Behavioural Modelling of Digital Circuits in System Verilog using Grammatical Evolution

Conor Ryan[1][a], Michael Kwaku Tetteh[1][b] and Douglas Mota Dias[1,2][c]

[1] *CSIS, University of Limerick, Limerick, Ireland*

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

Keywords: Grammatical Evolution, Digital Circuit Design, Evolvable Hardware, Hardware Description Languages (HDL), Lexicase Selection.

Abstract: Digital circuit design is an immensely complex and time consuming task that has been aided greatly by the use of *Hardware Description Languages* and powerful digital circuit simulators that permit a designer to program at a much higher level of abstraction, similar to how software programmers now rarely use Assembly Language, and also to test their circuits before committing them to hardware. We introduce Automatic Design of Digital Circuits (ADDC), a system comprised of Grammatical Evolution (GE), System Verilog, a high level Hardware Description Language (HDL) and *Icarus*, a powerful, but freely available, digital circuit simulator. ADDC operates at a much higher level than previous digital circuit evolution due to the fact that System Verilog supports behavioural modelling through the use of high level constructs such as *If-Then-Else*, *Case* and *Always* procedural blocks. Not only are HDLs very expressive, but they are also far more understandable than circuit diagrams, so solutions produced by ADDC are quite interpretable by humans. ADDC is applied to three benchmark problems from the Digital Circuit Literature. We show that ADDC is successful on all three benchmarks and further demonstrate how the integration of simple knowledge, e.g. the separation of input and output wires, is feasible through the grammars, and can have a major impact on overall performance.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Designing circuits for digital integrated circuits (ICs) is a highly skilled and demanding job and a slow and expensive process (Rabey et al., 2003), with minor errors in design costing millions of dollars to fix. To make the task tractable, specialized Hardware Description Languages (HDLs) are used to design the circuits, which are heavily tested in simulators before being committed to hardware.

This paper introduces ADDC, an automated digital circuit design tool that uses a combination of GE, Verilog and Icarus (a digital circuit simulator) to evolve programs for a number of classic benchmarks. While there has been some related work, described in Section 2, most of it operates at a lower level of *abstraction*, e.g. at the gate level, while the use of HDLs is akin to using a high level language such as `C++` instead of Assembly Language.

[a] https://orcid.org/0000-0002-7002-5815
[b] https://orcid.org/0000-0002-4351-0962
[c] https://orcid.org/0000-0002-1783-6352

This work investigates the applicability of GE to circuit design and how best to set it up. We use information known about the problems to obtain a grammar variant each, for all three problems, while observing their respective effect on evolutionary performance. Experimentally, we show how best to set up the system, most notably in terms of grammar design.

In Subsection 2.2 we describe related work; Section 3 details the problems tackled, parameters and grammars used for the experments; Section 4 presents and discusses the results obtained from the experiments conducted; and Section 5 summarizes the results and future works arising from this work.

2 BACKGROUND

Digital circuit design can be looked upon as operating at different levels. The lowest level, *gate level*, deals with logic gates, and specifies exactly how a circuit should be created, while *functional level*, sometimes referred to as *Register Transfer Level* (RTL), operates at a slightly higher level, consisting of pro-

Ryan, C., Tetteh, M. and Dias, D.
Behavioural Modelling of Digital Circuits in System Verilog using Grammatical Evolution.
DOI: 10.5220/0010066600280039
In *Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020)*, pages 28-39
ISBN: 978-989-758-475-6
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

grams made up of instructions that can directly be translated into gate level. The highest level is *behavioural*; this consists of programs written in Hardware Description Languages (HDLs), such as Verilog/System Verilog and VHDL, although many more exist. These programs mimic the desired functionality of the hardware, although, perhaps surprisingly, not all behavioural programs are *synthesizable*, meaning that they cannot be realized in hardware.

Most modern complex digital circuits are constructed using some form of HDL, and are then subject to numerous simulation and testing steps to ensure that not only is the circuit functionally correct, but also that it is indeed synthesizable. There are many powerful tools, such as Quartus (Intel/Altera) and Vivado (Xilinx), that are used by design experts to perform various tests on their programs before committing them to hardware.

Computer Aided Design (CAD) for circuits employs these sorts of tools in an iterative way, as shown in Figure 1, where designers repeatedly perform functional simulations to ensure that their circuit works as intended, before moving to the synthesis phase, in which they ensure that the circuit is fully synthesizable. The reasons why a circuit may not be synthesizable vary from timing issues (components may simply be too far from each other and thus introduce some unexpected delays) to *fitting* issues (it may not be possible to create this particular circuit given the real estate available), amongst others.

Key to any digital circuit design is the creation of a *testbench*. A testbench is analogous to a regression test in software programming, as it essentially contains a set of test cases, although testbenches typically contain extra code to actually run the circuit under test on the various test cases.

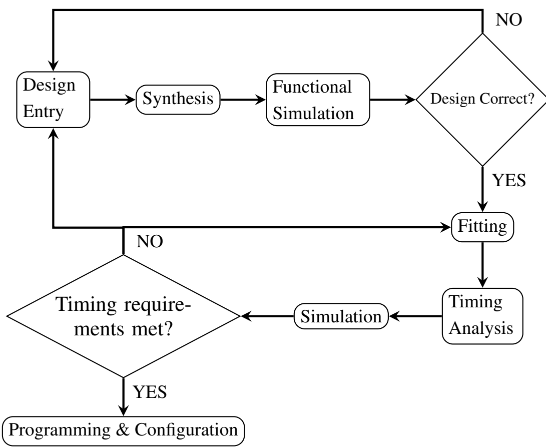

Figure 1: The standard CAD flow for digital circuit design.

2.1 Evolvable Hardware

Evolvable Hardware (EHW) is a field that deals with the application of evolutionary algorithms (EAs) to electronic circuit design. It is comprised of two major areas based on its application—adaptive hardware (a.k.a. Darwin Machine (de Garis, 1993)) and circuit synthesis (Yao and Higuchi, 1999). With adaptive hardware, hardware is equipped to autonomously adapt or reconfigure its architecture as it interacts with its environment, while, with circuit synthesis, behavioural requirements are known upfront. The circuit is evolved and tested using simulators similar to those noted above, before being committed to hardware; this is referred to as *extrinsic* evolution in EHW (de Garis, 1994). Other modes of evolution are *intrinsic* and *mixtrinsic*. In intrinsic evolution, candidate circuits are directly simulated on a target hardware such as Field Programmable Gate Arrays (FPGAs) (Thompson, 1997), while mixtrinsic evolution is a combination of both intrinsic and extrinsic evolution for simulating the evolved circuits—thus a population is divided in two, with each half evaluated with one of the evaluation modes, but not on both (Stoica et al., 2000).

2.2 Related Work

EAs have been applied to circuit design, both in the analog and digital circuits design domains. However, as the focus of this work is in the digital circuit domain, we restrict our review to such circuits. Most studies carried out in EHW applied to circuit synthesis have operated at the gate and functional level evolution. For example, Cartesian GP (CGP) (Miller and Thomson, 2000), a GP variant, has been used extensively to evolve digital circuits at both gate and functional level. CGP uses directed acyclic graphs as the program encoding structure instead of trees (Miller and Thomson, 2000). A CGP genotype consist of different kinds of genes: a *function* gene which holds an address to a function in a lookup table that a node in the graph must perform, *connection* genes that encode addresses of where a node can take its inputs from, which is usually in a feedforward manner (Miller and Thomson, 2000), and *output* genes which hold addresses of nodes where outputs of the program can be retrieved. This representation makes CGP ideal for low level simulation, and usually the output form of the final circuit design's phenotype is obtained or specified in boolean logic form. At the gate level, CGP has been applied successfully in evolving numerous digital circuits such as seven segment display, adders and multipliers of varying input sizes (Sekanina et al., 2011).

GE has also been used to evolve circuits such as one-bit adder, d-latch at the gate level (Cullen, 2008) using Verilog. However, gate-level evolution is that it is less likely to scale to highly complex circuits from scratch (Vassilev and Miller, 2000). In response to issues of scalability inherent in gate level evolution, Murakawa et al. proposed functional level evolution, which uses higher level functions such as multiplexers, adders, subtractors instead of primitive gates (Murakawa et al., 1996) to help reduce the search space. Similarly, Vassilev and Miller evolved a 3-bit multiplier using only binary multipexers (Vassilev and Miller, 2000). 9- and 25-Median approximate circuits have also been designed at the functional (Vasicek and Sekanina, 2015). We address the scalability concern by performing circuit evolution at a more abstract level —behavioural modelling, where focus is placed on describing the behaviour of the circuit.

The *Production Genetic Algorithm* (PGA), is a GA augmented with a rewriting system which uses a grammar to design digital circuits (Mizoguchi et al., 1994). The chromosome of an individual dictates the production rule to be used during the mapping process. Using Structured Function Description Language (SFL) as the HDL, PGA was applied to design a circuit for an ant to follow the John Muir Trail.

2.3 Higher Abstraction Levels

Higher abstraction levels (e.g RTL) are much more capable of evolving or finding solutions to complex circuits quicker but do not necessary guarantee more optimized designs when synthesized to lower levels (gate or transistor level) which is the normal process prior to the actual fabrication of any digital circuit (Gajda and Sekanina, 2007). Karpuzcu evolved a 1–bit full adder using an RTL grammar (Karpuzcu, 2005). Using a population size of 200, a termination criterion of a 100,000 fitness evaluations and 35 experimental runs found just two optimal solutions. These runs took 50,369 and 19,772 fitness evaluations respectively. We evolved the 1–bit full adder using the same grammar in (Karpuzcu, 2005), same population size (200) but opted for 200 generations as the termination criterion (thus 40,000 maximum evaluations). We obtained 8 optimal solutions out of 30 runs after 23 generations on average(thus 4,600 fitness evaluations). However, we do not include these results in this paper as 1–bit full adder is trivial (increasing the population size resulted in more successful runs) and the grammar design did not exploit System Verilog's behavioural constructs enough.

Cullen, on the other hand, specified the grammar at the gate level (Cullen, 2008). As the maximum number of gates and wires needed per circuit design cannot be determined due to nature of genetic algorithms, these variables cannot be added to the grammar prior to the experimental runs. As a result, a post-processing technique for declaration and usage of unique wires and gate instances was designed and referred to as a DEFINE/USE pair (Cullen, 2008).

2.4 Grammatical Evolution

GE evolves programs in any arbitrary language using grammars, either Backus Naur Form (BNF) or Attribute Grammars (AG), and has been applied to many domains such as bio-informatics, engineering and architecture (Ryan et al., 2018). Digital circuit design hasn't received much attention from GE and, as noted above, most previous EC work was at the gate level, but the use of HDLs and the good availability of powerful simulators suggests that it could be an ideal target.

GE operates by using a mapper that takes a genotype and a valid context grammar defined for a problem as inputs. The genotype consists of a sequence of codons; usually each codon is an integer equivalent of an 8–bit binary string decoded from a binary string genome. Each rule begins with a non-terminal and has a number of productions which expand into other non-terminals or terminals. The `modulo` rule is usually used for production rule selection (O'Neill and Ryan, 2001). A valid phenotype is obtained if all non-terminals have been fully expanded to terminals, in which case a syntactically correct program or program fragment in the target language has been derived.

3 EXPERIMENTAL DESIGN

The main driver for the design of digital circuits in this work is GE, with Icarus Verilog (Williams and Baxter, 2002). We use LibGE, an open source *C++*

Table 1: Experimental Run Parameters.

Parameter Type	Parameter Value
Initialization	Sensible Initialization
No of generations	200
Mutation rate	0.01
Crossover rate	0.8
Replacement rate	0.5
No of runs	30
Population	2,000
Selection	Lexicase Parent Selection

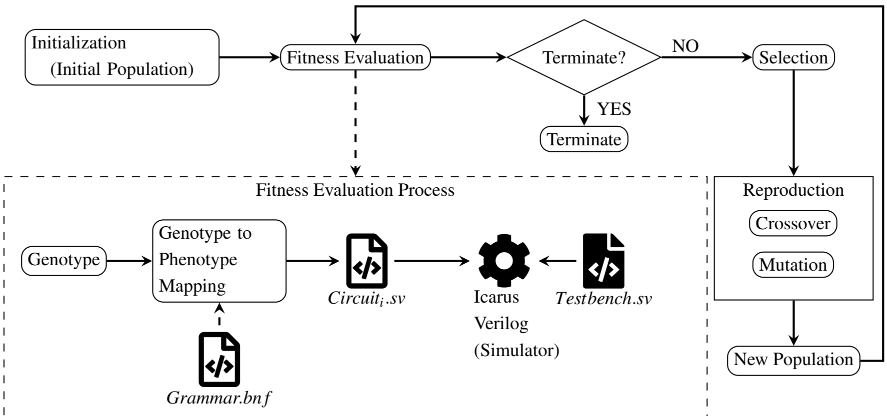

Figure 2: GE circuit design process.

Table 2: Number of Testcases Used for each problem.

Problem	Number of Testcases
11-Multiplexer	2,048
Seven Segment Display	16
Hamming Code (7,4) Decoder	128

implementation of Grammatical Evolution (Nicolau and Slattery, 2006) while Icarus Verilog is a lightweight and open source simulator, used for simulating the circuits. Thus, all circuits are evolved extrinsically using Icarus Verilog and testbenches designed based on the truth tables for all three problems described below. These testbenches are used by the simulator to simulate the evolved circuits to test for their correctness. The fitness of a circuit is simply the sum of all the testcases successfully passed.

We use a population size of 2,000. The parent selection technique employed is Lexicase selection (Spector, 2012). Lexicase selection selects individuals as parents depending on how well they perform on a randomly shuffled set of testcases, with every selection requiring reshuffling of the testcases. This increases the probability of choosing individuals which, although might be relatively poor performers overall, solve cases that much of the rest of the population do not. The circuit design process is captured in Figure 2 , the parameters for the experiments are detailed in Table 1 and number of testcases used is shown in Table 2.

3.1 Differences between ADDC and Existing Work

The key difference between the work presented in this paper, ADDC, and existing work is the language used to represent the circuits. Virtually all other work, including those that employ HDLs, operate at the gate level. ADDC operates at a higher level and thus has access to the feature rich language of Verilog. For example, common high level programming constructs such as `Always`, `If`, `Case`, Boolean operators and numeric constants are all standard in Verilog, but are not trivial to code at the gate level. Thus, for complex circuit designs, behavioural level modelling is preferable (Bryan Mealy, 2018) and generated circuits are intrepretable and maintainable by humans.

Furthermore, as demonstrated in Section 4, the use of a grammar permits us to impose simple constraints on the search space that dramatically improve the performance of the system. In this paper, we restrict these constraints to simple, commonsense, such as separating the input and output wires.

3.2 Problems

Our three problems and associated grammars are described below.

3.2.1 Multiplexer

A Multiplexer circuit switches data lines through a single line using address or select lines. The multiplexer problem has been addressed using GP (Kruse et al., 2013) and a classifer system (Fredivianus et al., 2010). The Multiplexer Grammar A in Figure 3 uses bitwise-and, bitwise-or, logical negation and ternary operator as the function set. The Multiplexer Grammar B in Figure 4 uses the ternary operator (a shortened form of `if-else` construct as the only functional construct, which we speculate to be adequate in solving the multiplexer problem. In both grammars, the data bits are distinguished from the address bits. This sort of constraint is straightforward to impose with GE, as it is simply a matter of modifying the grammar slightly (specifically, the <expr> non-terminal), while in standard GP, this would be a more complex exercise as it is essentially introducing a new *type*,

SCIENCE AND TECHNOLOGY PUBLICATIONS

```
<design-module>   ::= <begin-module><nextline><code-block>
                      <end-module><nextline>
<code-block>      ::= <always><statement>
<always>          ::= "always@(*)"<nextline>
<statement>       ::= <output>=<expr>;<nextline>
<expr>            ::= (<expr>&<expr>) | (<expr>"|"<expr>)
                      | !(<expr>)
                      | (<address-bit>?<expr>":"<expr>)
                      | <data-bit>
<output>          ::= out
<address-bit>     ::= a0 | a1 | a2
<data-bit>        ::= d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
<nextline>        ::= "\n"
<begin-module>    ::= module "mux(output logic out, input
                       logic a0, a1, a2, d0, d1, d2, d3,
                       d4, d5, d6, d7);"
<end-module>      ::= end-module
```

Figure 3: Multiplexer Grammar Version A; this grammar variant separates the address bits from the data bits and Uses the *bitwise and* (&), |, *ternary operator* (*if−else*), *bitwise or* (|) and *logical negation* (!) operators in the grammar.

```
<design-module>   ::= <begin-module><nextline><code-block>
                      <nextline><end-module>
<code-block>      ::= <always><statement>
<always>          ::= always@(*)<nextline>
<statement>       ::= <output>"="<expr>;<nextline>
<expr>            ::= (<address-bit>?<expr>":"<expr>) | <data-bit>
<output>          ::= out
<address-bit>     ::= a0 | a1 | a2
<data-bit>        ::= d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
<nextline>        ::= "\n"
<begin-module>    ::= module "mux(output logic out, input
                      logic a0, a1, a2, d0, d1, d2, d3,
                       d4, d5, d6, d7);"
<end-module>      ::= endmodule
```

Figure 4: Multiplexer Grammar B: The Grammar uses *if−else* construct as the only functional construct and separates the data and address bits.

which would complicate the task of satisfying the *closure* principle for GP. The grammar also features the `always` proceedural block, which behaves like a loop by continously executing all statements within its body, starting at time 0.

3.2.2 Seven Segment Display (SSD)

An SSD is an electronic device which consists of seven segments or LEDs for displaying decimal numbers as well as characters (limited to decimal numbers in this work). The device receives a 4-bit binary number (0000 − 1001) referred to as *binary coded decimal* (BCD) as input and in response uses a 7–bit number (each bit corresponding an ON/OFF state of a segment) to turn on the appropriate segments/LEDs in displaying the digit. The two grammars designed for the SSD problem both use the `switch-case` construct as the only functional construct. In SSD Grammar A and B in Figure 5 and Figure 6 respectively, using

```
<design-module>   ::= <begin-module><next-line><code-block>
                      <next-line>endmodule
<code-block>      ::= <always><statement>
<always>          ::= always@(bcd)<next-line>
<statement>       ::= begin<next-line><switch-case><next-line>end
<switch-case>     ::= case(bcd)<next-line><case-statement>
                      <next-line><default-case><next-line>endcase
<case-statement>  ::= <bcd-value>":"<output>"="<seven-segment>;
                      | <bcd-value>":"<output>"="<seven-segment>;
                      <next-line><case-statement>
<default-case>    ::= default":"<output>"="<seven-segment>;
<bcd-value>       ::= 4'b<bit><bit><bit><bit>
<seven-segment>   ::= 7'b<bit><bit><bit><bit><bit><bit><bit>
<next-line>       ::= "\n"
<output>          ::= segment
<bit>             ::= 0 | 1
<begin-module>    ::= module "ssdisplay(output logic[6:0]
                       seg, input logic[3:0] bcd);"
```

Figure 5: SSD Grammar Version A: The data and address bits grouped together in the *terminal rule* and uses the *bitwise and* (&), *ternary operator* (*if−else*), but has its BCD values separated from the seven segment values.

```
<design-module>   ::= <begin-module><next-line><code-block>
                      <next-line>endmodule
<code-block>      ::= <always><statement>
<always>          ::= always@(bcd)<next-line>
<statement>       ::= begin<next-line><switch-case><next-line>end
<switch-case>     ::= case(bcd)<next-line><case-statment>
                      <next-line><default-case><next-line>endcase
<case-statement>  ::= <terminal>:<output>"="<terminal>;
                      | <terminal>":"<output>=<terminal>;
                      <next-line><case-statement>
<default-case>    ::= default":"<output>"="<terminal>;
<nextline>        ::= "\n"
<output>          ::= segment
<bcd-value>       ::= 4'b0000 | 4'b0001 | 4'b0010 | 4'b0011
                      | 4'b0100 | 4'b0101 | 4'b0110
                      | 4'b0111 | 4'b1000 | 4'b1001
<seven-segment>   ::=  7'b1111110 | 7'b0110000 | 7'b1101101
                      | 7'b1111001 | 7'b0110011 | 7'b1011011
                      | 7'b1011111 | 7'b1110000 | 7'b1111111
                      | 7'b1111011 | 7'b0000000
<begin-module>    ::= module "ssdisplay(output logic[6:0]
                      seg, input logic[3:0] bcd);"
```

Figure 6: SSD Grammar Version B; this grammar is similar to *SSD Grammar Version A* in Figure 5 but its BCD and seven segment values are separated in the same they are in SSD Gramamr Version A.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

a similar approach to the Multiplexer Grammar design, the BCD-values and seven-segments are distinguished from each other. To investigate the impact of grammar design decisions/domain knowledge introduced into grammars on evolutionary performance, in Grammar A in Figure 5 the grammar is designed in such a way that it constructs the binary numbers from single bits (assuming the appropriate binary numbers required by the problem are not known). This is a non-trivial task, particularly as not all values are needed. Given that there is a defined set of inputs and outputs, we encode this information into grammar variant B in Figure 6. Note that the individuals still need to build the case statement, which is a variable length construct, in which each case can be one of 110 possibilities, so this is not a trivial problem. SSD Grammar B in Figure 6 has all the BCD and seven segment values explicitly defined and distinguished from each other making this grammar the least difficult and should have the best success rate in solving the SSD problem.

3.2.3 Hamming Code (7,4) Decoder

Hamming Codes belong to the category of error correcting codes known as Linear Block Codes (Miller et al., 2009). This category of codes have a minimum distance of 3, meaning that they are capable of correcting a single error and at most detecting two errors. The Hamming Code (7,4) encodes a dataword by generating three parity check bits using the 4–bit *dataword*. The 3 parity check bits and the 4–bit dataword are combined to create a 7–bit binary string referred to as the *codeword*. After transmission, the Hamming Code (7,4) *decoder* retrieves the dataword from the codeword by generating a 3–bit number referred to as the *syndrome*. Each bit serves as a parity check for 4–bit codeword. Figure 13 and Figure 14 represents the grammars designed to evolve the Hamming Code (7,4) decoder. Both grammars feature a `function` to generate the syndrome from the codeword received, as well as an `always` block together with an `if-else` construct that checks the generated syndrome for validity or corruption of the codeword. If the syndrome is non-zero then the codeword has been corrupted during transmission. To correct the codeword, the decimal equivalence of the generated syndrome determines the bit position where an error is assumed to have occured during transmission and as a result the bit at the location is flipped. The production(s) for the $\langle expr \rangle$ rule differentiates Hamming Code (7,4) Decoder Grammar A from Grammar B as shown in Figure 13 and Figure 14 respectively in the APPENDIX. The productions of the Hamming Code (7,4) Decoder Grammar A are designed in such a way that evolution has to find the expression that generates the correct parity bit which forms part of the 3–bit syndrome. However, from the problem statement of the Hamming Code (7,4) Decoder, only 4–bits out of the 7–bit codeword are required in deriving the expression that generates each parity bit that make up the 3–bit syndrome used for checking the correctness of the received dataword. As a result, we incorporate this information into the design of the production associated with the $\langle expr \rangle$ rule of the Hamming Code (7,4) Decoder Grammar B. This design decision ensures 4 bits out of the 7–bit codeword are selected every time in deriving the expression. We therefore speculate that Hamming Code (7,4) Grammar B should exhibit better chances of solving the Hamming Code (7,4) Decoder problem than its Grammar B counterpart.

4 RESULTS AND DISCUSSION

In this section we present and discuss the results obtained from the thirty experimental runs conducted across each of the three problems examined. Figures 10−11 show the mean average and best fitnesses across generations with error bars.

4.1 Successful Runs

We tabulated the success rate in order to visualize the impact of grammar design on the frequency of finding optimal solutions. Table 3 shows the success rate of the 11–bit multiplexer, SSD and Hamming Code (7,4) decoder, respectively, across all 30 experimental runs. Both Multiplexer Grammar A and Seven Segment Display Grammar A were unable to obtain any optimal solution within 200 generations with a population of 2,000. To investigate this further, we conducted another experiment using the Grammar A versions of the SSD and the 11-bit multiplexer problems while increasing the population size from 2,000 to 3,000. Again, no optimal solutions were obtained using the 11–bit Multiplexer Grammar A version, although 4 optimal solutions out of the 30 runs were obtained using the Seven Segement Grammar A version. Thus, more fitness evaluations are required to obtain optimal solutions if grammars are not concise enough. The Grammar A variants for the multiplexer and SSD problems performed worse than their B counterparts due to the absence of the commonsense domain knowledge infused into the grammars.

Table 3: Success Rate.

Problem Grammar	Successful Runs (out of 30)
11–bit Multiplexer Grammar A	0
11–bit Multiplexer Grammar B	22
Seven Segment Display Grammar A	0
Seven Segment Display Grammar B	28
Hamming Code (7,4) Decoder Grammar A	22
Hamming Code (7,4) Decoder Grammar B	30

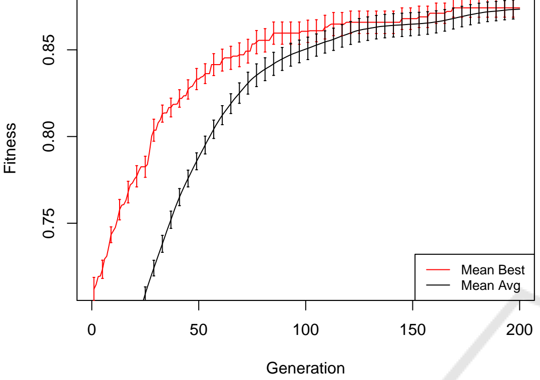

Figure 7: Mean Average / Mean Best for 11-bit Multiplexer Grammar Version A with error bars. The address bits are distinguished from data bits. Uses the *bitwise and*, *ternary operator* (`if-else`), *bitwise or* and *logical negation* operators in the grammar.

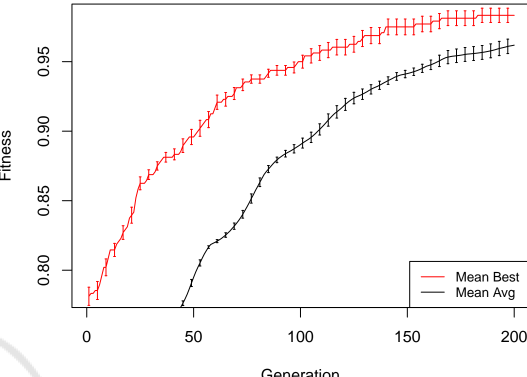

Figure 8: Mean Average / Mean Best for 11-bit Multiplexer Grammar B Version with error bars. Uses `if-else` construct as the only functional construct and has its data bits distinguished from the address bits.

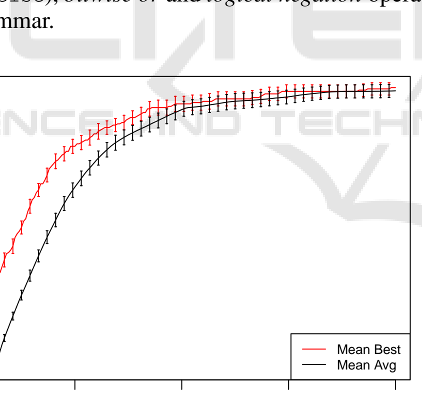

Figure 9: Mean Average / Mean Best for Seven Segment Display Grammar Version A with error bars. Has its BCD values separated from the seven segment values. Also, it contains the necessary building blocks for generating the required BCD and seven segment values.

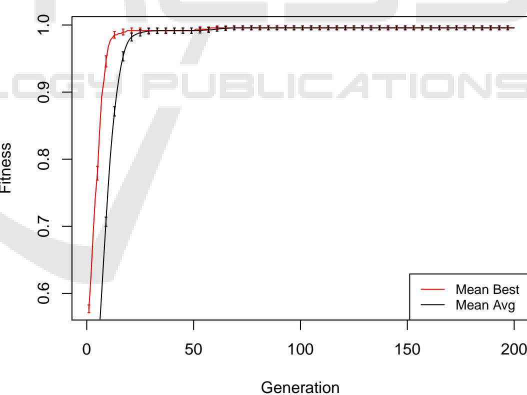

Figure 10: Mean Average / Mean Best for Seven Segment Display Grammar Version B with error bars. It's BCD and seven segment values are separated. Also, all the required BCD and seven-segment values were explicitly specified.

4.2 Grammar Choice

The use of domain knowledge in the derivation of the 11-bit Multiplexer, SSD Grammar and Hamming Code (7,4) Decoder variants had the intended influence in reducing the search space.

Recall that the Multiplexer Grammar A version had other functions in addition to the ternary operator which were not required, given that the ternary operator (`if-else`) is sufficient to model the behaviour of the 11-bit multiplexer problem. From Figure 7 and Figure 8, we observe that the Multipexer Grammar B which uses the ternary operator as the only functional construct outperformed Multiplexer Grammar A by a significant margin. Therefore, the impact of gram-

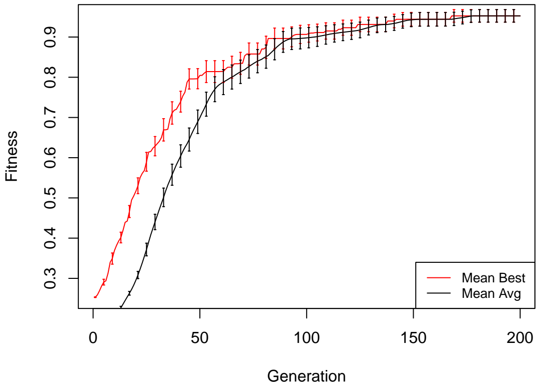

Figure 11: Mean Average/ Mean Best for Hamming Code (7,4) Decoder Grammar A Version with error bars. The ⟨expr⟩ is recursive by design and evolution has to derive the correct expressions that generate the syndrome.

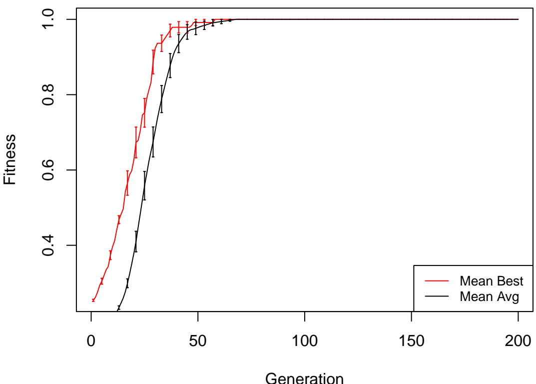

Figure 12: Mean Average/ Mean Best for Hamming Code (7,4) Decoder Grammar B Version with error bars. The ⟨expr⟩ for this Grammar Version is not recursive but instead limited in length which we know from the problem description.

mar design, specifically the use of highly expressive functions such as *if-else*, which we can do in this case because we are evolving at the behavioural level.

The SSD Grammar A version had to construct the BCD values and the 7 segment bits while the Grammar B version had these defined within the grammar. As speculated, from Figure 9 and Figure 10, the Grammar B Version performed better than its Grammar A counterpart. This is due to the fact that the Grammar A version had an additional task of constructing the BCD and 7 segment binary numbers. Also, from Table 3 Grammar A was unable to obtain an optimal solution in any of the 30 runs conducted.

The only difference between the Hamming Code (7,4) Decoder Grammar A and B is the ⟨*expr*⟩ rule. In the case of the Grammar A, the productions of the ⟨*expr*⟩ rule were designed in such a way that the resulting expression can be of an arbitrary length; in other words evolution has to figure out the right expression that generates the right parity bit. However, from the description of the Hamming Code (7,4) Decoder we know the resulting expression that generates each bit of the 3-bit syndrome has to select only 4 bits out of the 7–bit codeword. As a result we limit the production of the ⟨*expr*⟩ rule to select 4 bits out of 7-bit codeword, making Grammar B capable of solving the Hamming Code (7,4) Decoder better and quicker comparatively. From Figure 11 and Figure 12, we observe that Hamming Code (7,4) Decoder Grammar B outperforms it's Grammar A counterpart. This shows the impact a simple domain knowledge constraint can have on evolutionary performance with regards to grammar design. This further supports our claim that the use of grammars facilitates the introduction of simple and generic, but useful constraints on the search space.

An optimal solution for each successful grammar have been shown in Listings 1– 4 in the APPENDIX. The Multiplexer Grammar B solutions vary in length due to the nesting nature of the `if-else` construct. Similarly, the number of `case-statements` varies across solutions obtained using Seven Segement Display Grammar B, out of which some are redundant; since System Verilog permits duplicate cases but evaluates them based on preceedence. Hamming Code (7,4) Decoder Grammar A solutions are identical except the arrangement of the bits in the expressions that generate the syndromes vary but are of no relevance. However, solutions obtained using the Hamming Code (7,4) Decoder Grammar B differ in the length of the expressions that generate the syndromes due to the recursiveness of the ⟨expr⟩ rule. These expressions contain bitwise operations that are redundant, for example $cw[1]\&cw[1] = cw[1]$

5 CONCLUSION AND FUTURE WORK

We have presented ADDC, the Automated Design of Digital Circuits, which uses a combination of GE, Verilog and Icarus to evolve digital circuits. We have tested it on three difficult benchmarks and have demonstrated how grammar can be leveraged to dramatically reduce the search space by separating out clearly distinct wires, i.e. input and output and the inclusion of relevant functions such as the ternary operator (`if-else`) in the case of the Multiplexer Grammar B version.

These experiments have also demonstrated the impact of high-evel structures, such as `If`, `always` proceedural blocks and `case` on evolving circuits. Be-

cause these are so expressive, they can replace many gates that would otherwise have to be evolved, and instead permit evolution to focus on higher level behaviours. In all cases where we were able to test the impact of these, the best performance came from setups that used both of the features, i.e. separating the input and output space and the use of high level structures.

These findings show a roadmap for tackling much more complex tasks, specifically, the ways in which the grammars are designed and the use of Lexicase Selection to target poorly covered parts of the solution space. Additionally, results obtained based on the grammar design choices suggest that the use of Layered Learning or Attribute grammars may be better suited for tougher circuit problems.

The problems tackled in this work are interesting as they are of relevance in real-world applications. Moving forward, more difficult problems such as 70–bit and 135–bit Multiplexers, Hamming Code (127, 120) Decoder and multipliers of varying inputs will be looked at. Though the problems tackled in this work are not very difficult, we demonstrated that grammar design can impose another level of difficulty on problems; as a result no optimal solutions were found for the Grammar A versions of the SSD and Multiplexer problems. In this paper we also assume to be evolving circuits from scratch which may not be the case in industrial applications. Future works will also target using IP blocks as modules to evolve more complex circuits; adding them to our system will be trivial, as it will simply involve adding them to the grammar. Furthermore, all three problems were treated as single objective problems, that is, *functionality*. Future work will consider their optimization when synthesized to a lower level such as the gate level, probably using some multi-objective fitness measures. Also relevant is the need to adopt techniques to help decompose problems into subproblems in an attempt to reduce the search space, particularly for problems of high complexity, as is often done in digital circuit design. Furthermore, the incorporation of semantic information into GE will be key in ensuring only syntactically and semantically valid individuals are evaluated, as long simulation time is a difficult bottleneck to address.

Crucially, because the system we have assembled uses a full Verilog simulator that employs test benches to test the circuits, there is a clear path to extend ADDC to sequential circuits, that is, logic circuits that are time dependent and require a clock to operate. This opens up the possibility of employing highly powerful building blocks such as flip-flops and counters.

ACKNOWLEDGMENTS

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 third author is partially financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

Bryan Mealy, F. T. (2018). *Free Range VHDL*.

Cullen, J. (2008). Evolving Digital Circuits in an Industry Standard Hardware Description Language. In Li, X., Kirley, M., Zhang, M., Green, D., Ciesielski, V., Abbass, H., Michalewicz, Z., Hendtlass, T., Deb, K., Tan, K. C., Branke, J., and Shi, Y., editors, *Simulated Evolution and Learning*, pages 514–523, Berlin, Heidelberg. Springer Berlin Heidelberg.

de Garis, H. (1993). Evolvable Hardware Genetic Programming of a Darwin Machine. In Albrecht, R. F., Reeves, C. R., and Steele, N. C., editors, *Artificial Neural Nets and Genetic Algorithms*, pages 441–449, Vienna. Springer Vienna.

de Garis, H. (1994). An artificial brain ATR's CAM-Brain Project aims to build/evolve an artificial brain with a million neural net modules inside a trillion cell Cellular Automata Machine. *New Generation Computing*, 12(2):215–221.

Fredivianus, N., Prothmann, H., and Schmeck, H. (2010). Xcs revisited: A novel discovery component for the extended classifier system. In Deb, K., Bhattacharya, A., Chakraborti, N., Chakroborty, P., Das, S., Dutta, J., Gupta, S. K., Jain, A., Aggarwal, V., Branke, J., Louis, S. J., and Tan, K. C., editors, *Simulated Evolution and Learning*, pages 289–298, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gajda, Z. and Sekanina, L. (2007). Reducing the Number of Transistors in Digital Circuits Using Gate-level Evolutionary Design. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 245–252, New York, NY, USA. ACM.

Karpuzcu, U. R. (2005). Automatic Verilog Code Generation Through Grammatical Evolution. In *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation*, GECCO '05, pages 394–397, New York, NY, USA. ACM.

Kruse, R., Borgelt, C., Klawonn, F., Moewes, C., Steinbrecher, M., and Held, P. (2013). *Fundamental Evolutionary Algorithms*, pages 227–274. Springer London, London.

Miller, F. P., Vandome, A. F., and McBrewster, J. (2009). *Hamming Code: Parity Bit, Two- out- of- Five Code, Hamming(7,4), Reed-Muller Code, Reed-Solomon Error Correction, Turbo Code, Low- Density Parity-Check Code, Telecommunication, Linear Code*. Alpha Press.

Miller, J. F. and Thomson, P. (2000). Cartesian Genetic Programming. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., and Fogarty, T. C., editors, *Genetic Programming*, pages 121–132, Berlin, Heidelberg. Springer Berlin Heidelberg.

Mizoguchi, J., Hemmi, H., and Shimohara, K. (1994). Production genetic algorithms for automated hardware design through an evolutionary process. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 661–664 vol.2.

Murakawa, M., Yoshizawa, S., Kajitani, I., Furuya, T., Iwata, M., and Higuchi, T. (1996). Hardware evolution at function level. In Voigt, H.-M., Ebeling, W., Rechenberg, I., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature — PPSN IV*, pages 62–71, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nicolau, M. and Slattery, D. (2006). *libGE*. for version 0.27alpha1, 14 September 2006.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Rabey et al. (2003). *Digital Integrated Circuits (2nd Edition)*. Pearson.

Ryan, C., O'Neill, M., and Collins, J. J. (2018). *Handbook of Grammatical Evolution*. Springer Publishing Company, Incorporated, 1st edition.

Sekanina, L., Walker, J. A., Kaufmann, P., and Platzner, M. (2011). *Evolution of Electronic Circuits*, pages 125–179. Springer Berlin Heidelberg, Berlin, Heidelberg.

Spector, L. (2012). Assessment of Problem Modality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 401–408, New York, NY, USA. ACM.

Stoica, A., Zebulum, R., and Keymeulen, D. (2000). Mixtrinsic Evolution. In Miller, J., Thompson, A., Thomson, P., and Fogarty, T. C., editors, *Evolvable Systems: From Biology to Hardware*, pages 208–217, Berlin, Heidelberg. Springer Berlin Heidelberg.

Thompson, A. (1997). An evolved circuit, intrinsic in silicon, entwined with physics. In Higuchi, T., Iwata, M., and Liu, W., editors, *Evolvable Systems: From Biology to Hardware*, pages 390–405, Berlin, Heidelberg. Springer Berlin Heidelberg.

Vasicek, Z. and Sekanina, L. (2015). Evolutionary approach to approximate digital circuits design. *IEEE Transactions on Evolutionary Computation*, 19(3):432–444.

Vassilev, V. K. and Miller, J. E. (2000). Scalability problems of digital circuit evolution evolvability and efficient designs. In *Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware*, pages 55–64.

Vassilev, V. K. and Miller, J. F. (2000). Embedding Landscape Neutrality To Build a Bridge from the Conventional to a More Efficient Three-bit Multiplier Circuit. In *Proc. Genetic and Evolutionary Computation Conference*. Morgan Kaufmann.

Williams, S. and Baxter, M. (2002). Icarus verilog: Open-source verilog more than a year later. *Linux J.*, 2002(99):3–.

Yao, X. and Higuchi, T. (1999). Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 29(1):87–97.

APPENDIX

```
⟨code-block⟩        ::= ⟨begin-module⟩⟨declaration⟩⟨func⟩⟨newline⟩
                        ⟨always-block⟩⟨newline⟩⟨end-module⟩
⟨declaration⟩       ::= integer⟨d-var-name⟩;⟨newline⟩
⟨always-block⟩      ::= always@(⟨input-codeword⟩)begin⟨newline⟩
                        ⟨output⟩⟨newline⟩⟨syndrome⟩⟨newline⟩
                        ⟨if-else⟩⟨newline⟩end
⟨if-else⟩           ::= if(⟨condition⟩)begin end⟨new-line⟩
                        else begin⟨error-correction⟩end
⟨condition⟩         ::= ⟨d-var-name⟩ "==" 3'b00
⟨error-correction⟩  ::= ⟨output-codeword⟩[⟨d-var-name⟩]"="
                        ~⟨input-codeword⟩[⟨d-var-name⟩;]
⟨d-var-name⟩        ::= synd_value
⟨output⟩            ::= ⟨output-codeword⟩"="⟨input-codeword⟩;
⟨syndrome⟩          ::= ⟨d-var-name⟩"="⟨func-call⟩
⟨func-call⟩         ::= ⟨func-name⟩(⟨input-codeword⟩)
⟨func⟩              ::= function[1":"2]⟨func-name⟩
                        (⟨func-input⟩);⟨newline⟩begin⟨new-line⟩
                        ⟨func-stmt⟩endendfunction
⟨func-name⟩         ::= syndrome
⟨func-input⟩        ::= input[1":"7]⟨in-var⟩
⟨func-stmt⟩         ::= ⟨func-name⟩ [0]"="⟨expr⟩;⟨new-line⟩
                        ⟨func-name⟩ [1]"="⟨expr⟩;⟨new-line⟩
                        ⟨func-name⟩ [2]"="⟨expr⟩;⟨new-line⟩
⟨expr⟩              ::= ⟨invar⟩[⟨idx⟩] | ⟨invar⟩[⟨idx⟩]⟨op⟩⟨expr⟩
⟨idx⟩               ::= 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨input-codeword⟩    ::= i_codeword
⟨output-codeword⟩   ::= o_codeword
⟨invar⟩             ::= cw
⟨op⟩                ::= ^ | & | "|"
⟨begin-module⟩      ::= "module hm_decoder (input [1:7]
                         i_codeword, output logic [1:7]
                        o_codeword);"⟨newline⟩
⟨end-module⟩        ::= endmodule
⟨newline⟩           ::= "\n"
```

Figure 13: Hamming Code (7,4) Decoder Grammar A. The ⟨expr⟩ rule is recursive and therefore evolution has to find out the number of and the correct bits to select from the codeword in deriving the expression that generates the 3–bit syndrome.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
⟨code-block⟩         ::= ⟨begin-module⟩⟨declaration⟩⟨func⟩⟨newline⟩
                         ⟨always-block⟩⟨newline⟩⟨end-module⟩
⟨declaration⟩        ::= integer⟨d-var-name⟩;⟨newline⟩
⟨always-block⟩       ::= always@(⟨input-codeword⟩)begin⟨newline⟩
                         ⟨output⟩⟨newline⟩⟨syndrome⟩⟨newline⟩
                         ⟨if-else⟩⟨newline⟩end
⟨if-else⟩            ::= if(⟨condition⟩)begin end⟨new-line⟩
                         else begin⟨error-correction⟩end
⟨condition⟩          ::= ⟨d-var-name⟩ "==" 3'b00
⟨error-correction⟩   ::= ⟨output-codeword⟩[⟨d-var-name⟩]"="
                         ~⟨input-codeword⟩[⟨d-var-name⟩;]
⟨d-var-name⟩         ::= synd_value
⟨output⟩             ::= ⟨output-codeword⟩"="⟨input-codeword⟩;
⟨syndrome⟩           ::= ⟨d-var-name⟩"="⟨func-call⟩
⟨func-call⟩          ::= ⟨func-name⟩(⟨input-codeword⟩)
⟨func⟩               ::= function[1":"2]⟨func-name⟩
                         (⟨func-input⟩);⟨newline⟩begin⟨new-line⟩
                         ⟨func-stmt⟩endendfunction
⟨func-name⟩          ::= syndrome
⟨func-input⟩         ::= input[1":"7]⟨in-var⟩
⟨func-stmt⟩          ::= ⟨func-name⟩ [0]"="⟨expr⟩;⟨new-line⟩
                         ⟨func-name⟩ [1]"="⟨expr⟩;⟨new-line⟩
                         ⟨func-name⟩ [2]"="⟨expr⟩;⟨new-line⟩
⟨expr⟩               ::= ⟨invar⟩[⟨idx⟩]⟨op⟩⟨invar⟩[⟨idx⟩]⟨op⟩
                         ⟨invar⟩[⟨idx⟩]⟨op⟩⟨invar⟩[⟨idx⟩]
⟨idx⟩                ::= 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨input-codeword⟩     ::= i_codeword
⟨output-codeword⟩    ::= o_codeword
⟨invar⟩              ::= cw
⟨op⟩                 ::= ^ | & | "|"
⟨begin-module⟩       ::= "module hm_decoder (input [1:7]
                          i_codeword, output logic [1:7]
                         o_codeword);"⟨newline⟩
⟨end-module⟩         ::= endmodule
⟨newline⟩            ::= "\n"
```

Figure 14: Hamming Code (7,4) Decoder Grammar B. The production for ⟨expr⟩ rule has been designed to select 4 bits out of the 7–bit codeword every single time to form the expressions that generate the syndrome (information we know from the Hamming Code (7,4) Decoder problem description.

```
module mux(output logic out, input
    logic a0, a1, a2, d0, d1, d2, d3,
    d4, d5, d6, d7);
    always@(*)
        out = (a1 ? (a1 ? (a2 ? (a0 ? d0
        : d4) : (a0 ? d1 : d5)) : (a2 ?
        d0 : (a2 ? (a2 ? d0 : d1) : (a0
        ? d2 : d5)))) : (a2 ? (a0 ? d2 :
        d6) : (a0 ? d3 : d7)));
endmodule
```

Listing 1: An optimal solution obtained using 11–bit Multiplexer Grammar B.

```
module seven_segment_display(output
    logic[6:0] segment, input logic[3:0]
    bcd);

always@(bcd)
begin
    case(bcd)
         4'b0100 : segment = 7'b0110011;
         4'b1000 : segment = 7'b1111111;
         4'b0000 : segment = 7'b1111110;
         4'b0110 : segment = 7'b1011111;
         4'b0010 : segment = 7'b1101101;
         4'b0100 : segment = 7'b1011111;
         4'b0110 : segment = 7'b1101101;
         4'b0000 : segment = 7'b0000000;
         4'b0001 : segment = 7'b0110000;
         4'b0011 : segment = 7'b1111001;
         4'b1001 : segment = 7'b1111011;
         4'b0111 : segment = 7'b1110000;
         4'b0101 : segment = 7'b1011011;
         default : segment = 7'b0000000;
    endcase
end
endmodule
```

Listing 2: An optimal solution obtained using Seven Segment Grammar B.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
module hc_decoder(input [1:7] i_codeword,
        output logic [1:7] o_codeword);

 integer synd_value ;
 function [2:0] syndrome (input [1:7] cw);
    begin
       syndrome[0] = cw[5] ^ cw[1] & cw[1]
            ^ cw[3] ^ cw[7];
       syndrome[1] = cw[6] ^ cw[7] ^ cw[3]
            ^ cw[2];
       syndrome[2] = cw[5] ^ cw[4] ^ cw[6]
            ^ cw[7];
     end
 endfunction

 always @(i_codeword) begin
    o_codeword = i_codeword;
    synd_value = syndrome (i_codeword);
    if(syndrome_value == 3'b0) begin end
    else begin
        o_codeword[synd_value] =
        ~i_codeword[synd_value];
    end
 end
endmodule
```

Listing 3: An optimal solution obtained using Hamming Code (7,4) Decoder Grammar A.

```
module hc_decoder(input [1:7] i_codeword,
    output logic [1:7] o_codeword);

 integer synd_value ;
 function [2:0] syndrome (input [1:7] cw);
     begin
        syndrome[0] = cw[3] ^ cw[1] ^ cw[5]
            ^ cw[7];
        syndrome[1] = cw[2] ^ cw[6] ^ cw[3]
            ^ cw[7];
        syndrome[2] = cw[4] ^ cw[6] ^ cw[5]
            ^ cw[7];
     end
 endfunction

 always @(i_codeword) begin
     o_codeword = i_codeword;
     syndrome_value = syndrome (i_codeword);
     if(syndrome_value == 3'b0) begin end
     else begin
        o_codeword[synd_value] =
        ~i_codeword[synd_value];
     end
 end
 endmodule
```

Listing 4: An optimal solution obtained using Hamming Code (7,4) Decoder Grammar B.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS