MICROPROCESSOR VERIFICATION BY SYNTACTICALLY-CONTROLLED GENERATION OF THE TEST PROGRAMS

G. Bodean, PhD, assoc.prof.
Technical University of Moldova

INTRODUCTION

Random (stochastic) test generation is an actual up-to-date direction and efficient technique for simulation-based verification of large complex hardware (digital) designs such as microprocessors [1,…, 9]. From one hand, there exist sophisticated verification tools for generation, including controlled, random tests, which are used for functional verification of processors [10, 11, 12]. From the other hand, it is proposed to add specific language constructs to HDL to keep the randomization features [13, 14].

An important issue is the evaluation of the effectiveness of random test verification methods. A variety of coverage metrics have been proposed: the branch coverage and path coverage [15, 16] models used in software testing [17], finite state machine based metrics [18,…, 21], an observability metric [22, 23], and design-specific metrics such as architectural events [1, 24].

The predecessors paid more attention to development of the tools that help the user to control the process of test programs (TP) generation. Till now the generalized model of estimation of the test programs quality is not yet developed. Our objective is to make an advance in developing the estimation of quality and the controlled synthesis of the test program generator (TPG).

In this paper it is proposed the approach, called *syntactic* (keeping tradition in [25]), where the structure of stochastic grammar defines the controlled languages constructs for weighted random test generation. The structure of grammar is "restored" from the microprocessor (MP) specification just from the instructions set specification and the interface protocol description.

Here it is used the existing HDL languages (namely VHDL) features for implementation of the proposed methodology. In the next section of the paper the mathematical model for quality evaluation of the controlled random test generation is proposed. Then, in section 2, briefly is presented the unit under verification, and in sections 3, 4, and 5 is presented the methodology of synthesis of the (stochastic) grammar generator for simulation-based verification of a simple microprocessor. In Section 6 the results of microprocessor test benches simulation are presented and analyzed. The paper ends with some concluding remarks and ideas about future development of controlled random testing for verification.

1. THE LENGTH OF RANDOM TEST VERIFICATION

To compute the test length we take into account the tools (simulation coverage Report) of evaluation of the simulation coverage measure in a widely used CAD-system such as Quartus II from Altera. This "estimation" is based on computation of percentage of exercise (flip-flopping) of the design's nodes, more exactly, is checked as the ratio of output ports actually toggling between 1 and 0 during simulation, compared to the total number of output ports presented in the netlist.

Let us p_e is the probability of exercising of node e on feeding of a test pattern to inputs of unit under verification (UUV). So, the probability $P_e(l)$ of exercising of node e on feeding l test pattern is equal to

$$P_e(l) = 1 - (1 - p_e)^l, \quad (1)$$

where $e \in E$, E is the netlist of design.

For the small values of p_e results the inequality:

$$1 - (1 - p_e)^l \geq 1 - \exp(-l \cdot p_e) \quad (2)$$

Accepting λ as the level of confidence (vs risc) of the random test verification of length l, i.e. $P_e(l) = \lambda$, from (1) and (2) is follows:

$$l \leq \frac{1}{p_e} \ln \frac{1}{1 - \lambda}. \quad (3)$$

For computation l the level of confidence λ and the probability p_e must be known before. For example, if the values of λ are equal to 0.632; 0.865; 0,95; 0,982 etc., then values of random test length are: $l = 1/p_e$; $2/p_e$; $3/p_e$; $4/p_e$ etc.

It is easy to prove that the formula (3) is valid also if the path coverage metric is used. The value (3) is an apriori estimation of the verification test

length *l* and is well correlated with experimental data presented in referenced bibliography.

Thus, we obtain a theoretical model of estimation random test verification quality. But, any theoretical model must be approved practically. In the further sections of the paper we develop the syntactic model of random generator for microprocessor verification. Some verification experiments will be performed and the resulted test length will be compared with expected one.

2. BRIEF PRESENTATION OF UNIT UNDER VERIFICATION

The considered unit under verification is a 4-bit microprocessor slice AM2901. Its full VHDL description is available from [26]. Figure 1 depicts the AM2901 microprocessor as a gray-box. Slice-MP contains the following functional units: two-address RAM array RAM_Regs, the one-word shift register Q_Reg, the source operand multiplexer Src_op, the arithmetic logic unit ALU that performs three arithmetic operations and five logic functions of two 4-bit operands, the output multiplexer Out_Mux. The **a** and **b** 4-bit addresses are used to address the RAM_Regs 16-word register file, where size of the word is equal 4 bit.

Signal **d** is a direct input to the **r** source operand multiplexer. Signal **q** represents the contents of Q_Reg and feeds the **s** to source operand multiplexer. The ALU has carry-in c_{in}, carry-out c_{out} and other additional signals.

The function of MP-slice is defined by a 9-bit microinstruction *i*. Three bits *i*[2:0] define the source operands, *i*[5:3] – ALU function, and *i*[8:6] – ALU destination.

Thus, the stimuli are the signals and instructions with predefined structure (syntax). This

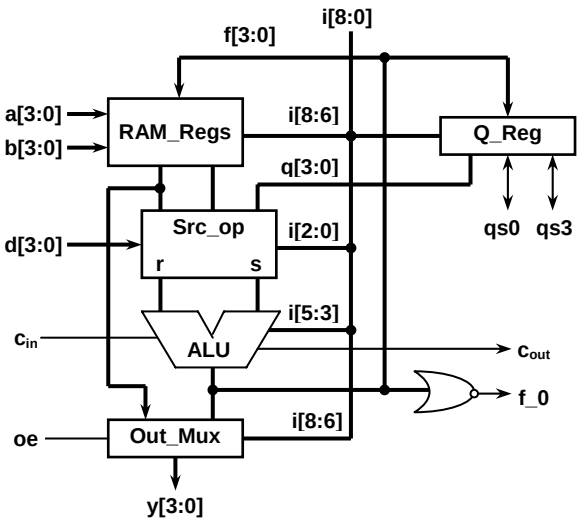

Figure 1. AM2901 block diagram.

property is common for all microprocessors. So, further analysis can be extended to general case.

3. PROPOSED TECHNIQUE

In the study case there is no limitations on the syntax of generated test sequences. But the instructions must be arranged under a certain rule. The objective of test generation process is the synthesis of test (micro)programs with a specific syntax. From this point of view some instructions load the data in the memory's elements, others - process or/and unload the data from them.

Thus, the rule of composition of the test program structure can be formulated by the next paradigm:

> "*data load* → *inherently data process* → *unload* (*and analysis*) *of results*". (4)

The rule (4) expresses the semantic aspect of the objective of random test programs generation. In the other words, rule (4) reflects the stochastic generation process of test programs with data dependency and can be accepted as a link between functional (behavioral) model of the MP [31] and rules of construction of TPG, proposed in [29].

More sophisticated type of dependences can be introduced: *primary*, *data*, *instructional*, and *functional*. Primary (structural) dependence defines the syntax of instructions. Data dependence appears when is needed to set up the transition of data between the instructions. There are 3 types of data dependences: read after write, write after read, and write after write. Data dependence frequently occurs in pipeline design. Instructional dependence defines the link between MP specific instructions, e.g. push-pop, loop-exit, call-return, etc. Functional (behavioral) dependence is established by an experienced designer on basis of his knowledge of behavior of the design entities (modules).

To provide the condition (4) let us represent the component parts of instruction, i.e. ALU functions and operands, by a graphical images (pictograms). The pictogram of component consists of terminal and internal nodes that represent the elements of memory (registers) and ALU functions (symbol ∗). There are 8 ALU source+function pictograms: s0..s7, and 8 destination pictograms: d0..d7 (see Tables 6-1, 6-2 and 6-3 in [26]). The arcs mean transition of data. The zigzag-arc means selection of the RAM word.

The pictograms are connected ("glued") by suitable terminal nodes (like in puzzle-game). The gluing of pictograms is performed in the following way: source with destination creates an instruction

structure (word in the sentence), and resulted structure is connected with another instruction

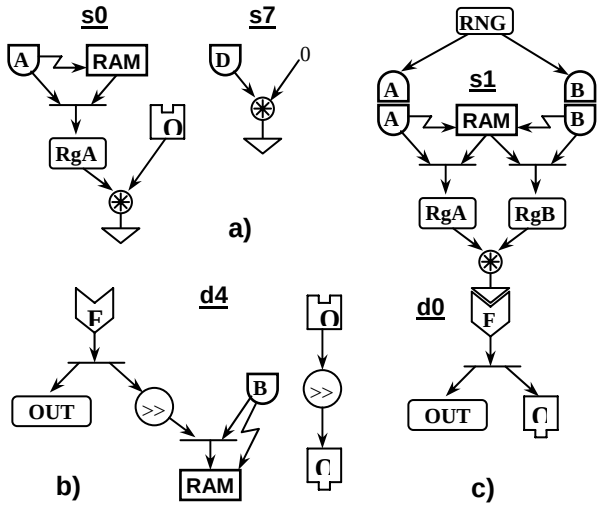

Figure 2. Pictograms of AM2901 μ-instructions: (a) source operand and ALU function, (b) ALU destination, and (c) example of "gluing" pictogram.

structure, thus "building" a sentence, i.e. test program. This process can be executed recursively. All connections are performed according rule (4). Some typical pictograms and a fragment of glued pictogram are shown in figure 2. The node RNG in figure 2, c) means a random number generator.

Introduced pictograms can be used prototype for wizard tools of automation of synthesis of the test program stochastic generator. This illustrative representation of the structure (syntax) of generated sequences is an intermediary step for jump to formal synthesis of generation grammar of the random test programs.

4. SYNTHESIS OF THE TEST PROGRAM GENERATOR

The technique of synthesis of the syntax test generator, inclusively generation of random test cases, is well known for compiler testing [27,28]. The test cases generator is controlled by programming language syntactic diagram (SD). And the SD is needed for a MP to generate the syntactic correct test programs. But the MP specification doesn't have such SD. Many efforts must be made to construct (synthesis) a syntactically (and, may be, semantically) correct model of the random TPG [29]. But in our study case the MP is a simple one. Accepting the rule (4) and assuming the degree=2 for data dependence it was synthesized the random syntactic tree, shown in figure 3.

In the figure 3 callout !(..) mean equiprobable generation of item and the non marked fan-out branches have uniform probabilities.

A stochastic grammar can be associated with this tree. A grammar is defined by a 4-tuple $G=(V_N, V_T, R, S)$ where V_N and V_T are nonempty sets of terminal and nonterminal symbols, respectively. The symbol S, $S \in V_N$ is called the starting symbol and $V=V_N \cup V_T$ is the vocabulary of G. The finite nonempty set R of $(V^*V_NV^*)\times V^*$ is called production rules. In a stochastic grammar G_s with each production $\alpha_i \to \beta_{ij}$ is associated a probability p_{ij}, where α and β are strings of symbols over the vocabulary, $0< p_{ij} \le 1$, $1 \le i \le k$, $1 \le j \le m_i$, $\sum_{j=1}^{m_i} p_{ij} = 1$.

For the study case we have the next stochastic grammar G_{St}:

V_N = {S, A, B, C, D, E, F, H, Z}
V_T = {s0,…,s7, d0,…,d7, 0,…,7}
$S = S$

$$R = \{ S \xrightarrow{p_{SA}} d3'0's7,\ S \xrightarrow{p_{SB}} BOC,$$
$$B \xrightarrow{1/7} s0,\ B \xrightarrow{1/7} s1,\ \ldots,\ B \xrightarrow{1/7} s6,$$
$$O \xrightarrow{1/8} '0',\ O \xrightarrow{1/8} '1',\ldots,\ O \xrightarrow{1/8} '7',$$
$$C \xrightarrow{p_{CD}} d0DC,\ C \xrightarrow{p_{CF}} FB,\ C \xrightarrow{p_{CH}} HS,$$
$$C \xrightarrow{p_{CZ}} d1S,\ D \xrightarrow{1/2} s2OC,\ D \xrightarrow{1/2} s6OC,$$
$$F \xrightarrow{1/4} d2,\ F \xrightarrow{1/4} d3,\ F \xrightarrow{1/4} d5,$$
$$F \xrightarrow{1/4} d7,\ H \xrightarrow{1/2} d4,\ H \xrightarrow{1/2} d6\}. \quad (4)$$

The synthesis of TPG is reduced to definition of generator grammar production rules. The syntax of grammar (structure of TPG) can be synthesized (builded), for example, by the top-down recursive

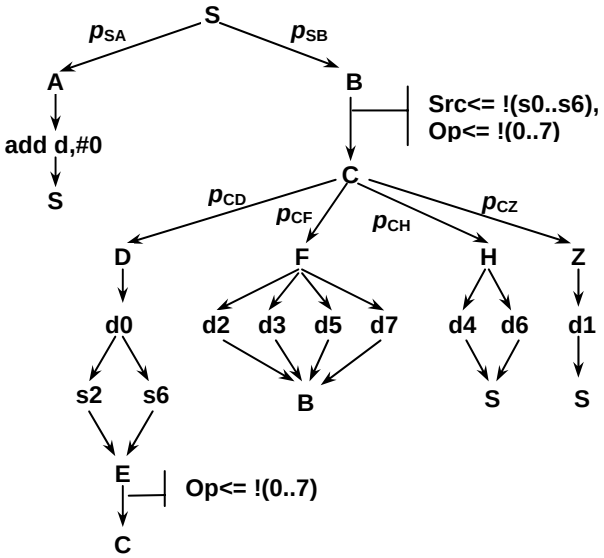

Figure 3. Probabilistic syntactic tree for generator of AM2901.

descent parsing method, analyzed in [29]. The syntax of grammar should guarantee that the derived TP would always be valid.

The repartition of probabilities $\mathcal{P}(R)$ on rules of set R is called syntactic style [30]. In accordance with Chomsky classification the grammar G_{St} is of type 2, i.e. is noncontextual (context-free). The grammar G_{St} defines the structure of the random (weighted) test program generator for verification of the AM2901 slice-MP. In the HDL language, in particular VHDL, is needed to have the corresponding mechanism to produce a sentence in grammar G_{St}. Also note that some transition probabilities have the predefined values and another probabilities, such as $p_{S(\bullet)}$ and $p_{C(\bullet)}$, are undefined. The undefined probabilities will be defined later.

5. WEIGHTED CASE STATEMENT

The weighted case statement differ form the classical one by that in accordance with predefined repartition of probabilities the selector variable get a value from the sample of numbers. It is obvious that the existing linguistic tools can emulate such weighted case. To do this, initially is needed to generate the value of selector, then, after this, to jump on the corresponding variant.

5.1. Model

The model of weighted number generator is the probabilistic binary tree (P-tree). Figure 4 depicts the binary tree of code of a 2-bit number and corresponding to it P-tree. The probability of an outcome is the product of the probabilities on the path from the root to the vertex. Starting from the known repartition of the probabilities on vertex (leafs) of the P-tree it is easy to restore the

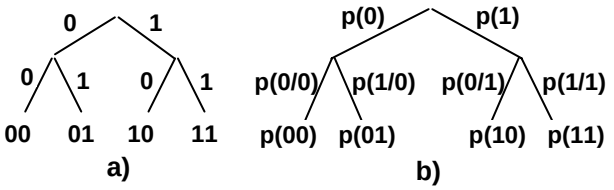

Figure 4. Binary (a) and probabilistic (b) trees of the 2-bit number.

conditional probabilities on each tree's branch.

From the practical point of view it is more suitable to represent the P-tree by a weighted binary tree (W-tree). In this case the weighted choice of a bit value consists in performing of one VHDL-statement:

```
if Weight> LFSR then return('1'); else return('0');   (5)
```

where LFSR is a state of the linear feedback shift

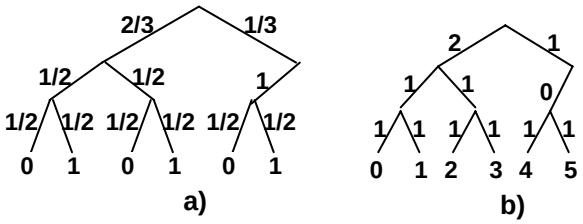

Figure 5. P-tree (a) and W-tree (b) of equiprobable generation of numbers from 0 to 5.

register of size n, $0 \le \text{Weight} \le 2^n-1$

5.2. Implementation

The weighted generation of a number is based on recursive execution of statement (5). For example, if it is needed to equiprobable select of a number from 0 to 5, then the corresponding P-tree and W-tree will looks as it is shown in figure 5. Note that the arcs of W-tree in figure 5 (b) are labeled by the *relative* weights.

On implementing the W-tree, each of its level is coded by a string of weights. Because each fan-out node contains the complementary probabilities then the i-th string contains 2^i weights, namely weight of the right branch, where i= 0,…,r-1,

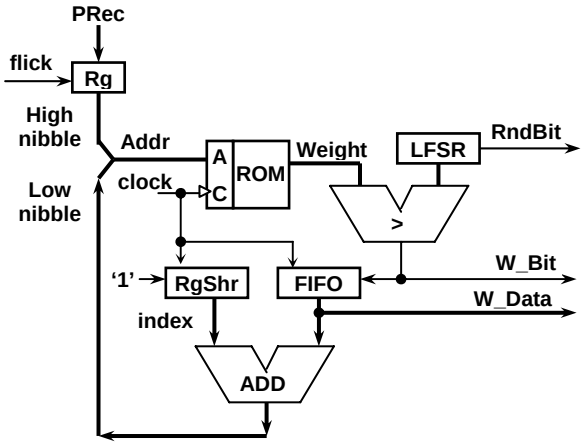

Figure 6. Block diagram of the weighted number generator.

$r=\lceil \log_2 N \rceil$, N is the maximum value of the number. The resulted record of strings is stored in the (RAM) array from where the weights are conditionally read. Thus, was described the (recursive) weighted number generator (WNG) unit, which diagram is shown in figure 6.

The behavior of WNG-unit is the following. Let be the W-tree shown in figure 5, b) and size of

LFSR equal to n=10. So, the corresponding record of weights is:

levels	**0:**	**341**			
	1:	**512,**	**0**		
	2:	**512,**	**512,**	**512,**	**0**

which is written in the ROM as **PRec**=0, where **PRec** points to the base address of a record.

Functioning of the WNG-unit starts with flicking the feeding value of the PRec, that is the high nibble(pointer to) of the record address. Also, the flick signal reset the shift registers RgShr and FIFO. The register RgShr indexes the string in the record: 0, 1, 3, 7 etc. The register FIFO indexes the relative address in the string. The sum of RgShr and FIFO contents form the low nibble of the weight absolute address in the record.

On the clock rising edge the first weight 341 is read from ROM. The comparator (>) applies the rule (5) and compares this weight with the LFSR state. The returned value of the W_Bit is stored in the FIFO. Next weight that can be read is 512 or 0. After r clocks of time FIFO will contains the resulted binary code W_Data of the number. This number will be the value of the case statement selector. The RndBit is a logic value, '0' or '1', generated with the probability 1/2.

In such a way can be generated the number with an arbitrary distribution of the probabilities. The discrepancy between the expected distribution and the one resulted from W-tree model (or generated by WNG-unit) will depend on the measure scale, i.e. on the weight (the same LFSR)

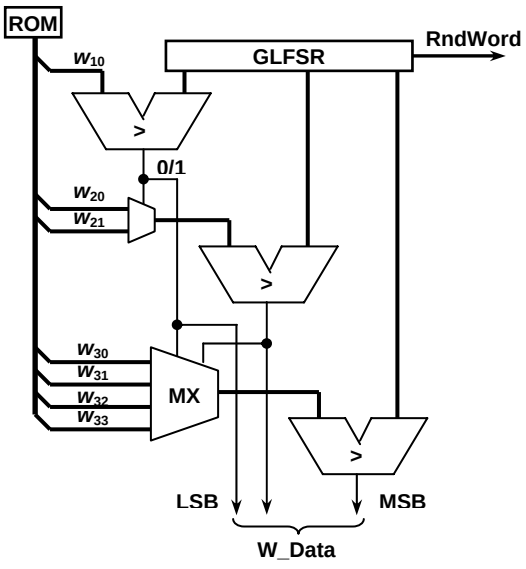

Figure 7. Block diagram of the concurrent WNG.

size (but this problem is the subject for another discussion).

In our implementation we have designed the concurrent (one-shot) version of the WNG unit as well. Figure 7 depicts the scheme of the concurrent weighted number generator (CWGN), which implements a three-level W-tree. On the clock edge the weights $w_{10},\ldots, w_{33}$ are read from ROM and are compared with states of the general linear feedback shift register (GLFSR). GLFSR contains the shift register with n-bitwise cells [32]. Feedback performs the multiplication and addition operations over extended Galois field **GF**(2^n). The GLFSR outputs are the uniform generated n-bitwise data RndWord as well as r-bitwise weighted random data W_Data.

5.3. VHDL-description

If one-shot WNG-unit is used then it is enough to feed the value of PRec and flick it. This operation can be performed in parallel with other statements of the selected case variant. So, state machine design of the test stochastic generator encoded in the VHDL language will be the following:

```
process (clock, reset, W_Data)
begin
 if reset = '1' then  State <= S;
 elsif rising_edge( clock) then
  case State is
   when S=>
      clkAM2901<= '1'; -- generate clock of time for MP
      flick<= '1';
     --equiprobable jump
      if     RndBit='0'  then   State<= A;
      else State<= B;       --if RndBit ='1'
      end if;
   when A=>
      clkAM2901<= '0';
      flick<= '1';
      dst_cod<= ramf;      -- destination code
      op_cod<= add;        -- ALU function code
      src_cod<= dz;        -- source code
      State<= S;
   when B=>
      clkAM2901<= '1';
      flick<= '0';
      PRec<= "000"; --set the pointer to !( 0..3)
      State<= C;
   when C=>
      clkAM2901<= '0';
      flick<= '1';
     --weighted (equiprobable) case selection
      if     W_Data ="000"  then State<= D;
      elsif  W_Data ="001" then State<= G;
      elsif  W_Data ="010" then State<= H;
      else State<= Z;       -- if  W_Data ="011"
      end if;
     ……..
  end case;
 end if;
end process;
```

In above listing is presented a fragment of the state machine description that implements the test

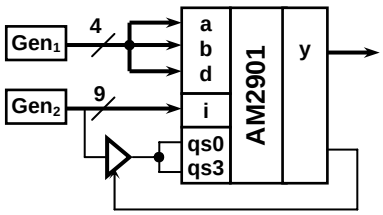

Figure 8. Test bench scheme of AM2901.

program stochastic generator described by grammar G_{St}.

6. TEST EXPERIMENTS AND RESULTS

The efficiency of syntactic approach will be estimated in comparison with other methods of generation, namely, deterministic and pure random.

In [26] is described an AM2901 deterministic test bench based on procedural approach. In the Quartus waveform editor we have created the verification test cases that must be generated by this test bench. We run the simulation. For 185 executed instructions the resulted simulation coverage was equal to 93,73%.

Further, we have implemented a simple test bench scheme shown in figure 8. Three types of test experiments were performed where generators Gen_1 and Gen_2 were counters, or maximum-length sequence generator, or syntactically controlled stochastic generator (Gen_2). Note that the transition probabilities $p_{S(\bullet)}$ and $p_{C(\bullet)}$ in G_{St} were set up equiprobable.

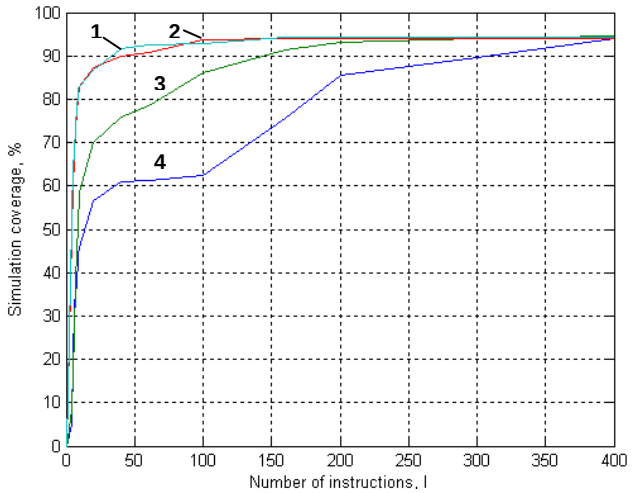

Figure 9. The AM2901 simulation coverage for instructions generated by:
1, 2 – stochastic generator;
3 – pure random generator;
4 – counter generator.

Test experiments were executed by increasing of simulation time (parameter End Time in the CAD Quartus) step-by-step. The obtained values of simulation coverage are plotted in the figure 9, where stochastic generator is the controlled test program generator which algorithm of functioning is defined by stochastic grammar G_{St}.

The counter as Gen_2 tries all possible states like the LFSR as Gen_2. At the same time, comparing curve 4 with 3, 2 and 1 it is easy to state that the simulation coverage of the counter is worse then for the random testing. It can be stated also that the further improvement of verification quality by random stimulus can be achieved only by qualitative change of the random generator (compare curves 1 and 2 with 3).

Also was established that the variation of transition probabilities $p_{S(\bullet)}$ and $p_{C(\bullet)}$ doesn't give an essential improvement of the simulation coverage (see curves 1 and 2). This is because the AM2901 microprocessor has a simple architecture that is "insensible" to the stylistic of generated stimuli.

Now, make the comparison of experimental results with the theoretical model. Assume that apriori probability p_e in (3) is the relative frequency of "switching" on-off of the elements of memory, i.e. the registers. The AM2901 microprocessor has 16+1=17 registers. So, the number of switching is twiced and is equal to 34. Then the probability p_e of event of switching of the registers is equal to $1/34 \approx 0{,}029$. In the graphic 1 (or 2), shown in figure 8, for test length l equal, for example, to 100, we obtain the value of confidence level λ equal to 0,93. Thus, in accordance with relation (3) the expected probability $P_e(l)$ of exercising of the MP registers on feeding l instructions is approximately equal to

$$\frac{1}{100}\ln\frac{1}{1-0{,}93} \approx 0{,}027\,. \qquad (5)$$

Deviation of experimental test length from expected one constitutes about 6% that is the acceptable discrepancy between theoretical model and experimental results.

In spite of achieved high level of exercising of the nodes of design netlist, some nodes remained unexercised. Therefore, for successful completion of design verification, the CAD-system should have the "ability" to summarize the list of nodes not yet exercised.

CONCLUSIONS

In this paper a syntactic approach to synthesis of stochastic program generator has been presented.

Theoretical and design issues have been analyzed. Experimental results justify the elaborated theoretical model. The proposed design solutions and VHDL constructions are in accordance with the proposals suggested recently in [13].

Further improvement of the stochastic test program generator can be reached by tuning of the transition probabilities repartition of the grammar. In fact, if suppose that it is given an arbitrary probabilities repartition on events E then the relation (3) will be transformed to:

$$l \cong \frac{1}{\min\limits_{e \in E}\{p_e\}} \ln \frac{1}{1-\lambda}. \qquad (6)$$

So, the task of test quality improvement is reduced to increase of low bound of repartition. Introduction of the Markov chain can successfully solve this new problem (because it is known that a noncontextual stochastic grammar can be adequately represented by a Markov process). In this case the principle of maximization of the entropy of Markov process should be applied to increase the low bound of analyzed repartition, and, so, to decrease the length of test-verification programs. From the other hand, when TPG structure corresponds to a stochastic context-free grammar then the branching Markov process is needed for analysis of the style of generated sentences (see birth and death processes).

What is the attractive aspect of the syntactic approach? Firstly, probabilities provide the varieties of test programs. Secondly, greater effect of synthesis of an AGP can be achieved by automation of the procedure of definition of the generator grammar rules.

The proposed tools facilitate the control of the process of the test programs stochastic generation and are ready for practical using in verification of microprocessors or complex systems on chips. These tools are good for synthesizable design as well as for (presynthesis) simulation design.

Also it is obvious, that for successful of synthesis of test program generators it is necessary to supply the hardware description language with constructions which provide not only specification of structural features, but also the properties (attributes) of behavior of microprocessor instructions.

Need to notice that the task of mapping of MP-structure to TPG-structure is not yet studied up to end. Therefore we hope that the proposed syntactic approach to construction of test program generators will accelerate the development of methods and tools of simulation-based verification of microprocessors.

References

***1. Kantrowitz M.**, **Noack L. M.** I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor. Proc. Design Automation Conf., 1996, pp. 325–330.*

***2. Walter J., Leenstra J., Dottling G., Leppla B., Munster H.-J., Kark K., Wile B.** Hierarchical Random Simulation Approach for the Verification of S/390 CMOS Multiprocessors. – Proceedings of the 34th DAC, 1997, pp.89-94.*

***3. Abts D., Roberts M.** Verifying Large Scale Multiprocessors Using an Abstract Verification Environment, DAC 1999, pp.163-168.*

***4. Khailany B., Dally W. J., Chang A., Kapasi U. J., Namkoong J., Towles B.** VLSI Design and Verification of the Imagine Processor Proceedings of the 2002 International Conference on Computer Design, 2002, pp.289-294.*

***5. Behm M., Ludden J., Lichtenstein Y., Rimon M., Vinov M.** Industrial Experience with Test Generation Languages for Processor Verification DAC 2004, June 7–11, 2004, San Diego, California, USA.- 2004, pp.36-40.*

***6. Wagner I., Bertacco V., Austin T.**. Stress Test: An automatic approach to test generation via activity monitors. DAC, Proceedings of Design Automation Conference, 2005, pp.783-788.*

***7. Bhaskar K.U., Prasanth M., Chandramouli G., Kamakoti V.** A universal random test generator for functional verification of microprocessors and system-on-chip VLSI Design, 2005. 18th International Conference on Volume , Issue , 3-7 Jan. 2005, pp. 207 – 212.*

***8. Henrique J.-P.** Verification of STMicroelectronics Configurable Processor CDN Live EMEA June 2006. [Online]. Available: http://www.cdnuser.org*

***9. Mallesham Boini,** Complex SoC Verification Using and ARM Processor – Design Strategies and Methodologies, Information Quarterly , Volume 5, Number 4, 2006, pp. 62- 65.*

***10. Poe E.A**. Theory of Operation Basic and Advanced Random Test Generators (2004). [Online]. Available: http:// www.obsidiansoft.com/files/operation.pdf*

***11. Rosebrugh C.** Using Vera and Constrained-Random Verification to Improve Design Ware Core Quality – The Synopsys Verification Avenue Technical Bulletin, Vol. 4, issue 4, December 2004. (2004). [Online]. Available: http:// www.open-vera.com/technical/*

12. *[Online]. Available: http:// www.aldec.com/solutions/hdlverification/*

13. Lewis J. *Accellera VHDL-TC Extensions-SC Randomization (2007). [Online]. Available: http:// www.accelera.org/apps/group_public/download.php/905/Randomization-V1.pdf*

14. Freitas A. *Shadow model and coverage driven processor verification using SystemVerilog [Online]. Available: http://www.edatechforum.com /journal/june2007shadow.cfm/*

15. Aharon A., Bar-David A., Dorfman B., Gofman E., Leibowitz M., and Schwartzburd V. *Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator. IBM Systems Journal, 1991,pp. 527–538.*

16. Vemuri R. and Kalyanaraman R.. *Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming. IEEE Trans. on VLSI, 1995, pp. 201–214.*

17. Beizer B., *Software Testing Techniques, Second Ed., Van Nostrand Reinhold, 1990. – 550 P.*

18. Cheng K.-T. and Krishnakumar A. S., *Automatic functional test bench generation using the extended finite state machine model, Design Automation Conference, 1993, pp. 1–6.*

19. Ho R. C., Yang C. H., Horowitz M. A., and Dill D. L., *Architecture validation for processors, International Symposium on Computer Architecture, 1995,.pp. 404–413.*

20. Lee D. and Yannakakis M. *Principles and methods of testing finite state machines - a survey, IEEE Transactions on Computers, vol. 84, pp. 1090–1123, August 1996.*

21. Moundanos D., Abraham J. A., and Hoskote Y. V. *Abstraction techniques for validation coverage analysis and test generation. IEEE Trans. Computers, vol. 47, no. 1, January 1998, pp. 2–14.*

22. Devadas S., Ghosh A., and Keutzer K. *Observability-based code coverage metric for functional simulation. Proc. Int. Conf. Computer-Aided Design, 1996, pp. 418–425.*

23. Fallah F., Devadas S., and Keutzer K.. *OCCOM: Efficient computation of observability-based code coverage metric for functional simulation. Proc. Design Automation Conf., 1998, pp. 152–157.*

24. Taylor S., Quinn M., Brown D., Dohm N., Hildebrandt S., Huggins J., and Ramey C.. *Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor - the DEC Alpha 21264 microprocessor. Proc. Design Automation Conf., 1998, pp. 638–643.*

25. .Fu K.S. *Syntactic pattern recognition and applications, Englewood Cliffs, NJ, Prentice Hall, 1982. – 596 P.*

26. Skahill K. *VHDL for Programmable Logic, Addison-Wesley Publ., 1996, - 594 P.*

27. Hanford K.V. *Automatic generation of test cases, IBM Systems Journal, vol. 9, No.4, 1970, pp.242-257.*

28. Bird D.L. and Munoz C.U. *Automatic generation of random self-checking test cases, IBM Systems Journal, vol. 22, No.3, 1983, pp.229-245.*

29. Wu L.-M., Wang K., and Chiu C.-Y. *A BNF-Based Automatic Test Program Generator for Compatible Microprocessor Verification, ACM Trans. on Design Automation of Electronic Systems, vol.9, No. 1, 2004, pp.105-132.*

30. Grenander U. *Pattern Analysis, Vol. II, Springer Verlag, 1978.*

31. Thatte S.M., and Abraham J.A. *Test generation for microprocessors. IEEE Trans. Comput. C-29, No. 6, 1980, pp.429–441.*

32. Pradhan D.K., Chatterjee M. *GLFSR– a new test pattern generator for built-in-self-test. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No.2, 1999, pp. 238-247.*

Recommended for publication: 25.02.2011