Input and Output Generation for the Verification of ALU: a Use Case

Ondrej Cekan, Richard Panek, Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology,
Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
Tel.: +420 54114-{1361, 1362, 1223}
{icekan, ipanek, kotasek}@fit.vutbr.cz

Abstract

The paper presents the approach to universal stimuli generation for an arithmetic-logic unit (ALU). It is not focused only on input data generation, but it is possible to generate also expected output in one stimulus. The process of generation is based on a probabilistic constrained grammar which is designed to universally describe stimuli for various circuits. This grammar is processed by our framework. The experiment in functional verification, which shows the quality of generated stimuli, is also presented.

1. Introduction

Random stimuli generation is currently a very important process of checking the correct behavior of various circuits [1]. Complex or also simple circuits must be properly tested or verified before real deployment to exclude design or implementation errors. It is also necessary to verify the correct output for expected and unexpected input combinations (stimuli). Stimuli are typically randomly constructed and may take many forms from binary values on simple circuit pins to a complex program in the data memory of a processor.

Each system is unique, and therefore, it requires specific input stimuli for its operation. In order to verify the correct behavior, it is necessary to create a set of test cases (input stimuli and expected outputs) to detect any possible mismatches in the circuit. Depending on the complexity of the circuit, this activity may be quite challenging, and therefore, tools that allow to generate random inputs automatically are created. These tools are targeted to a specific circuit and their use is considerably limited for different devices. Also, these tools do not allow the expected output to be generated, and further efforts must be made to create a reference system [2].

For the reasons outlined above, we have focused on developing a framework for universal stimuli generation that can be used for various circuits.

The paper is organized as follows. In section 2 our previous research is described. In section 3 the related work is summarized. Section 4 deals with our definition of probabilistic constrained grammar that we use for the generation process while section 5 devotes to the grammar definition for the arithmetic-logic unit. Experimental results are mentioned in section 6 and finally in section 7 the paper is concluded.

2. Previous Research

In our previous research, we designed and developed a framework for universal stimuli generation based on a probabilistic (stochastic) context-free grammar [3]. It is a common context-free grammar that defines probabilities for its production rules with which they are applied. We have extended this grammar by restrictive conditions (constraints) and defined the new grammar system - Probabilistic Constrained Grammar (PCG) [4] that we use in our research. Constraints are used to dynamically change the probabilities of production rules during the generation.

We have also defined the architecture of universal stimuli generation [5] that is shown in Fig. 1. This architecture consists of two input structures (Production Rules, Constraints) which are based on PCG. The first structure defines the production rules of a grammar, while the second structure includes constraints for the application of production rules. Together these two structures form the resultant grammar. Grammar defined in this way is processed by the Generator Core of the framework that assembles the resultant stimulus on its output.

We applied this framework to more complex circuits (e.g., RISC (Reduced Instruction Set Computer) [6] processors, control unit) to verify the possibility of generating stimuli using PCG. We verified the quality of obtained stimuli from the point of view of the generation speed and the achieved coverage [7] in functional verification.

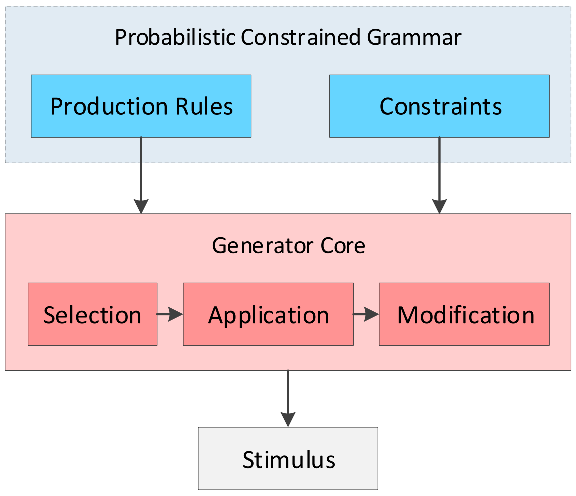

Fig. 1: The architecture of universal stimuli generation.

3. Related Work

The current trend in stimulus generation focuses primarily on more complex circuits (e.g. processors), because it is not trivial to construct a valid stimulus (working program). Simpler stimuli, including test vectors, can be generated directly in the simulation environment where verification takes place (e.g. Modelsim tool from Mentor Graphics [8]) or an external tool.

A number of specific stimuli generators exists for application-specific processors (ASICs) [9], digital signal processors (DSPs) [10], protocol interfaces, field programmable gate array (FPGA) converters [11], and more. These tools and their approaches are complex and their use is limited to the particular system.

As a universal stimuli generator, MicroGP tool [12] can be mentioned which does not only generate stimuli but it also finds the most optimal solution of hard problems.

In this paper, we use test stimuli which can be obtained directly from the verification environment from Modelsim tool for comparison with our approach.

4. Probabilistic Constrained Grammar

A probabilistic constrained grammar is a pair G:

`G = (H,C);` where:

H is a probabilistic context-free grammar.
C is an ordered list of constraints for the grammar H.

A probabilistic context-free grammar is a 5-tuple H:

`H = (N,T,R,S,P);` where:

N is a finite set of non-terminal symbols.
T is a finite set of terminal symbols, $N \cap T = 0$.
R is a finite set of production rules with form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.
S is the starting non-terminal.
P is a finite set of probabilities for production rules.

Constraints restrict the grammar in the application of production rules. The constraint is a 5-tuple C:

$C = (R_S, R_D, P, [R_E], [O]);$ where:

R_S is the activation rule the application of which sets this constraint.
R_D is the target rule which probability is modified.
P is the new probability value.
R_E (optional) is the stop rule which application cancels this constraint.
O (optional) is the count of application of the rule R_E before canceling this constraint.

The constraints limit the application of production rules for a given non-terminal through probabilities which can be modified throughout the generation process, and therefore, we are able to control the resultant stimulus.

5. Arithmetic-Logic Unit

In general, this paper focuses on the principles of random stimuli generation which can be used for many simple circuits. It is not just generating input values for these circuits, as in our previous work, but we would like to show the expressive power of PCG and the ability to simultaneously generate as input values as output values that will be part of the resultant stimulus. Thanks to this, it is possible to check quickly the correctness of the output in case of circuit testing or functional verification.

The arithmetic-logic unit (ALU) [13] is our test case for which we show the random generation of input stimuli and their result for the selected operation. An arithmetic logic unit performs arithmetic and bitwise operations on integer binary numbers. The symbolic representation of ALU is shown in Fig. 2.

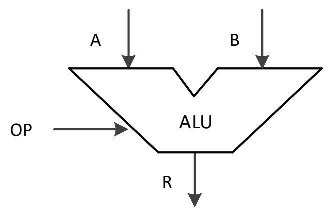

Fig. 2: The symbolic representation of ALU.

ALU has typically two input operands A and B which are N bits long. Its operation is selected by OP input bits. The R output represents the result of the operation over the operands. ALU can be variously complex, therefore, it can contain more input and output bits (e.g. status and control bits), and its supported operations can be also different in various versions.

In this paper, we limit only to inputs and outputs as shown in the figure. Among the operations under consideration, we include two arithmetic operations – addition with carry (ADD) and subtraction (SUB), and four bitwise operations - AND, OR, XOR and NOT. However, the principles that we use for the generation are applicable to other operations.

5.1.1. Arithmetic operations. In this paper, we show the generating of stimuli for the arithmetic addition with carry operation. We can divide the process of creating production rules into several sections - *Input values*, *Logic*, and *Result*. Each section includes specific rules that are applied during the generation. The most complex section is *Logic* the production rules of which must ensure the correct procedure for calculating the result of this operation. The schematic representation of these sections is shown in Fig. 3 which shows also the parts of resultant stimulus.

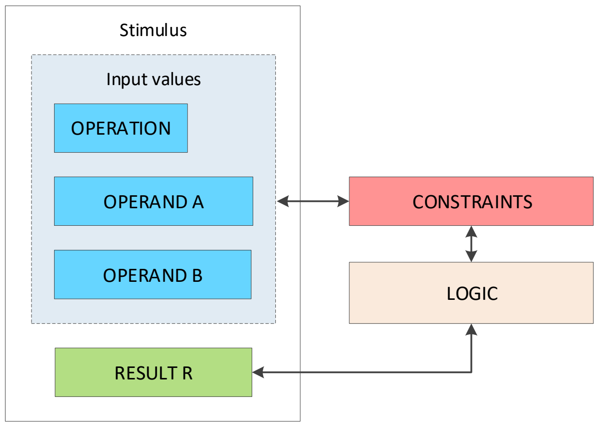

Fig. 3: The schematic representation of arithmetic operation in our framework.

As can be seen in the figure, stimulus is composed of four lines which are represented by integer binary numbers. The lines are generated sequentially as outlined, therefore, it is important to keep the context in which the rules were applied. The first line is the operation code followed by two operands (the numbers which are summed up) and the last line is a final result. The bit widths of inputs can be entered arbitrarily based on used ALU, e.g. for our ALU 1 bit can be long operation, 8 bits long operands, and 8 bits long result.

The constraints are also shown in the figure, because they are involved in the selection of production rules. Based on the random generation of input operands, certain constraints are set, and therefore, the logic is modified – the probabilities of production rules are deterministically set to produce an unambiguous result.

In the definition of production rules, each operand is divided into N non-terminals (N is equal to operand bit width). In our case, the operand A is divided into eight bit non-terminals A7-A0, where A7 is the most significant bit (MSB) and A0 is the least significant bit (LSB). The same applies for the operand B. The rules are as follows:

```
A -> A7 A6 A5 A4 A3 A2 A1 A0
B -> B7 B6 B5 B4 B3 B2 B1 B0
```

Each bit non-terminal A7-A0 can be zero or one, therefore, it can take one of the following two terminals (comma represents OR, terminals are in quotes):

```
A7 -> '0', '1'
A6 -> '0', '1'
…
A0 -> '0', '1'
```

Using these production rules, we have random value in the first operand. At the moment, we have not information about a carry bit propagation. The carry bit is determined during the generation of the operand B. For these purposes, it is necessary to keep the value of operand A. Therefore, each bit non-terminal B7-B0 can be replaced for non-terminal BiA0 (if Ai were zero), BiA1 (if Ai were one), BiA0C (if Ai were zero and a carry bit was set) or BiA1C (if Ai were one and a carry bit was set). These possibilities have to be reflected in production rules:

```
B7 -> B7A0, B7A1, B7A0C, B7A1C
B7A0, B7A1, B7A0C, B7A1C -> '0','1'
…
B0 -> B0A0, B0A1, B0A0C, B0A1C
B0A0, B0A1, B0A0C, B0A1C -> '0','1'
```

It remains to add production rules that will generate the final result:

```
R -> R7 R6 R5 R4 R3 R2 R1 R0
R8, R7, …, R0 -> '0', '1'
```

Now it is known which values the input operands have and whether the carry bits have been propagated. These rules without any control would generate random non-terminals and the result would not reflect the operation addition with carry. Therefore, constraints have to be utilized. The framework performs the right derivations (substitution of the rightmost non-terminals) for the both operands and result, therefore, the substitution will start with the bit A0 to A7, then with B0 to B7, and then R0 to R7.

The B0 does not have a carry bit, therefore, we change the probability to zero for two rules with carry on the start of generation (S is the default starting non-terminal):

```
cons(->S, B0->B0A0C, 0);
cons(->S, B0->B0A1C, 0);
```

The context of the application of the rules for operand A have to be stored in operand B, therefore, we keep the context by limiting the selection of rules for operand B and its corresponding bit:

```
cons(A0->'0', B0->B0A1, 0);
cons(A0->'0', B0->B0A1C, 0);
cons(A0->'1', B0->B0A0, 0);
cons(A0->'1', B0->B0A0C, 0);
...
cons(A7->'0', B7->B7A1, 0);
cons(A7->'0', B7->B7A1C, 0);
cons(A7->'1', B7->B7A0, 0);
cons(A7->'1', B7->B7A0C, 0);
```

After this limitation, we have two rules for each bit B7-B1 which can be used after the generation of the operand A. The bit B0 have only one deterministic rule without the carry bit. After the generation of operand A and the bit B0, we are able to determine the carry bit (rule) for the following bit B1 and the result for bit R0. The same applies for the other bits B2-B6:

```
cons(B0A0->'0', R0->'0', 100);
cons(B0A0->'0', B1->B1A0C, 0);
cons(B0A0->'0', B1->B1A1C, 0);
cons(B0A0->'1', R0->'1', 100);
cons(B0A0->'1', B1->B1A0C, 0);
cons(B0A0->'1', B1->B1A1C, 0);
cons(B0A1->'0', R0->'1', 100);
cons(B0A1->'0', B1->B1A0C, 0);
cons(B0A1->'0', B1->B1A1C, 0);
cons(B0A1->'1', R0->'0', 100);
cons(B0A1->'1', B1->B1A0, 0);
cons(B0A1->'1', B1->B1A1, 0);
...
```

In this logic, constraints for rules `BiA0C` and `BiA1C` can be easily completed to obtain the correct result.

The selection of result bit after applying the rules is based on the following Tab. 1 which defines the classical addition with carry operation.

Tab. 1: Grammar truth table of addition with carry C.

Ai bit	Bi bit	Ri	Ci+1
Ai->'0'	BiA0->'0'	Ri->'0'	0
Ai->'0'	BiA0->'1'	Ri->'1'	0
Ai->'0'	BiA0C->'0'	Ri->'1'	0
Ai->'0'	BiA0C->'1'	Ri->'0'	1
Ai->'1'	BiA1->'0'	Ri->'1'	0
Ai->'1'	BiA1->'1'	Ri->'0'	1
Ai->'1'	BiA1C->'0'	Ri->'0'	1
Ai->'1'	BiA1C->'1'	Ri->'1'	1

The final real result can be seen as in the following example:

```
0                           #OP
01101001                    #A
10001011                    #B
11110100                    #R
```

This process of creation is useful and usable for other arithmetic and bitwise operations. The main condition is to cover all possible cases (creation of corresponding production rules) which are then used or disabled by means of constraints during generation. The use of the constraints causes a fact that the defined grammar is more deterministic and the output is valid.

5.1.2. Bitwise operations. The process of creation grammar for the bitwise operations is very similar as in the previous subsection in the case of arithmetic operations. The basis is again to maintain the context through several production rules and their non-terminals. The difference is only in the generation of results, respectively the limitation of the rules for generating the partial bit of the result so that the output is correct for the given operation.

6. Experimental Results

We performed an experiment in functional verification in which we examined the highest coverage of the key functions of the presented ALU. Functional verification is the process of checking the correctness of a system based on comparing its inputs and outputs with reference model which implements the same specification. We had implemented verification environment in which we investigate the valid result of the ALU and the *code coverage*. The code coverage measures the system source code through typical metrics like statements, branches, expressions, conditions, and states. Through this information, we are able to determine, when the ALU is sufficiently verified. It is a percentage value suitable for comparison or different generators.

The result of our experiment can be seen in Fig. 4. From the experiment, it can be seen that there is a difference between our generator (USG) and the Build-in generator of test stimuli in verification environment. The both of the generators work on random stimuli construction but in our approach we are able to drive the generation process to direct the convergence to the better results. Verification environment checks also corner cases for input data (e.g. all ones or zeros in operands and result) and through probability values, we are able to increase the ability to generate this combinations. Therefore, the USG can hit this coverage points faster than only with clean random generation. The coverage was 94.91% for USG and 91.63% for Built-in generator for 100 stimuli. For 200 stimuli, the coverage was balanced for both generators on 94.91%.

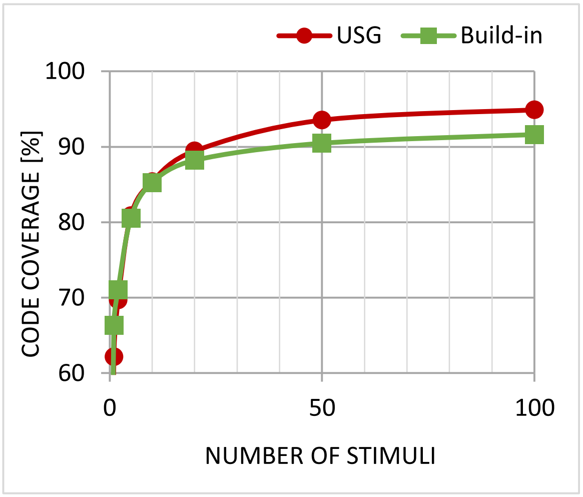

Fig. 4: The code coverage in functional verification.

7. Conclusions and Future Research

The aim of this paper was to show the possibility of generating as input as expected output. Automatic generation of random stimuli facilitates the work and time to test or verify a designed circuit. We showed on an arithmetic logic unit the generation of input and output together for which we defined our probabilistic constrained grammar. The output stimulus was composed of as randomly generated input operands as the expected result for this unit. The introduced mechanism has been shown on addition with carry operation, however, the defined principles are general and can be used for other arithmetic or bitwise operations, cyclic redundancy check generation, and so on. The experiment in functional verification showed that this principle is ductile to get better results than other ones.

This work is one of the partial goals for checking fault tolerance in Field Programmable Gate Array (FPGA). The main goal is to verify the correctness of affected system under a fault and to determine the importance of each of the configuration memory bits in FPGA. The future research will address this topic.

8. Acknowledgements

This research was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); JU ECSEL Project SECREDAS (Product Security for Cross Domain Reliable Dependable Automated Systems), Grant agreement No. 783119; project IT4Innovations excellence in science - LQ1602 and BUT project FIT-S-17-3994.

9. References

[1] A. Meyer. *Principles of Functional Verification*. Elsevier Science, 2003.

[2] N. Kitchen and A. Kuehlmann. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258-265, Nov 2007.

[3] R. Giegerich. *Introduction to Stochastic Context Free Grammars*. Humana Press, Totowa, NJ, 2014.

[4] O. Cekan,, J. Podivinsky, and Z. Kotasek. Program Generation Through a Probabilistic Constrained Grammar. In *2018 Euromicro Conference on Digital System Design (DSD)*, accepted to conference, 8 pages, Aug 2018.

[5] J. Podivinsky, O. Cekan, J. Lojda, M. Zachariasova, M. Krcma, and Z. Kotasek. Functional verification based platform for evaluating fault tolerance properties. *Microprocessors and Microsystems*, 52:145-159, 2017.

[6] D. A. Patterson. Reduced instruction set computers. *Commun*. ACM, 28(1):8-21, January 1985.

[7] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *Design and Test of Computers*, IEEE, 18(4):36-45, May 2001.

[8] M. Graphics. Verification academy - the most comprehensive resource for verification training, [Online] (2013). Available: www.verificationacademy.com.

[9] J. Hudec. An efficient technique for processor automatic functional test generation based on evolutionary strategies. In *Proceedings of the ITI, 33rd International Conference on Information Technology Interfaces*, 527-532, May 2011.

[10] B. Wess. Automatic code generation for integrated digital signal processors. In *1991., IEEE International Sympoisum on Circuits and Systems*, pages 33-36 vol.1, Jun 1991.

[11] A. M. Amiri, A. Khouas, and M. Boukadoum. Pseudorandom stimuli generation for testing time-to-digital converters on an fpga. *IEEE Transactions on Instrumentation and Measurement*, 58(7):2209-2215, July 2009.

[12] G. Squillero. Microgp-an evolutionary assembly program generator. Genetic Programming and Evolvable Machines, 6(3):247-263, 2005.

[13] J. G. Bartkowiak and M. A. Nix. Arithmetic logic unit, Jan. 25 1994. US Patent 5,282,153.