuit designs at different levels of abstraction. For example, in SystemVerilog, circuits can be designed as low as switch-level, gate-level, register-transfer level (RTL) and as high as behavioural level. To date, the most complex fully functional circuits evolved are: a 10-bit + 10-bit adder [22], a 6-bit $\times$ 6-bit multiplier [15], a 19-bit parity circuit [23] and a 28-input frg1 circuit [24].

In this paper we seek to answer the following research questions:

1. Can the use of HDLs (SystemVerilog in this case) evolve complex combinational and sequential circuits that compare favourably to state-of-the-art methods in terms of the size of inputs and gate count?
2. What are the drawbacks of evolving circuits using GE and HDLs?

This paper is an extension of the work published under the title: “Evolution of Complex Combinational Logic Circuits Using Grammatical Evolution with SystemVerilog” in European Conference on Genetic Programming 2021 [25]. The initial paper examined the concept of evolving complex and parameterized combinational circuits in SystemVerilog using GE with behavioural constructs. Additionally, corner case testing was employed to significantly reduce the amount of testing, since exhaustive testing is not feasible for complex circuits. This paper builds upon this work by

- Evolving a parameterized sequential circuit—Mod-N Up-Down Counter;
- Synthesizing and comparing of evolved parameterized circuits to the state-of-the-art works;
- Investigating the effect of varying population sizes on evolutionary performance;

– Performing statistical analysis on the effect of input bit-width sizes on simulation time, to determine appropriate input bit-width sizes for evolving parameterized circuits.

In summary, this paper tackles the issues of scalability through the use of SystemVerilog with GE to design fully functional and complex combinational circuits. This is achieved without the use of decomposition. The main contributions of this paper are:

1. Evolution of fully complex functional combinational digital circuits: 64×64-bit multiplier, 64 + 64-bit adder, 128-bit selective parity circuit and 64-bit Up–Down Counter;
2. All evolved circuits are parameterized. Thus, these design modules are reusable and only require the specification of input and output bit-width sizes without the need for re-run of experiments. The bit-width sizes mentioned in (1) above are the default bit-width sizes for the evolved circuits;
3. A significant reduction in the amount of testing through appropriate choice of operators and the combination of corner case and randomized testing in testbenches. Specifically, 3.40×10^{36}, 6.80×10^{36}, $2(6.80 \times 10^{36})$ and 2.09×10^{38} test cases reduction for Adder, Multiplier, Selective Parity and Mod-N Up–Down Counter circuits respectively.

Background

There are two main types of digital circuits, namely, combinational and sequential. Combinational circuits are a class of circuits whose output(s) depends solely on current inputs. Sequential circuits on the other hand, are a class of circuits whose output(s) depends on both current and past input(s). Sequential circuits consist of memory elements such as flip-flops which store the output of combinational logic which are then fed back as input. The vast majority of circuits evolved in EHW literature are combinational. This is partly due to complex circuit connections and unavailability of appropriate encoding structures for chromosomes that model feedback loops of sequential circuits [13]. For example, in [26], CGP was modified to allow levels forward in order to model feedback loops of sequential circuits.

Cartesian Genetic Programming (CGP) is a GP variant widely used in EHW [27]. CGP uses direct acyclic graphs to represent its phenotype [28]. A CGP genotype consist of function, connection and output genes which encode a node's operation, its source of input(s) and the program output locations respectively. CGP is a well known technique for the evolution of digital circuits and much interesting progress has been made since its inception [29]. CGP has not only been used to evolve a range of fully functional circuits at the gate-level, but also at the functional-level. It has also been applied to the design of approximate circuits [30]. Approximate circuits is a category of circuits which trades off hundred percent functionality for other equally important circuit parameters such as area, power consumption etc.

Grammatical Evolution

GE is a grammar-based GP variant which evolves programs in any Backus-Naur Form compliant language [31]. GE uses a mapper which employs a linear genome consisting of integer codons and a grammar describing a subset of the target language to generate programs. A modulo rule is used to dictate production rule selection. The process expands all non-terminals to terminal symbols. A valid program or phenotype is obtained once all non-terminals have been expanded to terminals. GE has been applied to several domain problems such as circuit design [32], symbolic regression [33], Explainable AI [34], Software Testing [35] etc.

Despite its success in many application domains, GE hasn't been applied often to circuit synthesis problems, although the availability of HDLs makes it an ideal tool for circuit design tasks. An initial exploration of this idea evolved a 1-bit full adder using GE in [36]. Other simple circuits have also been evolved at the gate level [37]. Both studies did so using a HDL. In [32], GE was used to design circuits such as 11-bit Multiplexer, Seven Segment Display and Hamming Code (7,4) Decoder using SystemVerilog. The work also explored the effect of grammar design with and without the incorporation of simple domain knowledge to guide the evolutionary search towards regions of the solution space where optimal solutions are perceived to be.

SystemVerilog

SystemVerilog is one of the major HDLs used in industry for the design of digital circuits. SystemVerilog is an extension of Verilog—essentially, a superset of Verilog. The extension was based on two objectives: efficient and more accurate modeling of digital circuits functionality; writing of efficient and race free verification code for large and complex designs [20]. Digital circuit design in SystemVerilog can be performed at four main abstraction levels, in decreasing order of abstraction: behavioural level, register-transfer level (RTL), gate level and digital switch level [20]. The higher the abstraction level, the less detail it reveals about the actual circuit representation.

Digital switch level deals with the design of circuits at the transistor level, while gate-level modelling involves the use of only primitive logic gates such as: `and`, `or`, `nor`, `xor`, `not` etc. to design circuits. RTL designs make use of two primary constructs: `continuous assignments` and `always` procedural blocks [20]. A statement preceded by the `assign` keyword is a continuous assignment. For example, `assign out = inA & inB;`. The output signal (`out`) is continuously driven by the right hand expression whenever any of the operands or signals change, triggering a re-evaluation of the right hand expression. RTL designs obscure circuit functionality realization in silicon which are only revealed after synthesis [20]. In comparison to gate-level designs, RTL designs are interpretable, more capable of dealing with huge input sizes, better suited for verification of large and complex designs [20] etc. Behavioural level models circuits at a higher abstraction level than RTL, utilizing all constructs in SystemVerilog and therefore may not be synthesizable.

SystemVerilog Instructions

An `always` procedural block can be used to model combinational and sequential logic [38]. An `always` block behaves like an infinite loop that continuously executes statements within its block, except it is triggered either by a time or event control. With time control, the `always` block repeatedly executes each time the specified delay time elapses. For example, an `always` block that toggles a clock (clk) signal after every ten time units will be described in SystemVerilog as $\texttt{always\#10clk} = \sim \texttt{clk;}$. Note, the time delay syntax is $\#\langle\texttt{time} - \texttt{unit}\rangle$. An always block with event control requires a *sensitivity list*. A sensitivity list is a list of signals that trigger the execution of the `always` block whenever any of the signal changes. For example, $\texttt{always@(signalA, signalB)begin}\langle\texttt{stmts}\rangle\texttt{end}$ and $\texttt{always@(posedgeclk)begin}\langle\texttt{stmts}\rangle\texttt{end}$ for combinational and sequential logic modelling respectively. The syntax for sensitivity list specification is: $\texttt{@(}\langle\texttt{sensitivity} - \texttt{list}\rangle\texttt{)}$. SystemVerilog supports other specialization of the `always` block for specific use cases [38].

SystemVerilog has various categories of operators: bitwise, reduction, conditional, arithmetic, logical operators etc. Another very useful SystemVerilog programming construct which we make use of in this work is `generate for-loop` (synthesizable `for-loop`), which is useful for the description of circuits with a fixed and repetitive structure. The `generate for-loop` instruction creates *n* module instances which can be specified through SystemVerilog parameters. SystemVerilog also has `switch-case` and `if-else` constructs, which are either synthesized to multiplexers or priority encoders by synthesis compilers based on mutual exclusivity of the selection items [38]. However, there exist modifiers to use in conjunction with `switch-case` and `if-else` statements to reduce ambiguities and ease the task of synthesis compilers [39].

Related Work

Research conducted to address scalability issues in evolutionary design of digital circuits has followed three main trajectories: Functional-Level Evolution, Increasing evolvability through the improvement of genetic operators and Problem decomposition.

Functional-Level Evolution

Functional-Level Evolution (FLE) uses secondary logic functions such as adders, multiplexers, multipliers etc. (usually in conjunction with primitive logic gates) other than only primitive logic gates in designing digital circuits [3].

In [40], half-adder, full adder and multiplier circuits were used as functions to design adders and multipliers. In comparison to gate-level evolution, function-level evolution obtained higher success rate as well as significant reduction in the number of generations in some instances [40]. Three-bit multipliers were evolved using binary multipexers in [41] and a proposed Constrained CGP evolved higher order approximate multipliers using imprecise lower order multipliers [42]. Approximate 9-input and 25-input median circuits [43], and an image filter [44] have also been evolved at functional-level.

Increasing Evolvability Through The Improvement of Genetic Operators

Evolutionary performance is largely dependent on the evolvability of genetic operators. Thus the ability of genetic operators to create offsprings with better or improved fitnesses than their parents. The traditional CGP setup uses point mutation as its variation operator. This incurs wasted evaluations and stalls evolution when point mutation operations affect only inactive genes. Single Active Mutation (SAM), unlike point mutation, ensures an active gene is mutated whenever mutation takes place [45].

Biased SAM is an improvement upon SAM, which increases its robustness when tackling combinational circuit designs. Biased SAM requires a transition probability matrix to work with. This is constructed for each problem by conducting preliminary experiments on the problem. During these initial experiments, functional mutations that increase the overall fitness of an individual are recorded. Subsequently, during the actual experiments, roulette wheel selection is applied on the transition probability matrix to determine the function the chosen functional node should be mutated to [45]. On selected circuit design benchmarks, Biased SAM outperformed SAM.

Guided Active Mutation (GAM), another CGP mutation operator was designed to mutate active node (s) in a subgraph corresponding to a single output [46]. The objective was to increase the overall fitness score of this output when compared to the truth table of the circuit. However, though GAM recorded fewer fitness evaluations to find feasible solutions, it recorded low success rates. As a result, SAM and GAM were merged in order to harness the strengths of both mutation operators. Their combination resulted in high success rates and best results on the selected benchmark problems [46].

Semantically-oriented mutation operator (SOMO) performs mutation in the phenotypic space after which the resultant phenotype is encoded to its corresponding genotype [22]. SOMO randomly selects an active node for mutation. A mutation operation may affect a node function or a node input connection. During node input connection mutation, the best connection point that results in an increase of individual's fitness is determined through the application of semantics [22]. SOMO evolved combinational circuits significantly faster compared to the multi-threaded parallel CGP implementation in [47].

Circuit Decomposition Methodologies

Several methodologies have been proposed to decompose complex circuit problems into subcircuits of evolvable complexity. These fall into two main categories: inputs and outputs decomposition. The former refers to decomposition methods that breakdown circuits into subcircuits using only the circuit's inputs while the latter uses the circuit's outputs.

Increased complexity evolution (ICE), also referred to as divide and conquer methodology, breaks a complex system into sub-systems of evolvable complexity before evolution is performed on each sub-system in a bottom–up manner [48, 49]. Sub-systems can be evolved in parallel (if no dependency exists between them) or sequentially. These evolved sub-systems serve as building blocks in a more complex sub-system(s) in an upper layer. One challenge with ICE is that it requires the fitness functions for the respective sub-systems to be manually defined. However, for certain problems such as circuit design, a solution to this challenge is either the partitioning of training vectors based on circuit outputs or the training set can be partitioned [48, 49].

In contrast to ICE, Bidirectional Incremental Evolution (BIE) performs automatic decomposition of a complex circuit into subsystems using either a suitable standard decomposition technique (such as Shannon's theorem) or an EHW-oriented decomposition technique; or a combination of both [50]. BIE progressively decomposes complex circuits into subcircuits in a top-down manner while evolving each subsystem. Subsystem(s) undergo further decomposition only if evolution is unable to obtain optimal solutions. During the second phase of BIE, evolved subsystems are merged and optimized in a bottom–up fashion. In comparison to direction evolution, BIE performed best on 7-inputs and 10-output circuit benchmark problems.

On more complex circuits, BIE performed poorly [15]. As a result, instead of output decomposition in the case of BIE, an input decomposition technique termed Generalized Disjunction Decomposition (GDD) was proposed in [15]. GDD automatically decomposes a complex circuit into two subcircuits: the evolvable and multiplexer subcircuits [15]. GDD requires the user to specify the number of inputs for the evolvable circuit, subject to the constraint that it must be less than the number of inputs of the original circuit. GDD then applies an algorithm to generate the evolvable subcircuit's outputs using the actual circuit's truth table. The evolvable subcircuit can be evolved using any EHW methods (such IEC or BIE) without any restrictions. The multiplexer subcircuit takes the outputs of the evolvable subcircuit as inputs and uses the remaining inputs of the actual circuit not used in the evolvable circuit as select lines for the multiplexer. The output of the multiplexer circuit is the same as that of the complete (original) circuit. Using a selection of circuit benchmark problems from Microelectronics Center of North Carolina (MCNC) benchmark suite and randomly generated circuit problems, compared to BIE, GDD required fewer number of generations, time and better fitness scores [15].

Stepwise Dimension Reduction (SDR) is a layered and output decomposition methodology that decomposes a circuit into two subcircuits. One of these is evolved to handle input combinations with an output value of 1 while the second is evolved to handle those with 0 as output [23]. Internal intermediate mappings are devised if subcircuits are not of evolvable complexity. SDR evolved a 19-bit circuit which GDD was unable to evolve. SDR obtained comparable results for few of the benchmark problems in less time compared to GDD, but not on more complex multiple output circuit benchmarks [23]. SDR is effective on single output circuit problems such as the parity circuit.

Common to all these approaches is that they are capable of evolving relatively complex circuits as it is feasible to first evolve subcircuits before merging them into a complete circuit with a high input dimension.

Other Scalability Approaches

In other works such as [47], multi-threaded parallel implementations of evolutionary algorithms, specifically CGP, have been developed to exploit modern processor architectures to allow for more fitness evaluations. However, just a single run was performed to evolve 5-bit $\times$ 5-bit multiplier due to the associated high computational effort [47]. In [24], CGP candidate circuits are transformed into a Binary Decision Diagram (BDD). Similarly, the circuit specification is also specified in BDD format. A BDD evaluation

Table 1 Summary of some proposed approaches to address scalability challenges in EHW

Approach		Authors	Circuit Type	Circuits Evolved
Functional Level Evolution	GA + FPGA + seven functions: adder, subtracter, if-then, sine generator, cosine generator, multiplier and a divider [1]	M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, T. Higuchi	Combinational	Classification of iris data, transformation of 2D images and distinguishing two intertwined spirals.
	Constrained CGP + Using lower order approximate mulltipliers to evolve higher order approximate multipliers [2]	K. Senthilkumar, K. Kumarasamy, V. Dhandapani		Evolution of approximate 4-bit×4-bit, 8-bit×8-bit, and 16-bit×16-bit multipliers using evolved lower order multipliers: 2-bit×2-bit, 3-bit×3-bit and 3-bit×2-bit multipliers.
Improved Genetic Operators	Biased Single Active Mutation [3]	F.A.L. Manfrini, H.S. Bernardino, H.J.C. Barbosa		Evolution of 3-bit parity, 16-to-4 bit encoder, 4-to-16 decoder and 3-bit multiplier.
	Guided Active Mutation and Single Active Mutation [4]	J.E.H. da Silva, L.A.M. de Souza, H.S. Bernardino		6-bit + 6-bit Adder and other combinational logic benchmark circuits.
	Semantically-oriented Mutation Operator [5]	D. Hodan, V. Mrazek, Z. Vasicek		2-bit+2bit to 10-bit+10-bit adders, 2-bit × 2-bit to 5-bit × 5-bit Multipliers and 5-bit to 10-bit parity circuits.
Problem Decomposition	Divide and Conquer Combinational	J. Torresen		2 and 4 characters recognition system [6], and Prostetic Hand· Controller [7]
	Bi-directional Incremental Evolution [8]	T. Kalganova		7-input 10-output logic function (z5xp1 d.pla)
	Generalized Disjunction Decomposition [9]	E. Stomeo, T. Kalganova, C. Lambert		Randomly generated circuits, 3-bit to 6-bit multiplier circuits, 13-bit t0 17-bit even parity circuits, some selected cir-·cuits from MCNC library [10]. 4-bit to 6-bit adders and multipliers.
	Stepwise Dimension Reduction [11]	Z. Li, W. Luo, X. Wang		10-bit to 19-bit parity circuits, 4-bit×4-bit, 5-bit×5-bit multipliers and some selected circuit problems from MCNC library.
Multi-thread and parallel implementation of CGP [12]		R. Hrbacek, L. Sekanina		6-bit+6-bit to 9-bit+9-bit adders and 2-bit × 2-bit to 5-bit × 5-bit multipliers.

is performed, where functional similarity is performed between the candidate circuit's BDD representation and circuit's specification also in BDD format. The fitness of the candidate circuit is the hamming distance between the output vectors of the BDD representations of the candidate circuit and circuit specification. Results reveal that BDD-based evaluation is faster than exhaustive testing [24] (Table 1).

Experimental Design

We evolve four complex circuits: three combinational and one sequential circuit. The combinational circuits are: multiplier, adder and selective parity generator. The chosen sequential circuit is Up–Down Counter. These problems are representative of evolutionary circuit design benchmark problems in current literature [22, 23, 26, 47]. All circuits are evolved as parameterized design modules to accommodate all possible range of values of the parameters. Thus, evolved parameterized circuit modules retain their functionality for all possible range of values of the specific parameter(s). For example, an evolved optimal parameterized adder (N-bit + N-bit adder) must function perfectly as 8-bit + 8-bit adder, 32-bit + 32-bit adder etc. Prior to simulation or synthesis, parameter values of evolved parameterized circuits can be specified without requiring a re-run of evolutionary experiments, as is the case for existing EHW methodologies.

Parameterized circuits are designed using the keyword `parameter` in SystemVerilog. The usage of the parameter keyword is highlighted in the corresponding production of

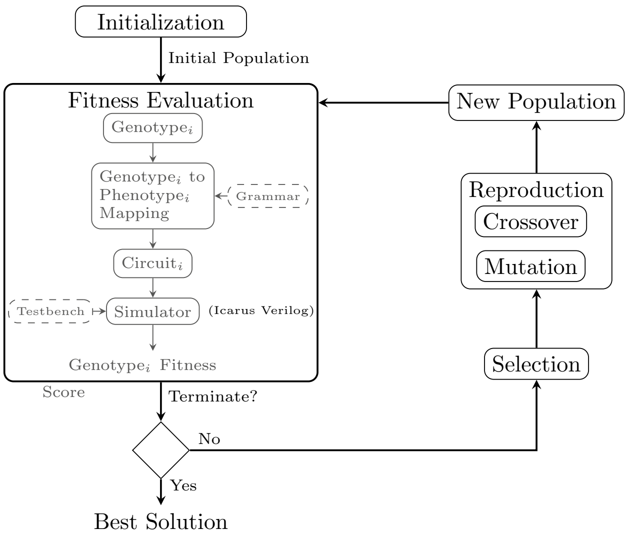

Fig. 1 GE functional simulation circuit design process

$\langle design - module\rangle$ rule for all benchmark circuit grammars. The grammars are divided into two segments separated by a dashed line. The top segment consist of rules that require expansion. In other words, the evolvable part of the grammar. The bottom segment (fixed grammar par) consist of rules that describe the corresponding circuit's interface, variable/register declaration and initialization, required programming constructs (e.g. `always` block, `generate for-loop`) etc. The bottom segment can be left out of the grammar as all the rules consist of single productions. Alternatively, a circuit module template can be designed with a placeholder to represent the evolved segment.

The GE circuit design process illustrated in Figure 1 is summarized as follows:

1. A randomly generated initial population is created.
2. For each individual in the population, GE mapper uses the specified subset grammar defined for the problem to derive the HDL code (phenotype). The phenotype (resultant circuit) is tested using the simulator (Icarus Verilog). The simulator uses the testbench to test the functionality of the evolved circuits. The total number of training cases passed by an individual represents its fitness score.
3. The termination criterion is checked to determine whether to terminate the experimental run. A termination criterion can be the maximum number of generations or fitness evaluations. If the termination criterion is met, the experimental run is terminated and the individual with the highest aggregated fitness score is returned as the best solution.
4. However, if the termination criterion has not been satisfied, individuals undergo a selection process. Usually, individuals with high fitness scores are chosen for reproduction.
5. The selected individuals undergo crossover and mutation to create offsprings that form a new population.
6. The new population of individuals undergo fitness evaluation (back to step 2).
7. The evolutionary cycle continues until the termination criterion is met and the best candidate solution returned.

Benchmark Problems

Selective Parity Circuit

A parity generator circuit generates a single bit (either 0 or 1) from a string of bits in order to obtain an even or odd number of 1s. An odd parity circuit generates a bit value of 1 and 0 when the data to be transmitted contains an even and odd number of 1s respectively. Conversely, an even parity circuit generates a 0 and 1 when the data to transmitted contains an even and odd number of 1s.

We design two different parity generator circuit grammars: Parity Grammar A and Parity Grammar B, shown in Listings 1 and 2 respectively. The purpose of designing two separate grammars using different operators is to investigate the benefit of using tuned operators in grammars. Selective Parity Grammar A uses generate loops (synthesizable $\langle for - loop\rangle$) and bitwise operators ($\langle bitwise - op\rangle$) to loop through the data bits while performing bitwise operations until a parity bit is obtained. Selective Parity Grammar B uses both reduction operators ($\langle reduction - op\rangle$) and bitwise-operators ($\langle bitwise - op\rangle$). Reduction operators apply bitwise operations on the bits of an $n - bit$ operand (a vector) recursively to produce a scalar output.

```
        <expr> ::= <select-bit> | <select-bit><bitwise-op><expr> | ~(<expr>)
  <select-bit> ::= intmd_parity[i] | data[i]
  <bitwise-op> ::= ^ | & | "|" | ~^
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<design-module> ::= <begin-module><declarations><initialize><for-loop><newline><output>
                    <end-module>
     <for-loop> ::= "for(i=0; i<width; i=i+1)"begin <newline> <stmt> end <newline>
         <stmt> ::= assign intmd_parity[i+1]"="<expr>;<newline>
       <output> ::= assign parity "=" intmd_parity[width]; <newline>
   <initialize> ::= assign parity "=" even_odd;<newline>
 <declarations> ::= "logic [width:0] intmd_parity;" <newline> "genvar i;" <newline>
 <begin-module> ::= "module parity #(parameter width=128) (input logic [width-1:0] data,
                    input logic even_odd, output logic parity);" <newline>
   <end-module> ::= endmodule
      <newline> ::= "\n"
```

```
             <expr> ::= <reduct-expr> | (<reduct-expr><bitwise-op><reduct-expr>)
                      | (<reduct-expr><bitwise-op><expr>)
      <reduct-expr> ::= <reduction-op><input> | <reduction-op>(<reduct-expr>
                      | (<reduction-op>{<multi-concat-expr>})
<multi-concat-expr> ::= <input> | <reduct-expr> | <multi-concat-expr>,<multi-concat-expr>
     <reduction-op> ::= & | ~& | "|" | ~"|" | ^ | ~^
       <bitwise-op> ::= ^ | & | "|" | ~^
            <input> ::= data | even_odd
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    <design-module> ::= <begin-module> <code-block> <end-module>
       <code-block> ::= <always-block><newline>
     <always-block> ::= always"@(data or even_odd)"<newline><stmt>
             <stmt> ::= parity "=" <expr>;
     <begin-module> ::= "module parity #(parameter width=128) (input  logic[width-1:0]
                        data, input logic even_odd, output logic parity);"<newline>
       <end-module> ::= endmodule
```

N-bit + N-bit Adder

An adder circuit performs addition in digital electronic devices. The adder grammar is shown in Listing 3 and uses the `always` (⟨*always* − *block*⟩) procedural block, which takes a sensitivity list as arguments and executes the statements within its code-block whenever a signal within the list changes. The operators used are binary arithmetic operator (+) and bitwise operators (&, |, ^). The default input bit-width of the addends is 128 specified using the parameter keyword in the ⟨*begin* − *module*⟩ rule.

```
          <io> ::= a | b | carry in
        <expr> ::= <io> | <io><op>(<expr>) | ~(<expr>)
          <op> ::= + | & | "|" | ^
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<design-module> ::= <begin-module><always-block><end-module>
 <always-block> ::= always @(a, b or carry in)begin<newline><stmt><end><newline>
   <sum-output> ::= {carry out, sum} "=" <expr>;<newline>
 <begin-module> ::= "module adder  #(parameter width=128) (input logic [width-1:0] a, b,
                    input logic carry in, output logic carry out,
                    output logic [width-1:0] sum);" <newline>
   <end-module> ::= endmodule
      <newline> ::= "\n"
```

```
 <if-else> ::= if(<condition>)<nextline> begin <nextline> product "=" <expr>;
               <next-line> end
             | if(<condition>)<nextline> begin <nextline> product "=" <expr>;
               <next-line> end <nextline> else <nextline> begin <nextline>
               product "=" <expr>; <nextline>
    <expr> ::= <io> | <io><op>(<expr>)
      <op> ::= + | << | >> | & | "|"
      <io> ::= multiplier | multiplicand | product | i
     <bit> ::= 1'b0 | 1'b1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<design-module> ::= <begin-module><declarations><always><end-module>
       <always> ::= always @(multiplicand or multiplier)<nextline> begin <nextline>
     <for-loop> ::= "for(i=0; i<i_width; i=i+1)" begin <nextline><if-else><nextline>
                    end <nextline>
    <condition> ::= <io>[i]"=="<bit>
 <declarations> ::= integer<loop-var>;<nextline>
 <begin-module> ::= module mult  #(parameter i_width=64,o_width=128) (input logic
                    [i_width-1":"0] multiplier, multiplicand, output logic
                    [o_width-1":"0] product); <nextline>
   <end-module> ::= endmodule
     <nextline> ::= "\n"
```

Table 2 N-bit Up–Down counter transition table

Clk	Reset	Enable_Load	Enable_Up	Load	Up_Down	Q (Output)
↑	0	?	?	?	?	0
↑	1	1	?	N-bit Number	?	Load
↑	1	0	1	?	1	Q + 1 (count up)
↑	1	0	1	?	0	Q - 1 (count down)
↑	1	0	0	?	?	Q (no change)
↓	?	?	?	?	?	?

The symbols ↑ and ↓ represent positive and negative transitions of the clock respectively. The ? symbol represents don't care bits

N-bit × N-bit Multiplier

A multiplier performs multiplication in digital electronic devices such as computers, calculators, etc. The particular multiplier type considered here is the Add-Shift Multiplier. The Add-Shift Multiplier's operation is based on longhand multiplication. Each digit of the multiplier multiplies the multiplicand to obtain an intermediate product shifted a digit to the left of the preceding intermediate product. All intermediate products are then summed up to obtain the product of the multiplication. The Multiplier Grammar is shown in Listing 4. The grammar makes use of `always` ($\langle always - block\rangle$), `for-loop` ($\langle for - loop\rangle$) and `if-else` ($\langle if - else\rangle$) programming constructs in SystemVerilog. Three different operators ($\langle op\rangle$) are used: *binary arithmetic operator* (+), shift operators (<<, >>) and bitwise operators (&, |).

Table 3 Evolutionary run parameters

Parameter	Value	
Initialization	Sensible Initialization	
Selection	Lexicase Parent Selection	
Crossover Rate	0.8	
Mutation Rate	0.01	
Replacement Rate	0.05	
Number of Runs	50	
Generations	30	Parity and Adder
	50	Multiplier and N-bit Counter
Population Size	200	Parity
	500	Adder
	2000	Multiplier and N-bit Counter
Termination Condition	When mean, minimum and maximum fitness equals the maximum fitness or generation number equal specified generation number	

In bold are the default population sizes used for the actual experiments

N-bit Up–Down Counter

Counters are sequential logic devices that store the number of occurrences of an event, usually from a pre-defined state. The counting operation is triggered by a clock signal. A counter that increases its content after every clock cycle is known as an up-counter, while one that decreases its content is referred to as down-counter. A bi-directional counter functions as an up and down counter.

An N-bit Up–Down Counter is a counter that functions as a mod-2^n Up-Down Counter. That is, it can function as a mod- 2, 4, 8, 16, 32 etc. Up–Down Counters. The transition table for N-bit Up-Down Counter is captured in Table 2. The counter has an active low-reset signal; thus it resets to State 0 when the reset signal is 0. The Enable_Load input triggers a specified input value to be loaded by the counter. The Enable_Up and Up–Down inputs control the type of counting operation to be performed by the counter. If both inputs are high, the counter behaves as an up-counter. If Up–Down input is high or 1 and the Enable_Up input is low or 0, the counter behaves as a down-counter. The grammar designed for the N-bit Up–Down Counter is shown in Listing 5.

Testing

The selected default bit-width sizes of the parameterized circuits are high, rendering exhaustive testing infeasible. Furthermore, lexicase selection, our preferred selection method for digital circuit evolution, becomes computationally expensive with increasing number of test cases. As a result, we sample our training and testing cases using two strategies as shown in Table 4. First, we identify the corner test case(s) for each circuit problem, usually targeting the test vectors located at the boundaries of all the possible test vectors (truth table). Second, given that the corner cases have been identified, only a few samples of the remaining cases are required. For the combinational circuits, the total number of cases is 50. The number of the remaining training and testing cases were generated by uniformly random

Table 4 Training cases for benchmark problems

Circuit	General	Corner Cases	General Cases	Total No. of Cases

N-bit Selective Parity	N=32
	N=1024
		1. data = 0
		2. data = $2^N - 1$
			48 sampled test inputs 128-bit data	50 (2+48)

N-bit + N-bit	N = 32
	N = 256
		1. a = 0, b = 0
		2. $2^N - 1$, b = $2^N - 1$
		3. a = 0, b = N-bit num
			47 sampled test inputs between 0 to $(2^N - 1)$	50 $(3 + 47)$

N-bit x N-bit	N = 32
	N = 256
		1. a = 0, b = 0
		2. a = $2^N - 1$
		3. a = 0, b = N-bit num
		4. a = 1, b = N-bit num
			46 sampled test inputs between 0 to $(2^N - 1)$	50 $(4 + 46)$

N-bit Up-Down Counter	N = 32
	N = 64
		1. All 4-bit numbers (14 states of the counter) minus 1010 and 1011.
		2. An additional load state testing with a load value of 0 to distinguish between resetting to 0 and loading a 0.
			1. Put the counter in an initial state (load state)
			2. 10 cases to test counting up and down operations.
				$26(15 + 11)$

a and *b* refers to addends for the adder circuit. For the multiplier circuit, *a* and *b* are multiplier and multplicand respectively. N = bit-width

sampling (using `$urandom()` function in SystemVerilog) the internal test vectors within the input range of each circuit; thus, $50 - \underline{no}$ of corner cases. The total number of cases for the N-bit Up–Down Counter is 26 (15 and 11 corner and remaining cases respectively). The remaining cases target the counting up and down operations (5 cases each); a single case to set the counter in an initial state by loading a pre-specified value prior to training and testing. The use of corner cases and sampled test cases ensures our testbenches have good coverage over all possible test vectors.

The total number of cases for the Selective Parity and Adder circuits is 100. The Selective Parity circuit can behave as an odd or even parity circuit based on the `even_odd input signal`. During training and testing, both behaviours are tested. The Adder's `carry_in input signal` can be either a 0 or 1. In a similar approach, the Adder circuit is trained and tested with the same cases when the `carry_in` input signal is 0 and 1 (Table 3).

All obtained solutions are tested for functional accuracy. Given that the evolved solutions for the parameterized circuits cannot possibly be tested by instantiating them with every possible bit-width size, we choose input bit-width sizes less and greater than the input bit-width with which the circuits were evolved/trained with. Test case generation is done using the same procedure outlined in Table 4 for generating the training cases. For example, the Adder was tested using 32-bit + 32-bit and 256-bit + 256-bit test vectors as shown in Table 4.

Experiments were ran on a Dell Precision 7520 (laptop) configured with a RAM size of 64 GB and an Intel Xeon CPU (E3-1545M v5) with 4 CPU cores and a base frequency of 2.90 GHz.

```
        <expr> ::= <terminal> | <terminal> <op> <terminal> | <terminal> <op> <expr>
   <case-stmt> ::= <case> ":" <stmt> | <case> ":" <stmt> <case-stmt>
    <terminal> ::= 0 | 1 | q | load
         <bit> ::= 0 | 1 | ?
          <op> ::= - | +
-------------------------------------------------------------------------------------
<design-module> ::= <begin-module> <always> <endmodule>
      <always> ::= "always @(posedge clk)" <nextline> <switch-case>
  <switch-case> ::= "casex ({reset, en_load, enable_ud, up_ndown})" <nextline> <case-stmt>
                   endcase <nextline>
 <case-default> ::= default ":" "q<=" <expr>;
        <stmt> ::= "q<=" <expr>; <nextline>
        <case> ::= 4'b<bit><bit><bit><bit>
 <begin-module> ::= "module nbit_udcounter #(parameter width=64) (input logic clk, up_ndown,
                   en_load, enable_ud, reset, input logic [width-1:0] load, output logic
                   [width-1:0] q);" <nextline>
   <endmodule> ::= endmodule
    <nextline> ::= "\n"
```

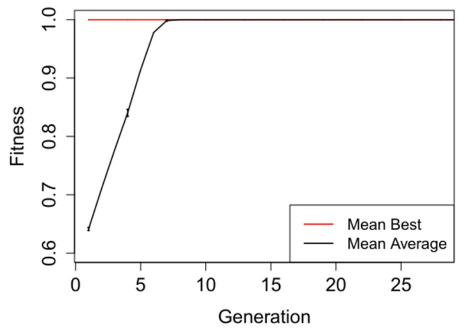

Fig. 2 Mean best and mean average with error bars for N-bit Parity Generator Grammar A

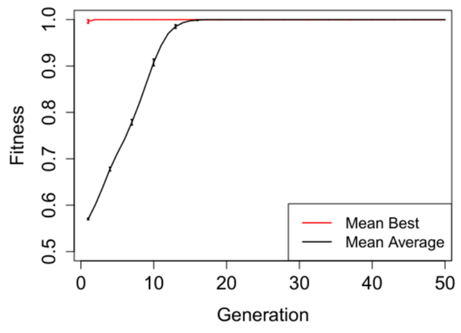

Fig. 3 Mean best and mean average with error bars for N-bit Parity Generator Grammar B

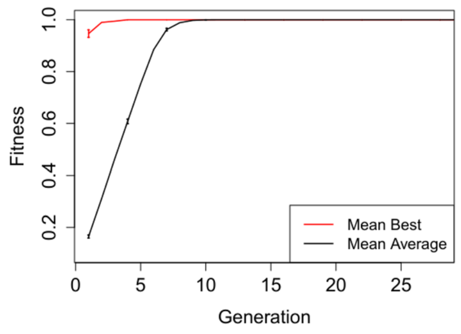

Fig. 4 Mean best and mean average with error bars for N-bit + N-bit Adder

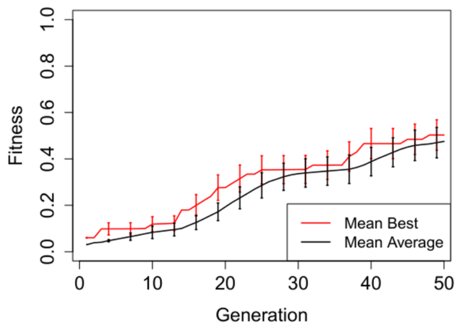

Fig. 5 Mean best and mean average with error bars for N-bit × N-bit Multiplier

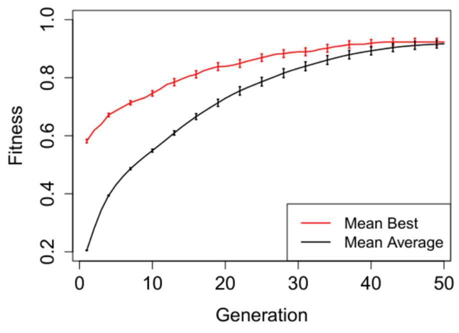

Fig. 6 Mean best and mean average with error bars for N-bit Up-Down Counter

Evolutionary Parameters

Preliminary experiments were conducted to determine reasonable parameter values for the population size and number of generation. Based on the preliminary results obtained, 30 generations and a population size of 500 are used for the parity and adder circuits; 50 generations and 2000 population size are used for the multiplier and Up–Down Counter circuits. All other parameters remain the same across all benchmark problems. Table 3 details the operators and parameters used in the experimental setup.

Results and Discussion

Results obtained from our experiments are shown in this section. Additionally, we compare our results to those obtained by state-of-the-art approaches. In summary, based on results obtained, we conclude that the use of GE and an HDL (SystemVerilog) is capable of evolving complex combinational and sequential circuits, which answers the research questions set out in Sect. 1. However, it should be noted that when comparing with state-of-the-art approaches, despite our evolved circuits requiring less gates to realize in silicon, the optimization of circuits evolved at an abstract level like RTL is largely dependent on the robustness of the synthesis tool.

The rest of the section discusses the evolutionary performance of each benchmark circuit, success rate, analyzes of the effect of input sizes on circuit evaluation and compares of our results to state-of-the-art methods.

Evolutionary Performance of Each Benchmark Circuit

In Figures 2, 3, 4, 5 and 6 we plot the mean best and mean average fitnesses across the 50 independent runs conducted for each benchmark problem, in order to visualize evolutionary performance across generations.

Figures 2 and 3 show the plot for the Selective Parity Grammar A and Grammar B respectively. Both grammars attained maximum mean fitnesses from the initial generation, revealing the triviality in evolving parity generator circuits at this level of abstraction. Fully functional Selective Parity circuits were created via initialization without resorting to any form of evolution. The performance difference between the two grammars from these plots is that the Selective Parity Grammar B obtained lower mean average fitnesses than the Selective Parity Grammar A.

Evolution attains maximum mean best fitness within the first five generations for the adder circuit as observed in Figure 4. The starting mean average fitness is approximately 18% of the maximum fitness and increases rapidly to attain maximum mean fitness from generation 9 onwards. Both the mean best and mean average fitnesses for the Multiplier and N-bit Up–Down Counter progresses steadily, but do not attain maximum mean fitnesses. Furthermore, there exist short to non-existent error bars for the Selective Parity and Adder circuits in Figs. 2, 3 and 4 respectively, indicating little variability. The Up–Down Counter in Fig. 6 has non-existent to short error bars as the evolution progressed. For the multiplier circuit in Fig. 5, short error bars are observed in the earlier generations and widens gradually in subsequent generations, signifying increasing variability in individual fitness values compared to the mean fitness value.

Success Rate

Fifty independent runs were conducted for each benchmark problem. The success rate for each problem is shown in Table 5. We define a successful run as an independent evolutionary run that evolves an optimal solution.

The N-bit Selective Parity (both Grammar A and B) and N-bit + N-bit Adder circuits obtained a 100% successful runs. The N-bit × N-bit Multiplier and N-bit Up–Down Counter attained 46% and 48% successful runs respectively. Based on the population size and success rate of each circuit problem, the N-bit Selective Parity benchmark is the least difficult problem followed by the N-bit + N-bit Adder. The N-bit × N-bit Multiplier and N-bit Up-Down Counter were the most challenging to evolve, requiring a higher population size.

Table 5 Success rate (Out of 50 Independent Runs)

Problem	Population Size	Success Rate
Selective Even Parity Grammar A Selective Even Parity Grammar B	200	50
Adder	500	50
Multiplier	2000	23
Mod-N Up-Down Counter		24

Table 6 Effect of input size on experiment duration

Circuit Problem	Population Size	No of Runs	Input Size	Mean Duration per Run (seconds)	P-value
N-bit Selective Parity (Grammar A)	200	50	64 128	589.6141 8689.829	2.2e-16
N-bit Selective Parity (Grammar B)	200		64 128	15.02446 15.16393	0.4841
N-bit + *N-bit* Adder	500		32 + 32 64 + 64	95.41906 200.4862	2.2e-16
N-bit x *N-bit* Multiplier	2000		32 x 32 64 x 64	844.9755 888.784	0.005294
N-*bit* Up-Down Counter	2000		32 64	864.0152 875.6796	0.5418

Table 7 Number of logic gates required to realize a representative solution of each benchmark circuit evolved

Circuit	Circuit Instance	Total Gates	Respective Gate Counts							
			AND	OR	NOT	NAND	NOR	XNOR	XOR	D-FF
N-bit Selective Parity	4	4	-	-	-	-	-	2	2	-
	6	6	-	-	-	-	-	6	-	-
	8	8	-	-	-	-	-	6	2	-
	10	10	-	-	-	-	-	8	2	-
	12	12	-	-	-	-	-	10	2	-
	16	16	-	-	-	-	-	14	2	-
	19	**19**	-	-	-	-	-	**13**	**6**	-
	20	20	-	-	-	-	-	14	6	-
	32	32	-	-	-	-	-	24	8	-
	64	64	-	-	-	-	-	54	10	-
	128	128	-	-	-	-	-	100	28	-
N-bit + *N-bit* Adder	2 + 2	10	-	-	-	6	-	-	4	-
	3 + 3	15	-	-	-	9	-	-	6	-
	4 + 4	21	-	1	-	13	-	1	6	-
	5 + 5	26	-	1	-	16	-	1	8	-
	6 + 6	31	-	1	-	19	-	1	10	-
	8 + 8	41	-	1	-	25	-	1	14	-
	10+10	**52**	**3**	**2**	-	**29**	-	**2**	**16**	-
	16 + 16	87	21	7	-	34	-	7	18	-
	32 + 32	175	45	15	-	66	-	15	34	-
	64 + 64	369	38	34	1	194	4	30	68	-
N-bit × *N-bit* Multiplier	2 × 2	8	4	-	-	1	2	1	-	-
	3 × 3	35	11	4	1	14	-	4	1	-
	4 × 4	86	18	13	3	36	1	12	3	-
	5 × 5	**140**	**27**	**23**	**3**	**59**	**1**	**20**	**7**	-
	6 × 6	**220**	**39**	**38**	**3**	**95**	-	**35**	**10**	-
	8 × 8	423	77	76	6	170	1	75	18	-
	10 × 10	702	133	134	9	266	3	137	20	-
	16 × 16	1,916	387	412	16	655	6	398	42	-
	32 × 32	8,101	1,600	1,794	46	2,751	12	1,771	127	-
	64 × 64	32,674	7,175	7,681	60	9,878	8	7,451	421	-
Mod-*N* Up-Down Counter	Mod-4	66	-	-	11	34	17	-	-	4
	Mod-8	144	-	-	18	56	62	-	-	8
	Mod-16	280	-	-	32	115	117	-	-	16
	Mod-32	550	-	-	57	224	237	-	-	32
	Mod-64	1,104	-	-	112	458	470	-	-	64
	Mod-128	2,264	-	-	271	745	1,120	-	-	128

Effect of Input Sizes on Experiment Duration

Equally of importance is the effect of increasing input sizes (bit-width) on evaluation time. In order to make this assessment, we conducted separate experiments using smaller input-width sizes: $N = 32$ for Adder, Multiplier and Up–Down Counter circuits; $N = 64$ for the Selective parity circuit as shown in Table 6. A Wilcoxon test was then performed on the simulation time.

Increasing bit-width sizes of Selective Parity Grammar A, N-bit + N-bit Adder and N-bit × N-bit Multiplier circuits has a significant effect on the evaluation time of circuits. In the case of Selective Parity Grammar A, recall we used bitwise operators together with a `generate for-loop`, as a result it requires a minimum of width evaluation events each time an input changes to generate the parity. However, Selective Parity Grammar B uses reduction operators that take a vector (multi-bit input), apply the respective operation to each individual bit and return a single bit as output. Reduction operators execute in a single evaluation event. Therefore, the use of reduction operators takes significantly less time to simulate Selective Parity circuits created with

Grammar B. The Adder performs more additions as the input bit-width sizes increases. Similarly, the multiplier performs several additions to sum up the partial products. As a consequence, the significant increase in evaluation time as the input bit-width size increases for these circuits. In the case of the N-bit Up–Down Counter, no significant difference is observed because we did not exhaustively test the counting up and down operations over the entire input range. However, we do observe a slight difference in the mean experiment duration, signifying that the increase in input bit-width size does have some effect on the experiment duration.

Based on these observations, we suggest the choice of operators to be incorporated into grammars be carefully selected. Furthermore, selecting a reasonable default input bit-width size when evolving parameterized circuits can potentially reduce the experimental time. For example, an input bit-width size of 10 is reasonable for evolving a parameterized adder compared to a bit-width size of 1. A 1-bit + 1-bit Adder has only four test cases which may not be sufficient to properly train a parameterized Adder.

Comparison with Literature

In Table 7, we synthesized a representative solution per benchmark problem to obtain the respective number of gates required to realize each circuit functionality in silicon. Given that all evolved circuits are parameterized, different circuit instances with different bit-width sizes were created for each and synthesized as shown in Table 7. The synthesis tool of choice is Yosys [1]. Yosys is an open source framework for Verilog RTL synthesis. Note that synthesis using commercial tools such as Quartus Prime, Vivado etc. may provide different gate counts. We compare our results (gate counts in Table 7 and mean execution time in Table 6) to the most complex circuits evolved in literature captured in Table 8.

Comparing the number of gates required to realize the respective circuits evolved by state-of-the-art approaches in Table 8 to our evolved circuits in Table 7, GE-evolved circuits require fewer gates. Similarly, less simulation time is required to evolve circuits (in Table 6) compared to state-of-art approaches in Table 8.

Since no N-bit Up-Down Counter has been evolved based on our literature review, in order to make a comparison to the 3-bit and 4-bit Up Counters in Table 8, we modified our evolved Up-Down Counter to disable the count down operation before synthesis. Our synthesized 3-bit requires 38 gates and 3 D-FFs; the 4-bit Up Counters require 46 gates and 4 D-FFs. Comparing these gate and D-FF counts to the number of gates and D-FFs to the Up Counter in Table 8, our resultant Up Counters use less number of D-FFs but require more gates. The number of D-FFs are consistent with the rule of thumb that an N-bit Counter requires an N number of flip-flops. However, our Up Counter comparison isn't entirely fair as no specification on the number of inputs or transition table is provided for the evolved 3-bit and 4-bit Up Counters in [26], since they affect gate count.

Conclusion

In this work, we have shown that designing circuits at a higher level of abstraction, *RTL*, than gate level, allows for the design of significantly larger circuits than existing approaches in literature. Fully functional N-bit selective parity, N-bit + N-bit adder, N-bit × N-bit multiplier and N-bit Up-Down counter circuits have been successful evolved, representing the most complex circuits evolved in literature to

Table 8 Most complex fully functional parity, adder, multiplier and counter circuits evolved

	Circuit	Inputs	Min Number of Gates	Mean Execution Time	Approach
Combinational	**Parity**	19	NA	13,801s	CGP + Decomposition (SDR) [1]
	Adder	10 + 10	78	1,326s	CGP + semantic mutation [2]
	Multiplier	5 × 5	3,671	2,556	CGP + semantic mutation [2]
			NA	548 core-hours single run [2]	Parallel CGP Implementation [3]
		6 × 6	NA	364,752s	GA + Decomposition (GDD + BIE) [4]
Sequential	**Up Counter**	3	39 (32 gates and 7 D flip-flops)	5hr 36min 12s	CGP + VHDL Translation System + Timing Analysis (Single Run) [5]
		4	39 (26 gates and 13 D flip-flops)	2days 20hr 13min 28s	

D-FF represent D flip-flop. In bold are the circuit instances used for comparing the gate count of our evolved circuits to those evolved by state-of-the-art approaches in Table 8 date. The availability of expressive programming constructs and operators in SystemVerilog aided greatly in the evolution of these circuits. Furthermore, our approach recorded better mean execution time and requires less number of gates after synthesis compared to state-of-the-art results.

The evolved circuits are parameterized, meaning once a circuit is evolved, different circuit instances of varying input sizes can be obtained, simply by specifying the desired input sizes without requiring a re-run of experiments compared to existing methodologies. The work also established how appropriate choice of operators can significantly reduce simulation time in the case of the selective parity benchmark. Also, using reasonable input bit-width sizes to evolve parameterized circuits generally reduces the evaluation time, while the evolutionary performance remains the same.

Despite the availability of high level programming constructs in HDLs which helps to evolve complex circuits, we acknowledge certain circuit problems require decomposition. Decomposition is a normal practice in industry. HDLs support hierarchical modelling for the design of large and complex circuits. In hierarchical modelling, design modules in lower layers are included in design modules in higher layers. Communication between circuits are established through the input and output ports of the circuits. Very large complex circuit will be slow to test. Therefore, the conventional generate and test approach of standard evolutionary systems may not suffice. A more efficient approach that will exploit the power of modern processors will be of high priority when we begin to venture into industrial scale circuit designs.

In a typical circuit design flow, SystemVerilog evolved circuits are advanced to the synthesis phase, where synthesis tools are used to obtain the equivalent gate-level representation, as we did using Yosys. As a result, circuit optimization is dependent on the robustness of open-source or commercial synthesis tools. However, progress made in Evolutionary Optimization of Digital Circuits, a sub-field of EHW which focuses on the optimization of functional circuits, have obtained competitive results compared to conventional synthesis tools [52]. Most of the evolutionary optimization of circuits use CGP. Therefore, a merger of GE and CGP for functional evolution and circuit optimization respectively will be the starting point of realizing evolutionary circuit design tool chains.

Funding Open Access funding provided by the IReL Consortium.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Ethics Approval This article does not contain any studies with human participants performed by any of the authors.

Informed Consent Not applicable.

References

1. Wolf C. Yosys open synthesis suite. http://www.clifford.at/yosys/.
2. Higuchi T, Iwata M, Kajitani I, Yamada H, Manderick B, Hirao Y, Murakawa M, Yoshizawa S, Furuya T. In: 1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96, vol. 4 1996;4, pp. 29–32. https://doi.org/10.1109/ISCAS.1996.541893
3. Murakawa M, Yoshizawa S, Kajitani I, Furuya T, Iwata M. T. Higuchi. In: Voigt HM, Ebeling W, Rechenberg I, Schwefel HP (eds) Parallel Problem Solving from Nature—PPSN IV. Berlin Heidelberg, Berlin, Heidelberg: Springer; 1996, pp 62–71.
4. Higuchi T, Murakawa M, Iwata M, Kajitani I, Liu Weixin, Salami M. In: Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97) 1997;187–192. https://doi.org/10.1109/ICEC.1997.592293
5. Thompson A. In: ICES 1996.
6. Yao XT. Higuchi. In: Higuchi T, Iwata M, Liu W, editors. Evolvable systems: from biology to hardware. Berlin Heidelberg, Berlin, Heidelberg: Springer; 1997, pp 55–78.
7. Haddow PC, Tyrrell AM. Evolvable hardware challenges: past, present and the path to a promising future. Springer International Publishing, Cham, 2018, pp. 3–37. https://doi.org/10.1007/978-3-319-67997-6_1.
8. Sekanina L. First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06) 2006, pp 171–178. https://doi.org/10.1109/AHS.2006.36.
9. Lohn J, Haith G, Colombano S, Stassinopoulos D. 2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484), vol. 5 2000; pp. 473–486. https://doi.org/10.1109/AERO.2000.878523.
10. Keymeulen D, Zebulum R, Rajeshuni R, Stoica A, Katkoori S, Graves S, Novak F, Antill C. First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06) 2006, pp 296–300. https://doi.org/10.1109/AHS.2006.64.
11. Zebulum RS, Stoica AM, Keymeulen D, Sekanina L, Ramesham R, Guo X. ICES 2005.
12. Vassilev V, Job D, Miller J. Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware (2000), pp. 151–60. https://doi.org/10.1109/EH.2000.869353.
13. Ali B, Almaini AEA, Kalganova T. Genet Program Evol Mach. 2004;5(1):11. https://doi.org/10.1023/B:GENP.0000017009.11392.e2.
14. Zdenek V. Bridging the gap between evolvable hardware and industry using cartesian genetic programming. Springer

International Publishing, Cham, 2018, pp. 39–55. https://doi.org/10.1007/978-3-319-67997-6_2.
15. Stomeo E, Kalganova T, Lambert C. IEEE Trans Syst Man Cybern Part B (Cybern). 2006;36(5):1024.
16. Coffman K. Real World FPGA Design with Verilog, chap. Chapter 1: Verilog Designs in the realword, pp. 137–206. 1999.
17. LaMeres BJ. Introduction to Logic Circuits & Logic Design with Verilog, chap. Verilog (Part 1), p. 157. in [19] 2019. https://doi.org/10.1007/978-3-030-13605-5.
18. Coffman K. Real World FPGA Design with Verilog. 1999.
19. LaMeres BJ. Introduction to logic circuits & logic design with Verilog. Berlin: Springer International Publishing; 2019. https://doi.org/10.1007/978-3-030-13605-5.
20. RTL Modeling with SystemVerilog For Simulation and Synthesis. Using SystemVerilog for ASIC and FPGA Design. Tualatin, Oregon: Sutherland HDL Inc; 2017.
21. Resurrecting FPGA Intrinsic Analog Evolvable Hardware, ALIFE 2021: The 2021 Conference on Artificial Life, vol. ALIFE 2021: The 2021 Conference on Artificial Life, p 106. https://doi.org/10.1162/isal_a_00448.
22. Hodan D, Mrazek V, Vasicek Z. Proceedings of the 2020 genetic and evolutionary computation conference. association for computing machinery, New York, NY, USA, 2020, GECCO'20, pp 940–948. https://doi.org/10.1145/3377930.3390188.
23. Li Z, Luo W. X. Wang. In: Hornby GS, Sekanina L, Haddow PC. Evolvable systems: from biology to hardware. Berlin Heidelberg, Berlin, Heidelberg: Springer; 2008, pp 47–58.
24. Vasicek Z, Sekanina L. 2014 IEEE International Conference on Evolvable Systems. 2014;133–140.
25. Tetteh MKD, Mota D, Ryan. In: Hu T, Lourenço N, Medvet E (eds) Genetic programming. Cham: Springer International Publishing; 2021, pp 146–61.
26. Henson B, Walker JA, Trefzer MA, Tyrrell AM. Designing digital systems using cartesian genetic programming and VHDL. Springer International Publishing, Cham, 2018, pp 57–86. https://doi.org/10.1007/978-3-319-67997-6_3.
27. Miller JF. Cartesian genetic programming. Berlin, Heidelberg: Springer; 2011.
28. Miller JF, Thomson P. In: Poli R, Banzhaf W, Langdon WB, Miller V, Nordin P, Fogarty TC (eds) Programming genetic. Springer Berlin, Heidelberg: Berlin Heidelberg; 2000, pp 121–32.
29. Sekanina L, Walker JA, Kaufmann P, Platzner M. Evolution of Electronic Circuits. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 125–179. https://doi.org/10.1007/978-3-642-17310-3_5.
30. Sekanina L, Vasicek Z. 2013 IEEE International Conference on Evolvable Systems (ICES). 2013;21–28. https://doi.org/10.1109/ICES.2013.6613278.
31. Ryan C, Collins J, Neill MO. In: Banzhaf W, Poli R, Schoenauer M, Fogarty TC (eds) Programming genetic. Springer. Berlin, Heidelberg: Berlin Heidelberg; 1998, pp. 83–96.
32. Ryan C, Tetteh MK, Dias DM. Proceedings of the 12th International Joint Conference on Computational Intelligence—Volume 1: ECTA. INSTICC (SciTePress, 2020), pp. 28–39. https://doi.org/10.5220/0010066600280039.
33. Patten JV, Ryan C. In: Machado P, Heywood MI, McDermott J, Castelli M, García-Sánchez P, Burelli P, Risi R (eds) Programming genetic. Cham: Sim Springer International Publishing; 2015. pp 105–12.
34. Mota D, Naredo E, Ryan C. Genetic Programming: 24th European Conference, EuroGP 2021, Held as Part of EvoStar 2021, Virtual Event, April 7–9, 2021, Proceedings, vol. 12691, Springer Nature, 2021, vol. 12691, pp. 66.
35. Anjum MS, Ryan C. In: Hu T, Lourenço N, Medvet E (eds) Programming Genetic. Cham: Divina Springer International Publishing; 2020, pp 18–34.
36. Karpuzcu UR. Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation. ACM, New York, NY, USA, 2005, GECCO '05, pp. 394–397. https://doi.org/10.1145/1102256.1102346.
37. Cullen J. Simulated evolution and learning, ed. by Li X, Kirley M, Zhang M, Green D, Ciesielski V, Abbass H, Michalewicz Z, Hendtlass T, Deb K, Tan KC, Branke J, Shi S. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 514–523.
38. Stuart Sutherland PF, Davidmann S. A Guide to Using SystemVerilog for Hardware Design and Modeling, chap. Chapter 6: SystemVerilog Procedural Blocks, Tasks and Functions and Chapter 7: SystemVerilog Procedural Statements, pp. 137–206. in [39]. https://doi.org/10.1007/0-387-36495-1.
39. A Guide to Using SystemVerilog for Hardware Design and Modeling. Springer US. https://doi.org/10.1007/0-387-36495-1.
40. Kalganova T. In: Poli R, Banzhaf W, Langdon WB, Miller J, Nordin P, Fogarty TC (eds) Programming genetic. Springer Berlin, Heidelberg: Berlin Heidelberg; 2000, pp 60–75.
41. Vassilev VK, MillerJF. Proc. Genetic and Evolutionary Computation Conference. Morgan Kaufmann, 2000.
42. Senthilkumar K, Kumarasamy K, Dhandapani V. J Electric Eng Technol. 2020;16:559.
43. Vasicek Z, Sekanina L. IEEE Trans Evol Comput. 2015;19(3):432. https://doi.org/10.1109/TEVC.2014.2336175.
44. Slaný K, Sekanina L. In: O'Neill EM, Ekárt A, Vanneschi L, Esparcia-Alcázar AI (eds) Programming Genetic. Springer Berlin, Heidelberg: Berlin Heidelberg; 2007, pp 311–20.
45. Manfrini FAL, Bernardino HS. Barbosa HJC. In: Handl J, Hart E, Lewis PR, López-Ibáñez M, Ochoa G, Paechter B (eds) Parallel problem solving from nature—PPSN XIV. Cham: Springer International Publishing; 2016, pp 665–74.
46. da Silva JEH, de Souza LAM. Bernardino HS. In: Nicosia G, Pardalos P, Umeton R, Giuffrida G, Sciacca V (eds) Machine learning, optimization, and data science. Cham: Springer International Publishing; 2019, pp 396–408.
47. Hrbacek R, Sekanina L. Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. Association for Computing Machinery, New York, NY, USA, 2014, GECCO '14, pp 1015-1022. https://doi.org/10.1145/2576768.2598343.
48. Torresen J. In: Sipper M, Mange D, Pérez-Uribe A (eds) Evolvable systems: from biology to hardware. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 57–65.
49. Torresen J. In: Liu Y, Tanaka K, Iwata M, Higuchi T, Yasunaga M (eds) Evolvable systems: from biology to hardware. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 1–13.
50. Kalganova T. Proceedings. The Second NASA/DoD Workshop on Evolvable Hardwar. 2000;65–74. https://doi.org/10.1109/EH.2000.869343.
51. Stomeo E, Kalganova T, Lambert C. First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06) 2006;179–185. https://doi.org/10.1109/AHS.2006.47.
52. Fišer P, Schmidt J, Vašíček Z, Sekanina L. 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems 2010;346–351. https://doi.org/10.1109/DDECS.2010.5491755.