УДК 004.415.53
DOI https://doi.org/10.32782/2663-5941/2026.1.2/48

Хаханова І.В.
https://orcid.org/0000-0002-8319-0430
Харківський національний університет радіоелектроніки

Хаханов І.В.
https://orcid.org/0009-0009-3449-9595
Харківський національний університет радіоелектроніки

Філіппенко І.В.
https://orcid.org/0000-0002-3584-2107
Харківський національний університет радіоелектроніки

Філіппенко О.І.
https://orcid.org/0000-0003-4616-250X
Харківський національний університет радіоелектроніки

ВІДТВОРЮВАНІСТЬ ПСЕВДОВИПАДКОВИХ ТЕСТІВ У SYSTEMVERILOG

Тестування цифрових систем у процесі їхньої функціональної верифікації є критично необхідним, оскільки воно гарантує коректність роботи складних проектів. Одним із ключових підходів є метод псевдовипадкового генерування тестів. Виникає критична потреба в забезпеченні стабільності та повної відтворюваності таких псевдовипадкових тестів при фіксованому значенні Seed, незалежно від паралельного виконання процесів, ієрархічної структури моделі чи можливих змін у архітектурі TestBench. Навіть невеликі зміни коду можуть порушувати стійкість псевдовипадкових послідовностей SystemVerilog, роблячи тести невідтворюваними та ускладнюючи налагодження. Робота присвячена дослідженню особливостей функціонування операторів мови SystemVerilog з метою забезпечення стабільності і відтворюваності псевдовипадкових тестів при фіксованому Seed, незалежно від паралельності процесів, ієрархії моделі та змін у структурі TestBench. Основний акцент дослідження зосереджено на детальному вивченні впливу структури коду, зокрема порядку та ієрархії процесів, на відтворюваність і стабільність згенерованих послідовностей. Проведений аналіз дозволив сформулювати особливості роботи системних функцій та методів генерації випадкових значень у SystemVerilog визначив ключові підходи до їх застосування, що сприяють забезпеченню стійкості та відтворюваності послідовностей випадкових даних. Дослідження було спрямоване на теоретичний аналіз, формування методів та практичних рекомендацій, пропонуючи комплексний підхід до вирішення проблеми стійкості псевдовипадкових тестів у SystemVerilog. Проведене теоретичне та експериментальне дослідження продемонструвало, що стійкість послідовностей, що генеруються засобами рандомізації, визначається не лише вибором початкового стану генератора псевдовипадкових чисел, а й структурними особливостями тестового коду, включаючи порядок та ієрархію процесів.

Ключові слова: *псевдовипадкови тести, Seed, SystemVerilog, методи формування випадкових тестів, рандомізація, ПЛІС.*

Постановка проблеми. Одним із ключових методів функціональної верифікації цифрових систем є псевдовипадкове генерування тестів, яке дозволяє алгоритмічно створювати тестові сценарії [IEEE 1800-2017, розділ 18.13]. Такі тести охоплюють широкий спектр вхідних даних і станів системи, включно з рідкісними та граничними випадками (corner cases), важко передбачуваними вручну.

При псевдовипадковому генеруванні тестів SystemVerilog початкове значення (Seed) – це числове значення, яке ініціалізує генератор псевдови-

© Хаханова І.В., Хаханов І.В., Філіппенко І.В. та ін., 2026
Стаття поширюється на умовах ліцензії відкритого доступу CC BY 4.0

падкових чисел (ДПСЧ). Воно визначає початкову точку для алгоритму, що генерує послідовність псевдовипадкових чисел, забезпечуючи відтворюваність генерованих послідовностей. Таким чином, Seed – це ключ до відтворюваності псевдовипадкових тестів. Фіксоване початкове значення Seed дозволяє генерувати однакові послідовності чисел при повторних запусках, що спрощує налагодження та верифікацію. Стійкість або стабільність таких послідовностей критично важлива для відтворюваності тестів, налагодження та забезпечення однакових результатів при багаторазових симуляціях. Стійкість псевдовипадкових послідовностей SystemVerilog передбачає їхню здатність залишатися відтворюваними і передбачуваними при фіксованому Seed, незважаючи на зміни в коді або умовах виконання тесту.

Теоретично, для повного відтворення послідовності достатньо знати Seed, оскільки ДПСЧ є детермінованим. Проте в складних тестових наборах після виправлення помилок або модифікації коду можуть виникати проблеми з відтворюваністю. У регресійних тестах часто використовують множинні запуски з різними Seed, а їх повторне задання в різних частинах тесту може призводити до непередбачуваних змін стану генератора.

У результаті налагодження може виникнути ситуація, коли початкова тестова послідовність більше не відтворюється, навіть за однакових умов запуску. Це порушує стійкість псевдовипадкових послідовностей і унеможливлює впевнене підтвердження того, що помилка була дійсно усунена, а не зникла через зміну конфігурації тесту. Таким чином, забезпечення відтворюваності є критично важливим для ефективної функціональної верифікації та аналізу результатів тестування.

Аналіз останніх досліджень і публікацій. Аналіз доступної літератури показує, що питання відтворюваності та стійкості псевдовипадкових тестів у SystemVerilog висвітлені фрагментарно. Існуючі дослідження недостатньо приділяють уваги впливу ієрархічних та паралельних процесів на стійкість рандомізації. Вони присвячені відтворюваності та стійкості ізольовано, без інтеграції теоретичного аналізу з практичними рекомендаціями.

Основним документом за цією тематикою є стандарт IEEE 1800-2017 (SystemVerilog Language Reference Manual). У [1] описуються механізми псевдовипадкової генерації SystemVerilog, включаючи такі функції як $random, $urandom, $dist_* і методи std::randomize() і obj.randomize(). Розділ 18.13 деталізує роботу генератора псевдовипадкових чисел (ГПСЧ) та управління початковим значенням (Seed).

Стандарт надає формальне визначення операторів рандомізації, включаючи їхню детерміновану поведінку при фіксованому Seed. Це основа для аналізу відтворюваності тестів, оскільки стандарт гарантує ідентичність послідовностей за однакових початкових умов. Однак [1] не поглиблюється у практичні аспекти стійкості при змінах коду чи ієрархії процесів.

Оновлений стандарт IEEE 1800-2023, опублікований у грудні 2023 року, є ключовим джерелом для розуміння механізмів псевдовипадкової генерації SystemVerilog. Розділ, присвячений рандомізації (18.13), описує функції $random, $urandom, методи std::randomize() та obj.randomize(), а також управління початковим значенням (Seed). Стандарт уточнює, як забезпечується детермінованість послідовностей при фіксованому Seed, та обговорює вплив паралельних процесів на ГПСЧ. Однак [2] не надає практичних прикладів чи рекомендацій щодо управління стійкістю у складних тестових наборах, що робить його доповненням до практичних досліджень.

У [3] безпосередньо приділяє увагу щодо проблеми відтворюваності псевдовипадкових тестів, пояснюючи, як правильно використовувати Seed та механізми рандомізації в SystemVerilog для забезпечення детермінованості послідовностей. Обговорюється вплив паралельних процесів та ієрархії на генерацію випадкових чисел. Наголошується на необхідності обережного використання $random у кількох процесах через їх взаємний вплив. Автор пропонує підходи до структурування тестів, які мінімізують порушення відтворюваності. Однак у [3] не поглиблюється у специфічні проблеми стійкості при додаванні нових процесів чи зміні ієрархії.

Робота [4] надає фундаментальне розуміння механізмів рандомізації в SystemVerilog, включаючи різницю між $random і $urandom, що робить її цінним джерелом для обґрунтування теоретичних аспектів даної теми. Наведено приклади коду та пояснення щодо створення testbench у SystemVerilog. Але докладно не розглядається вплив структури коду (наприклад, порядку та ієрархії процесів) на відтворюваність.

Робота [5] обговорює загальні проблеми відтворюваності псевдовипадкових тестів у контексті функціональної верифікації. Вона наголошує на важливості фіксованого Seed для повторю-

ваності тестів, але не надає детального аналізу впливу структури коду. Автор [5] формує теоретичну базу розуміння відтворюваності, але з розглядається вплив ієрархічних чи паралельних процесів на стійкість послідовностей. Також відсутні приклади моделювання, не наводяться конкретні рекомендації щодо управління Seed у складних тестових наборах. Не приділяється достатньої уваги впливу структури коду (наприклад, порядку та ієрархії процесів) на відтворюваність тестів. Це створює пропуск у розумінні, як зміни в коді можуть порушувати стійкість послідовностей.

Дослідження [6] фокусується на використанні Universal Verification Methodology (UVM) для функціональної верифікації. Воно підкреслює переваги UVM в управлінні псевдовипадковими тестами, але не надає детальних прикладів управління Seed у різних сценаріях. Робота зосереджена на UVM, але не пропонує конкретних прикладів управління Seed у різних сценаріях, що обмежує її застосовність для розробників, які працюють з паралельними чи ієрархічними процесами. Не наводяться конкретні приклади та аналізу впливу змін коду на відтворюваність, що обмежує глибину дослідження.

Постановка завдання. Дане дослідження спрямоване на теоретичне обґрунтування, аналіз методів та формування практичних рекомендацій у галузі функціональної верифікації цифрових систем. Метою роботи є вивчення особливостей роботи операторів мови SystemVerilog, призначених для генерування випадкових чисел, з точки зору стабільності процесів рандомізації, а також аналіз підходів до розробки моделей, що забезпечують відтворюваність тестових наборів у процесі верифікації. Основна увага приділяється дослідженню впливу структури програмного коду, зокрема порядку виконання та ієрархії процесів, на відтворюваність і стабільність генерованих послідовностей.

Виклад основного матеріалу.

1. Генерування випадкових чисел в SystemVerilog

У SystemVerilog оператори для генерації випадкових чисел можна поділити на дві групи за принципом використання початкового числа seed: функції на основі ймовірнісних розподілів (standard probabilistic functions) та функції з обмеженнями (constrained pseudo-random value generation).

До першої групи належать $random та функції $dist_*, що реалізують різні стандартні розподіли. Де $random успадкована з Verilog і генерує 32-бітові знакові числа, тоді як $dist_* додані в SystemVerilog. Їх алгоритми визначені у стандарті на C, що забезпечує однакову поведінку у різних симуляторах.

До другої групи відносяться $urandom, $urandom_range, std::randomize() та метод randomize() для об'єктів класів SystemVerilog. Стандарт визначає лише основні принципи їх роботи, залишаючи деталі реалізації розробникам симуляторів, що може призводити до відмінностей у поведінці на різних платформах.

1.1. Формування послідовностей функціями з ймовірнісним розподілом. За замовчуванням функції цієї групи використовують єдине глобальне початкове число (Seed), від якого формуються всі їхні значення. Як приклад розглянемо функцію $random. Якщо в коді є кілька паралельних процесів, що використовують $random, виклики цієї функції в одному процесі можуть впливати на послідовність даних в іншому (див. рис. 1). Таким чином, навіть невеликі зміни в TestBench можуть призвести до втрати відтворюваності тесту.

У SystemVerilog під процесом розуміють різні оператори, що формують незалежні потоки команд, зокрема процедурні оператори initial, наведені в лістингу 1. У першому варіанті реалізації (лістинг 1, а) використовується один процес, у якому для генерування даних застосовується функція $random. Приклад сформованої послідовності наведено в лістингу 1, б); вона відтворюється при повторних викликах за умови використання того самого Seed.

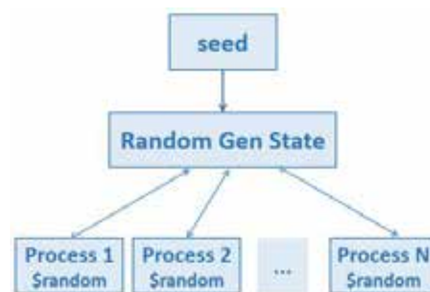

Рис 1. Залежність $random від одного джерела формування послідовності.

У другому варіанті (лістинг 1, в) використано два блоки initial, кожен з яких викликає $random. Результати моделювання (лістинг 1, г) показують, що загальна послідовність чисел збігається з першим варіантом, однак значення змінної r1 відрізняються. Це зумовлено тим, що після першого виклику $random два наступні виклики

виконуються в іншому процесі, унаслідок чого друге та третє числа послідовності присвоюються r2.

Таким чином, наочно показано, що паралельні процеси з викликами $random взаємно впливають один на одного. Для стабілізації результатів можна задавати окремі seed для кожного процесу; аналогічний механізм уже реалізовано у функціях другої групи рандомізації SystemVerilog.

Лістинг 1. Приклад моделей із функцією $random

а) Один процес із $random

```
module random_ex();
  int r1;

  initial
    repeat(8) begin
      r1 = $random;    $display("R1 = 0x% 8h",
r1);   #1;
    end
  endmodule
```

б) Результати моделювання першого варіанта реалізації (а):

```
R1 = 0x12153524
R1 = 0xc0895e81
R1 = 0x8484d609
R1 = 0xb1f05663
R1 = 0x06b97b0d
R1 = 0x46df998d
R1 = 0xb2c28465
R1 = 0x89375212
```

в) Два процеси з $random.

```
module random_ex();
  int r1, r2;
  initial
    repeat(6)  begin
        r1 = $random;   $display("R1 = 0x%
8h", r1);   #1;
    end

  initial
    repeat(2) begin
    r2 = $random;        $display("R2 = 0x%
8h", r2);
  end
endmodule
```

г) Результати моделювання другого варіанта реалізації (в) :

```
R1 = 0x12153524 // Перше число, взяте першим
процесом
R2 = 0xc0895e81 // Друге число, взяте другим
процесом
R2 = 0x8484d609 // Третє число, взяте другим
процесом
R1 = 0xb1f05663 // Четверте число, взяте пер-
шим процесом
R1 = 0x06b97b0d // П'яте число
R1 = 0x46df998d // Шосте число
R1 = 0xb2c28465 // Сьоме число
R1 = 0x89375212 // Восьме число
```

1.2. Методи рандомізації, що використовують обмеження. Друга група методів ґрунтується на механізмі RNG (Random Number Generator), у якому кожен процес має власний генератор випадкових чисел із початковим станом (Seed), що задається під час створення процесу. У результаті кожен процес формує незалежну послідовність випадкових чисел, на яку не впливають виклики аналогічних функцій в інших процесах. Вбудований у SystemVerilog клас process забезпечує керування процесами, задання Seed та отримання стану RNG за допомогою методу get_randstate().

Поведінка цієї групи продемонстрована на прикладі функції $urandom, що генерує 32-бітові беззнакові числа. На рисунку 2 наведено схему роботи N паралельних процесів, кожен з яких при створенні отримує власний RNG з унікальним початковим Seed.

Рис 2. Формування RNG для кожного процесу

Лістинг 2 містить два приклади моделей з одним процесом (2, а)) та з двома паралельними процесами (2, б)). З результатів моделювання можна побачити, що додавання другого процесу не вплинуло на послідовність чисел, що формуються першим процесом (сигнал R1 на результатах а) та б)). Це доводить стабільність рандомізації в процесі використання $urandom.

Стабільність рандомізації в рамках усієї моделі для цієї групи операторів SystemVerilog залежить від двох факторів: порядку створення процесів та ієрархії процесів.

Лістинг 2. Приклад використання функції $urandom

а) Один процес із $urandom.

```
module random_ex();
    int r1;   process p1;

  initial begin
    p1 = process::self();   $display("Process 1: rand_state = %s", p1.get_randstate());
    repeat(8)begin
         r1 = $iurandom;    $display("R1 = 0x% 8h", r1);  #1;
    end
  end

endmodule
```

б) Два процеси з $urandom.

```
module random_ex();
     int r1, r2;     process p1, p2;
initial begin
    p1 = process::self();
$display("Process 1: rand_state = %s", p1.get_randstate());
    repeat(6)  begin
        r1 = $urandom;   $display("R1 = 0x% 8h", r1);   #1;
    end
  end

  initial begin
    p2 = process::self();
$display("Process 2: rand_state = %s", p2.get_randstate());
    repeat(2)     begin
         r2 = $urandom;      $display("R2 = 0x% 8h", r2);
    end
  end
endmodule
```

Результати моделювання:

а)

```
Process 1: rand_state = svSeed = 1; 209583384;
R1 = 0x63d5b317
R1 = 0x440c06ab
R1 = 0x4be7c468
R1 = 0x61df6913
R1 = 0x016d515e
R1 = 0xa12acbb7
R1 = 0x490ed783
R1 = 0x765bfc75
```

б)

```
Process 1: rand_state = svSeed = 1; 209583384;
R1 = 0x63d5b317
Process 2: rand_state = svSeed = 1; 1624058817;
R2 = 0x1b453960
R2 = 0xd7e56dec
R1 = 0x440c06ab
R1 = 0x4be7c468
R1 = 0x61df6913
R1 = 0x016d515e
R1 = 0xa12acbb7
```

З результатів моделювання видно, що при запуску одного процесу rand_state ініціалізується початковим станом RNG (209583384). Функція $urandom генерує послідовність із восьми 32-бітових беззнакових чисел, яка відтворюється при постійному seed, що підтверджує стабільність генерації в межах одного процесу.

При запуску двох процесів перший з них отримує той самий початковий стан RNG (209583384), що й у попередньому варіанті. Послідовність чисел для r1 збігається з першими шістьма значеннями попереднього варіанту, підтверджуючи, що виклики $urandom у першому процесі не залежать від другого. Другий процес ініціалізується іншим seed (1624058817) і генерує два числа r2, незалежні від першого процесу.

Таким чином, додавання другого процесу не впливає на послідовність r1, оскільки кожен процес використовує власний RNG. Це демонструє стабільність $urandom у паралельних процесах, хоча стійкість результатів залежить від порядку створення процесів та їхньої ієрархії.

1.2.1. Порядок процесів. У цьому підході додавання нового процесу не впливає на генерацію випадкових значень в попередніх процесах, але змінює послідовності у наступних. Це пояснюється тим, що батьківський процес послідовно формує початкові стани RNG для породжених процесів на основі власного RNG, і введення додаткового процесу змінює Seed для наступних генераторів.

Розглянемо модифікацію прикладу з лістингу 2 з доданим процесом P1_1 (листинг 3). Новий процес P1_1 успадковує початковий стан RNG, який раніше використовував процес P2. У результаті P1_1 генерує ту саму послідовність, що раніше належала P2, а процес P2 починає формувати нову послідовність. Значення процесу P1 залишаються незмінними, що підтверджує незалежність раніше створених процесів. Результати моделювання наведено у таблиці 1, де незмінені значення виділені для наочності.

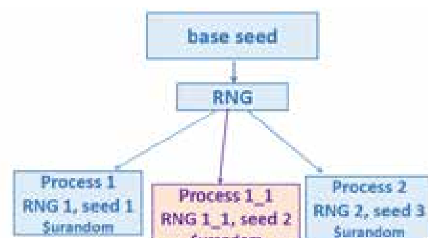

Рис 3. Додавання процесу на одному рівні з попередніми

Лістинг 3. Модуль із доданим процесом

```
module random_ex();
  int r1, r2, r3;
  process p1, p2, p3;

   initial begin: P1
    p1 = process::self();
    $display("Process 1: rand_state = %s",
p1.get_randstate());
    repeat(6) begin
      r1 = $urandom;    $display("R1 = 0x%
8h", r1);     #1;
    end
  end

  initial begin :P1_1
    p3 = process::self();    $display("Process
1_1: rand_state = %s", p3.get_randstate());
    repeat(3)     begin
        r3 = $urandom;    $display("R3 = 0x%
8h", r3);    #1;
   end
  end

  initial begin :P2
     p2 = process::self();
$display("Process 2: rand_state = %s", p2.get_
randstate());
    repeat(2)    begin
        r2 = $urandom;      $display("R2 = 0x%
8h", r2);
    end
  end
endmodule
```

Результати моделювання з доданим процесом

	Process 1	Process 1_1	Process 2
seed	1	1	1
RNG	209583384	1624058817	409697584
Data	0x63d5b317	0x1b453960	0x809ebd6c
	0x440c06ab	0xd7e56dec	0x69211c9c
	0x4be7c468	0x5886eeab	
	0x61df6913		
	0x016d515e		
	0xa12acbb7		

Результати моделювання підтверджують, що додавання нового процесу P1_1, як описано в лістингу 3, не впливає на послідовність значень, генерованих процесом P1, демонструючи незалежність раніше створених процесів. Одночасно P1_1 успадковує початковий стан RNG, який у вихідній конфігурації належав процесу P2, і формує відповідну послідовність, тоді як P2 генерує нову через зміну свого початкового стану. Цей ефект обумовлений послідовним визначенням батьківським процесом початкових станів RNG для породжених процесів і підкреслює важливість врахування порядку їх створення для забезпечення передбачуваності та відтворюваності результатів у багатопроцесних системах.

1.2.2. Ієрархія процесів. При створенні ієрархічних процесів дочірні успадковують початкові стани RNG від батьківського, що впливає на подальші значення, що генеруються в батьківському процесі, як показано на рисунку 4, де наведена схема формування ієрархічних процесів та їхніх початкових станів RNG.

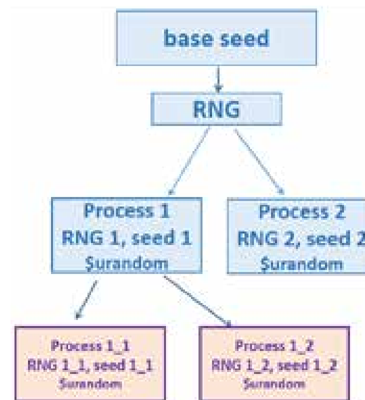

Рис 4. Схема формування ієрархічних процесів

Процес 1 породжує два підпроцеси, реалізовані в моделі (листинг 4) через паралельні виклики функції gen_f у конструкції fork–join. Згідно з даними таблиці 2, створення нових дочірніх процесів не вплинуло на послідовність значень, що генерується процесом 2 (листинг 2). Водночас у процесі 1 було втрачено значення 0x016d515e, яке генератор RNG використав як початковий стан (Seed) для підпроцесів 1_1 та 1_2.

Таким чином, введення ієрархічних процесів змінює послідовність даних, що генерується батьківським процесом, починаючи з моменту їх створення.

Лістинг 4. Приклад із вбудованими ієрархічними процесами

```
module random_ex();
  int r1, r2;
  process p1, p2;

  function void gen_f(int unsigned _i);
    int r0;
    process p0;
    p0 = process::self();   $display("Process
1_%0d: rand_state = %s", _i, p0.get_
randstate());
```

```
    repeat(3)     begin
      r0 = $urandom;     $display("R1_%0d = 
0x%8h", _i, r0);
    end
  endfunction : gen_f

  initial begin
    p1 = process::self();   $display("Process 
1: rand_state = %s", p1.get_randstate());
    repeat(4) begin
      r1 = $urandom;    $display("R1 = 0x%
8h", r1);  i   #1;
    end
    fork
      gen_f(1);
      gen_f(2);
    join
     $display("Process 1: rand_state = %s", 
p1.get_randstate());
    repeat(4)  begin
      r1 = $urandom;    $display("R1 = 0x%
8h", r1);    #1;
    end
  end

  initial begin
    p2 = process::self();
$display("Process 2: rand_state = %s", p2.get_
randstate());
    repeat(2)     begin
    r2 = $urandom;     $display("R2 = 0x% 8h", 
r2);
    end
  end
endmodule
```

Результати симуляції рандомізації ієрархічних процесів

	Process 1	Process 1_1	Process 1_2	Process 2
seed	1	1	1	1
RNG	209583384	2171444226	3512642210	1624058817
Data	0x63d5b317	0xd1b41927	0x221a8a81	0x1b453960
	0x440c06ab	0x229f8dda	0x84ca6502	0xd7e56dec
	0x4be7c468	0xe591e801	0xc8978379	
	0x61df6913			
	~~0x016d515e~~			
	0xa12acbb7			
	0x490ed783			
	0x765bfc75			
	0x1a8f218d			

Створення вкладених процесів змінює послідовність значень, що генерується батьківським процесом після точки їх створення. Для усунення цього ефекту використовується збереження та відновлення стану RNG, як показано в лістингу 5: стан генератора процесу P1 зчитується методом get_randstate() та відновлюється викликом set_randstate(), унаслідок чого вкладені процеси, зокрема виклики gen_f, не впливають на подальшу генерацію даних. Такий підхід широко застосовується в методології UVM для забезпечення стабільності рандомізації тестових компонентів.

Лістинг 5. Приклад збереження та відновлення стану RGN для процесу

```
process p1;
string p_randstate;
…
initial begin
    p1 = process::self();
    repeat(4)begin
       r1 = $urandom;        $display("R1 = 
0x% 8h", r1);      #1;
    end
    p_randstate = p1.get_randstate(); // Збе-
реження стану в змінній p_randstate
    fork
       gen_f(1);
       gen_f(2);
    join
     p1.set_randstate(p_randstate); // віднов-
лення стану процесу з p_randstate
     repeat(4)begin
      r1 = $urandom;       $display("R1 = 0x%
8h", r1);      #1;
    end
  end
```

2. Відсутність взаємозалежності між методами рандомізації в SystemVerilog

У SystemVerilog використовуються два підходи до рандомізації: заснований на розподілі ймовірностей та обмеженнях. Вони застосовують різні механізми керування початковим станом генератора випадкових чисел (Seed), що унеможливлює їх взаємний вплив. Це проілюстровано в прикладі з лістингу 6, а), який містить два процеси, реалізовані за допомогою оператора initial: у першому використовується функція $random, у другому – $urandom.

Згідно з результатами моделювання (таблиця 3, а)), послідовність r1, згенерована функцією $random, повністю збігається з послідовністю прикладу 1, а), тоді як послідовність r2, згенерована $urandom, відповідає даним другого процесу прикладу 2, б). Це пояснюється тим, що кожен процес при створенні отримує власний початковий стан RNG незалежно від використовуваного методу рандомізації. Початкові стани RNG для даного прикладу наведено в таблиці 3.

При зміні порядку створення процесів (листинг 6, б)) значення r1, що генеруються функцією $random, залишаються незмінними (таблиця 3, б)), тоді як процес із функцією $urandom, створений першим, отримує інший початковий стан RNG. У результаті послідовність r2 збігається з даними першого процесу прикладів 2, а) і 2, б). Це підтверджує відсутність взаємозалежності між методами рандомізації та їх незалежність від порядку створення процесів.

Лістинг 6. Спільне використання $random та $urandom

а)

```
module random_ex();
   int r1, r2;process p1, p2;
   initial begin : P1
    p1 = process::self();    $display("Process 1: rand_state =  %s", p1.get_randstate());
    repeat(6) begin
        r1 = $random;    $display("R1 = 0x%8h", r1);    #3;
    end
  end
  initial begin : P2
    p2 = process::self();    $display("Process 2: rand_state =  %s", p2.get_randstate());
    repeat(6)    begin
        r2 = $urandom;       $display("R2 = 0x%8h", r2);  #1;
    end
  end
endmodule
```

б)

```
module random_ex();
   int r1, r2;process p1, p2;
initial begin :P2
      p2 = process::self();
    $display("Process 2: rand_state =  %s", p2.get_randstate());
      repeat(6)  begin
          r2 = $urandom;   $display("R2 = 0x%8h", r2);   #1;
    end
  end
  initial begin :P1
    p1 = process::self();
    $display("Process 1: rand_state =  %s", p1.get_randstate());
    repeat(6)   begin
      r1 = $random;    $display("R1 = 0x%8h", r1);    #3;
    end
  end
endmodule
```

Генеровані послідовності

а)

$random RNG=209583384	urandom RNG=1624058817
R1 = 0x12153524	R2 = 0x1b453960
R1 = 0xc0895e81	R2 = 0xd7e56dec
R1 = 0x8484d609	R2 = 0x5886eeab
R1 = 0xb1f05663	R2 = 0x68473a84
R1 = 0x06b97b0d	R2 = 0x1b0d23c4
R1 = 0x46df998d	R2 = 0x1fd6c40f

б)

$random RNG=1624058817	urandom RNG=209583384
R1 = 0x12153524	R2 = 0x63d5b317
R1 = 0xc0895e81	R2 = 0x440c06ab
R1 = 0x8484d609	R2 = 0x4be7c468
R1 = 0xb1f05663	R2 = 0x61df6913
R1 = 0x06b97b0d	R2 = 0x016d515e
R1 = 0x46df998d	R2 = 0xa12acbb7

Наведений приклад підтверджує, що генерування випадкових чисел за допомогою RNG залежить від порядку створення процесів, у яких він використовується. Водночас порядок та ієрархія процесів не впливають на результати функцій, заснованих на розподілі ймовірностей. Обидві групи операторів рандомізації функціонують незалежно одна від одної.

3. Методи формування випадкових тестів, стабільних до змін коду

Проведений аналіз особливостей роботи системних функцій і методів генерації випадкових значень у SystemVerilog дозволяє сформулювати ключові підходи до їх застосування, спрямовані на забезпечення стійкості та відтворюваності послідовностей випадкових даних.

1) Слід уникати використання функції $random одночасно в кількох процесах, оскільки такі виклики взаємно впливають один на одного. Для підвищення стійкості рекомендується використовувати $srandom з індивідуальними значеннями seed для кожного процесу або застосовувати методи, що базуються на генераторах випадкових чисел (RNG), зокрема $urandom.

2) При використанні методів на основі RNG для збереження стабільності випадкових послідовностей у вже розробленому коді рекомендується додавати нові процеси на тому самому рівні ієрархії лише після наявних.

3) При створенні ієрархічних процесів слід враховувати, що це змінює послідовність значень, генерованих батьківським процесом після їх додавання. Для усунення цього ефекту доцільно використовувати методи get_randstate() та set_randstate() для збереження і відновлення стану RNG батьківського процесу, як показано в лістингу 5. Це усуває вплив ієрархічних процесів на послідовність даних, що генерується батьківським процесом. Наприклад:

```
p_randstate = p1.get_randstate(); // Збереження стану RNG
```

```
fork
  gen_f(1);
  gen_f(2);
join
p1.set_randstate(p_randstate); // Відновлення
стану RNG
```

4) Для мінімізації впливу вкладених підпроцесів на генерацію випадкових даних доцільно використовувати збереження та подальше відновлення стану RNG.

Висновки. Дане дослідження було спрямоване на теоретичний аналіз, формування методів та практичних рекомендацій, пропонуючи комплексний підхід до вирішення проблеми стійкості псевдовипадкових тестів у SystemVerilog. Проведене теоретичне та експериментальне дослідження продемонструвало, що стабільність рандомізації визначається не лише вибором початкового стану генератора випадкових чисел, а й структурою тестового коду, зокрема порядком створення та ієрархією процесів.

Встановлено, що функції з розподілом ймовірностей ($random, $dist_) і методи з обмеженнями ($urandom, std::randomize()) відрізняються механізмами ініціалізації RNG та принципами формування послідовностей, що зумовлює необхідність роздільного підходу до їх використання. Показано, що додавання нових процесів на одному рівні ієрархії після існуючих дозволяє зберегти стабільність раніше розробленого коду, тоді як створення вкладених процесів може змінювати стан RNG батьківського процесу та впливати на подальшу генерацію значень.

Запропоновано практичну реалізацію методів, зокрема збереження та відновлення стану генератора і керування seed у кожному процесі, що дозволяє суттєво знизити вплив архітектурних змін тестового оточення на результати моделювання. Застосування цих підходів забезпечує відтворюваність тестів, критично важливу для локалізації та усунення помилок, спрощує налагодження та гарантує коректну узгодженість результатів при модифікаціях проєкту.

У роботі розглянуто приклади з використанням методів $random і $urandom, однак наведені особливості можуть бути узагальнені на інші системні функції кожної з груп. Процеси реалізовано за допомогою процедурних операторів initial та функцій; моделювання виконувалося у симуляторі Cadence Xcelium 23.09 з використанням стандартного значення seed, рівного 1.

Отримані результати мають прикладну цінність і можуть бути безпосередньо застосовані під час розробки тестбенчів на основі UVM та верифікації цифрових систем у промислових симуляторах. Таким чином, робота формує методологічну основу для забезпечення стійкості рандомізації у складних верифікаційних проєктах і сприяє підвищенню надійності та відтворюваності функціональних тестів.

Список літератури:

1. IEEE Std 1800-2017. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. New York : IEEE, 2017. 1315 p. ISBN 978-1-5044-4509-2.

2. IEEE Std 1800-2023. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language : Revision of IEEE Std 1800-2017. New York : IEEE, 2023. DOI: 10.1109/IEEESTD.2023.10308596.

3. Bergeron J. Writing Testbenches Using SystemVerilog. New York : Springer Science+Business Media, 2006. 431 p. ISBN 0-387-31275-7.

4. Thomas D., Moorby P. The Verilog Hardware Description Language. New York : Springer, 2019. ISBN 978-0-387-84930-0.

5. Smith D. Random Stability in SystemVerilog. *Proceedings of the SNUG (Synopsys Users Group) Conference*. 2013. Synopsys. URL: https://www.doulos.com/media/1293/snug2013_sv_random_stability_paper.pdf (дата звернення: 17.10.2025).

6. Efody A. UVM Random Stability: Don't leave it to chance. *Proceedings of DVCon*. 2012. URL: https://dvcon-proceedings.org/document/uvm-random-stability/ (дата звернення: 15.10.2025).

Hahanova I.V., Hahanov I.V., Filippenko I.V., Filippenko O.I. REPEATABILITY AND STABILITY OF CONSTRAINED RANDOM TESTS IN SYSTEMVERILOG

Testing of digital systems during the process of their functional verification is critically necessary, as it guarantees the correctness of operation of complex designs. One of the key approaches is the method of pseudorandom test generation. A critical need arises to ensure the stability and full reproducibility of such pseudorandom tests with a fixed Seed value, regardless of parallel execution of processes, the hierarchical structure of the model, or possible changes in the TestBench architecture. Even minor code changes can disrupt

the stability of SystemVerilog pseudorandom sequences, making tests non-reproducible and complicating debugging. This work is devoted to the study of the features of the operation of SystemVerilog language constructs with the aim of ensuring the stability and reproducibility of pseudorandom tests with a fixed Seed, regardless of process parallelism, model hierarchy, and changes in the TestBench structure. The main emphasis of the study is placed on a detailed examination of the influence of code structure, in particular the order and hierarchy of processes, on the reproducibility and stability of the generated sequences. The conducted analysis made it possible to formulate the features of the operation of system functions and methods for generating random values in SystemVerilog and to identify key approaches to their application that contribute to ensuring the stability and reproducibility of random data sequences. The study was aimed at theoretical analysis, the formation of methods, and practical recommendations, offering a comprehensive approach to solving the problem of stability of pseudorandom tests in SystemVerilog. The conducted theoretical and experimental research demonstrated that the stability of sequences generated by randomization mechanisms is determined not only by the choice of the initial state of the pseudorandom number generator, but also by the structural features of the test code, including the order and hierarchy of processes.

Keywords: *constrained random tests, seed, SystemVerilog, methods of random test generation, randomization, FPGA.*

Дата першого надходження статті до видання: 10.01.2026
Дата прийняття статті до друку після рецензування: 06.02.2026
Дата публікації (оприлюднення) статті: 08.04.2026