Создание псевдослучайных значений (Randomization)

Тест может быть создан с помощью генератора псевдослучайных значений (constrained-random tests (CRT)). Прямой тест (directed test) находит запланированные ошибки, в то время как CRT-тест обнаруживает случайные (незапланированные) ошибки:

13.1. Простейший класс включающий псевдослучайные переменные.

Листингом 13.1 представлен простейший класс Packet, формирующий псевдослучайные последовательности для четырех переменных: src, dst, data, kind.

Модификатор **rand** означает, что данные могут формироваться, получая новое значение или повторяя какое-либо из предыдущих. При применении модификатора **randc** значение может появиться повторно, только после того как будут перебраны все возможные комбинации.

Листинг 13.1. Пример класса с псевдослучайными значениями

```
class Packet;
    // Случайные переменные
    rand bit [31:0] src, dst, data[8];
    randc bit [7:0] kind;
    // Ограничение значений src
    constraint c {src > 10; src < 15;}
endclass

Packet p;
initial begin
    p = new; // Создание пакета
    assert(p.randomize());
    transmit(p); // Какая-то пользовательская функция, которая передает значения
дальше
end
```

Управляющие генерацией константы записываются в конструкцию constraint и группируется с помощью фигурных скобок { }. Такие скобки используются, потому что в них заключается декларативный код, а не процедурный, где необходимо применять begin...end. Если в процессе генерации значения возникнет ошибка, то функция randomize возвращает 0, что отслеживается с помощью прямой ассерции.

Класс Stim (Листинг 13.2) содержит класс с псевдослучайной генерацией значений и несколькими блоками констант. Первый блок c_stim задает диапазон возможных значения для len. Множество значений может быть создано также с помощью оператора inside, как для переменной dst. При этом задается условие, что данное ограничение будет действовать, если бит congestion_test равен 1. SystemVerilog создает множество возможных значений и выбирает из него значения с равной вероятностью, если нет других условий ограничения переменной.

Листинг 13.2. Псевдослучайный класс с ограничениями

```
class Stim;
    const bit [31:0] SRC_ADDR = 42;
    typedef enum {READ, WRITE, CONTROL} stim_t;
    randc stim_t type;                    // Переменная типа перечисления
    rand bit [31:0] len, src, dst;
    bit congestion_test;
    constraint c_stim {
            len < 1000;
            len > 0;
```

```
    src inside {0, [2:10], [100:107]};
    if (congestion_test) {
            dst inside {[SRC_ADDR - 100 : SRC_ADDR + 100]};
            }
    }
endclass
```

Пример правильного и неправильного описания констант с помощью операторов отношения представлен листингом 13.3. Как видно из примера, переменная может появляться в нескольких условиях, при этом допускается в выражении использовать только один оператор отношения (<, <=, ==, >= или >).

Листинг 13.3 Переменные ограничения описаны в определенном порядке

```
class bad;
    rand bit [15:0] a, b, c;
    constraint good {0 < a;   // Правильное описание
            a < b;
            b < c;}
    constraint bad {0 < a < b < c;} // Неправильно, приведет к ошибке
endclass
```

Диапазон возможных значений можно задавать с помощью переменных, как это показано в примере с листинга 13.4, где переменные lo и hi определяют нижнюю и верхнюю границу множества.

Листинг 13.4 Псевдослучайное множество значений

```
rand int c;                     // Псевдослучайная переменная
int lo, hi;                     // Обычная переменная, используемая для описания границ
constraint c_range {
            c inside {[lo:hi]};         // lo <= c и c <= hi
    }
```

Символ инверсии (!) указывает, что допустимыми являются значения, не входящие в указанное множество.

```
constraint c_range {
            !(c inside {[lo:hi]});      // c < lo или c > hi
    }
```

Все значения множества будут иметь одинаковую вероятность выбора, даже если они появляются несколько раз.

```
constraint c_even_weight {
            (c inside {0,1,1,1,1,1});   // 0 и 1 имеют одинаковую вероятность
    }
```

Изменить вероятность появления значений можно оператором dist.

В следующем примере диапазон значений для переменной x определяется случайногенерируемой переменной y.

```
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2*y], z};}
```

13.2. Использование массивов в качестве границ множеств значений

Допускается выполнять выбор значения из набора, представленного массивом.

```
integer fives[4] = '{ 5, 10, 15, 20 };
rand integer v; constraint c3 { v inside {fives}; }
```

В примере (листинг 13.5) таким образом осуществляется выбор дня недели DAYS_E из массива choices, имеющего тип enum. Список choices может быть легко изменен перед генерированием значения. В случае использования оператора randc, система генерирования переберет все возможные значения перед повтором уже присвоенных.

Листинг 13.5 Выбор из множества возможных значений

```
class Days;
    typedef enum {SUN, MON, TUE, WED,
            THU, FRI, SAT} DAYS_E;
    DAYS_E choices[$];
    rand DAYS_E choice;
    constraint cday {choice inside {choices};}
endclass

Days days;
initial begin
    days = new;
    days.choices = '{Days::SUN, Days::SAT};
    assert (days.randomize());
    $display("Random weekend day %s\n", days.choice.name);
    days.choices = '{Days::MON, Days::TUE, Days::WED, Days::THU, Days::FRI};
    assert (days.randomize());
    $display("Random week day %s", days.choice.name);
end
```

Если необходимо добавить или удалить данные из динамического массива, следует осторожно использовать оператор inside, из-за его особенностей. Например, существует множество значений, заданное очередью, из которых должно быть выбрано и удалено только одно. Это потребует создать N ограничений, где N–количество элементов, остающихся в очереди. В этом случае для массива выбора предпочтительней использовать randc (Листинг 13.6). Выбор значения с модификатором randc занимает значительно меньше времени, в отличие от создания большого числа ограничений, особенно, если констант больше 10.

Листинг 13.6 Использование randc для выбора значений массива в случайном порядке

```
class RandcInside;
    int array[];                    // Значения для выбора
    randc bit [15:0] index;    // Индексация массива
    function new(input int a[]);        // Конструктор и инициализация
            array = a;
    endfunction
    function int pick;                  // Возвращение нового результата выбора
            return array[index];
    endfunction
    constraint c_size {index < array.size;}
endclass

initial begin
    RandcInside ri;
    ri = new('{1,3,5,7,9,11,13});
    repeat (ri.array.size) begin
            assert(ri.randomize());
            $display("Picked %2d [%0d]", ri.pick(), ri.index);
    end
end
```

13.3. Взвешенное распределение

Оператор dist позволяет выполнять взвешенное случайное генерирование значений таким образом, что некоторые значения будут появляться чаще, чем другие. Оператор dist (листинг 13.7) принимает список значений и весов, разделенных символами ":=" или ":/". Значения и веса могут быть константами или переменными. Значение может быть задано числом или диапазоном вида [lo:hi]. Оператор := означает одинаковый вес для каждого описанного значения диапазона, в то время как :/ означает, что вес должен быть поделен всеми значениями диапазона. Вес является процентами и не может быть в сумме больше 100.

В примере (листинг 13.7), **src** получает значения 0, 1, 2 или 3. Вес 0 равен 40, а 1, 2 и 3 - по 60, всего получается 220. Тогда вероятность выбора 0 равна 40/220, а вероятности 1, 2 или 3 - по 60/220 каждая. Далее **dst** получает значения 0, 1, 2 или 3. Вес 0 равен 40, а 1, 2 и 3 разделили общий вес 60, полный вес равен 100. В этом случае вероятность выбора нуля равна 40/100, а вероятность выбора 1, 2 или 3 по 20/100 каждая.

Листинг 13.7 Взвешенное псевдослучайное генерирование значений

```
rand int src, dst;
constraint c_dist {
src dist {0:=40, [1:3]:=60};
    // src = 0, weight = 40/220
    // src = 1, weight = 60/220
    // src = 2, weight = 60/220
    // src = 3, weight = 60/220
dst dist {0:/40, [1:3]:/60};
    // dst = 0, weight = 40/100
    // dst = 1, weight = 20/100
    // dst = 2, weight = 20/100
    // dst = 3, weight = 20/100
}
```

Использование переменных для весов выбора позволяет динамически менять распределение случайных переменных, как это представлено в листинге 13.8.

Листинг 13.8 Динамическое изменение распределения весов

```
// Операторы шины длиной в байт, слово или длинное слово
class BusOp;
        // Длина операндов
    typedef enum {BYTE, WORD, LWRD } length_t;
    rand length_t len;
        // Псевдослучайные веса для ограничений dist
    bit [31:0] w_byte=1, w_word=3, w_lwrd=5;
    constraint c_len {
        len dist {BYTE := w_byte,      // Выбор псевдослучайного значения
        WORD := w_word,                // Использование длины
        LWRD := w_lwrd};               // Переменный вес
    }
endclass
```

13.4. Двунаправленные ограничения

В примере (листинг 13.9) SystemVerilog рассматривает все четыре ограничения одновременно: b меньше d, которое в свою очередь меньше 30. Но b имеет ограничение на равенство с переменной c, которая должно быть больше 25. Даже если здесь нет прямого

ограничения на минимальное значение d, оно осуществляется ограничением для переменной c (таблица 13.1).

Листинг 13.9 Двунаправленные ограничения

```
rand logic [15:0] b, c, d;
constraint c_bidir {
        b < d;
        c == b;
        d < 30;
        c > 25;
}
```

Таблица 13.1 Решения для двунаправленных ограничений

Решение	b	c	d
1	26	26	27
2	27	27	28
3	28	28	29

13.5. Условные ограничения

Обычно ограничения активны все время. В ситуациях, когда необходимо отключить действие ограничений на некоторое время SystemVerilog предлагает две формы условных операторов: импликации “->” и “if-else”.

```
// Блок ограничений с оператором импликации
class BusOp;
    ...
    constraint c_io {
        (io_space_mode) -> addr[31] == 1’b1;
        }
```

Если присутствуют выражения для обеих ветвей условия true-false, то предпочтительней использовать оператор if-else.

```
// Блок ограничений с оператором if-else
class BusOp;
    ...
    constraint c_len_rw {
        if (op == READ)
            len inside {[BYTE:LWRD]};
        else
            len == LWRD;
        }
```

В блоках ограничения, поскольку это декларативный код, используются фигурные скобки {}. В процедурном коде применяется “begin...end”.

13.6. Использование арифметических операторов

Для ограничений могут быть применены простые арифметические операторы: сложение, вычитание, выделение битов, сдвиг. По умолчанию ограничения имеют размер 32 бита.

Также может быть использован оператор “solve...before» (Листинг 13.10). Конструкция “solve...before” не влияет на пространство выбора решений, а только на вероятность. Выбор значений x выполняется с равной вероятностью для (0, 1). Из 1000 случайных значений x, 500 будет равно 0, а 500 – 1.Когда x равно 0, y должен быть равен 0. Когда x равен 1, то y может быть 0, 1, 2 или 3 с равной вероятностью (таблица 13.2).

Листинг 13.10 Класс с импликацией и solve...before

```
class SolveBefore;
   rand bit x; // 0 or 1
   rand bit [1:0] y; // 0, 1, 2, or 3
   constraint c_xy {
         (x==0) -> y==0;
         solve x before y;
         }
endclass
```

Таблица 13.2 Возможные значения x и y для ограничений solve x before y

Решение	x	y	Вероятность
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/9
H	1	3	1/8

Но в случае использования ограничения “solve y before x”, будет получено другое распределение (таблица 13.3).

Таблица 13.2 Возможные значения x и y для ограничений solve y before x

Решение	x	y	Вероятность
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4

13.7. Управление несколькими блоками ограничений

Класс может содержать несколько блоков ограничений. Например, одна группа управляет разрядностью данных транзакции, а другая – длиной транзакции. Тогда для того, чтобы включать или отключать ограничения, можно применять метод constraint_mode(). В формате <имя ограничения>.constraint_mode(), этот метод управляет единственным ограничением.

Листинг 13.11 Использование constraint_mode

```
class Packet;
   rand int length;
   constraint c_short {length inside {[1:32]}; }
   constraint c_long {length inside {[1000:1023]}; }
endclass
```

```
Packet p;
initial begin
    p = new;
    // Создание длинных данных long, отключаа ограниничение для
коротких short
    p.c_short.constraint_mode(0);
    assert (p.randomize());
    transmit(p);
    // Создание коротких данных, через отключение всех ограничени
    // а затем разрешается использования ограничений для коротких
данных
    p.constraint_mode(0);
    p.c_short.constraint_mode(1);
    assert (p.randomize());
transmit(p);
end
```

13.8. Использование ограничений для управления правильностью данных

Ограничения могут быть использованы для управления формированием только допустимых для модели значений. Например, в следующем примере операция RMW (Read-Modify-Write) разрешена только для длинных данных LWRD.

```
// Проверка длины переменной с использованием ограничения valid
class BusTrans;
    rand enum {BYTE, WORD, LWRD, QWRD} length;
    rand enum {READ, WRITE, RMW, INTR} opc;
    constraint valid_RMW_LWRD {
                    (opc == RMW) -> length == LWRD;
            }
endclass
```

13.8.1. Линейные ограничения

Дополнительное управление ограничениями в момент генерации значений реализуется с помощью конструкции randomize() with (листинг 13.12).

Листинг 13.12 Оператор randomize() with

```
class Transaction;
    rand bit [31:0] addr, data;
    constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t = new();
initial begin
    int s;
    t = new();
    // addr может принимать значения 50-100, 1000-1500, data < 10
    assert(t.randomize() with {addr >= 50; addr <= 1500; data < 10;});
    driveBus(t);
    // Формирование определенного значения для  addr, data > 10
    assert(t.randomize() with {addr == 2000; data > 10;});
    driveBus(t);
end
```

Для отключения конфликтующих ограничений можно использовать функцию constraint_mode.

13.9. Функции pre_randomize и post_randomize

SystemVerilog имеет функции для нелинейного распределения случайно генерируемых данных. Это void-функции, которые не возвращают значения, но в отличие от задач, в них не допускается использование параметров времени. С их помощью можно выполнять действия сразу до или после запуска функции формирования псевдослучайных значений.

Пример использования функции **pre_randomize** представлен листингом 13.13, формируемые ими значения имеют распределение указанное на рисунке 13.1.

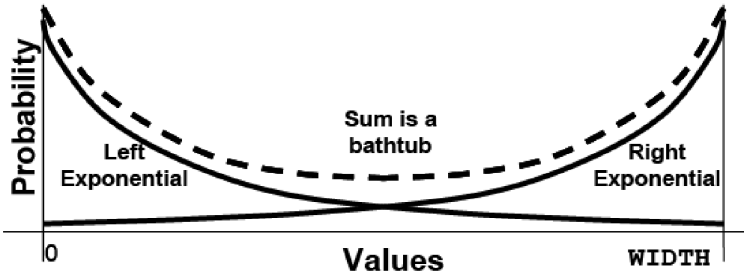

Рис. 13.1 Встроенное U-образное распределение

Листинг 13.13 Встроенное U-образное распределение

```
class Bathtub;
   int value;        // Псевдослучайная переменная с U-образным
распределением
   int WIDTH = 50, DEPTH=4, seed=1;
   function void pre_randomize();
            // Вычисление левой части кривой
            value = $dist_exponential(seed, DEPTH);
            if (value > WIDTH) value = WIDTH;
            // Псевдослучайная расстановка точек в левой или правой части
кривой
            if ($urandom_range(1))
                  value = WIDTH - value;
   endfunction
endclass
```

Для управления генерированием значений могут быть использованы следующие системные задачи

- $dist_exponential — Экспоненциальное затухание (см. рис. 13.1)
- $dist_normal — Нормальное распределение (гауссово распределение)
- $dist_poisson — Распределение Пуассона
- $dist_uniform — Равномерное распределение
- $random — Равномерное распределение, возвращает знаковое 32-битовое значение
- $urandom — Равномерное распределение, возвращает беззнаковое 32-битовое значение
- $urandom_range — Равномерное распределение с указанием диапазона

13.10. Настраиваемые тесты со случайной генерацией

Для указания границ псевдослучайной генерации могут быть использованы переменные.

```
class bounds;
    rand int size;
    int max_size = 100;
    constraint c_size {
            size inside {[1:max_size]};
            }
endclass
```

По умолчанию, класс присваивает случайной переменной size значения от 1 до 100, но меняя величину переменной max_size, можно изменять верхнюю границу диапазона.

Можно использовать переменную с ограничениями dist для того чтобы включать и отключать значения и диапазоны. В примере (Листинг 13.14), каждая команда шины имеет различные весовые переменные.

Листинг 13.14 Ограничения dist с весами переменной

```
typedef enum (READ8, READ16, READ32) read_t;
class ReadCommands;
    rand read_t read_cmd;
    int read8_wt=1, read16_wt=1, read32_wt=1;
    constraint c_read {
            read_cmd dist {READ8 := read8_wt,
            READ16 := read16_wt,
            READ32 := read32_wt};
                    }
endclass
```

13.11. Использование неслучайных значений

Оператор rand_mode позволяет отключить псевдослучайную генерацию для заданной переменной, как это сделано в примере с листинга 13.15 для поля length, определяющая длину массива payload.

Листинг 13.15 Отключение псевдослучайной генерации значений с помощью `rand_mode`

```
// Класс Packet с переменными length и payload
class Packet;
    rand bit [7:0] length;
    rand bit [7:0] payload[];
    constraint c_valid {length > 0; payload.size == length;}
    function void display(string msg);
            $write("Packet len=%0d, payload size=%0d, bytes = ", length,
payload.size);
            for(int i=0; (i<4 && i<payload.size); i++)
            $write(" %0d", payload[i]);
            $display;
    endfunction
endclass

Packet p;
initial begin
    p = new();
```

```
    // Псевдослучайное генерирование всех переменных
    assert (p.randomize());
    p.display("Simple randomize");
    // Отключение псевдослучайной генерации значений length, затем
генерирование значений для пакета
    p.length.rand_mode(0);
    p.length = 42;
    assert (p.randomize());
    p.display("Randomize with rand_mode");
end
```

Если после псевдослучайного генерирования значений объекта выполняется изменение значений некоторых переменных, то необходимо выполнить проверку правильности данных согласно ограничениям. Вызов handle.randomize(null) SystemVerilog рассматривает все переменные как неслучайные (переменные состояние) и гарантирует, что все ограничения, указанные константами, выполняются (листинг 13.16).

Листинг 13.16 Включение и отключение ограничений с помощью constraint_mode

```
class Instruction;
    rand OPCODE_T opcode;
    …
    constraint c_no_operands {
            opcode == NOP || opcode == HALT;}
    constraint c_one_operand {
            opcode == CLR || opcode == NOT;}
endclass
Instruction instr;
initial begin
    instr = new;
    // Генерирование instruction без операндов
    instr.constraint_mode(0);      // Отключение всех ограничений
    instr.c_no_operands.constraint_mode(1);
    assert (instr.randomize());
    // Генерирование instruction с одним операндом
    instr.constraint_mode(0);              // Отключение всех ограничений
    instr.c_one_operand.constraint_mode(1);
    assert (instr.randomize());
end
```

13.12. Использование внешних определений ограничений

Константы, управляющие генерированием значений, могут быть заданы за границами класса. В примере (Листинг 13.17) в классе Packet объявлены внутреннее ограничение c_valid и прототип внешнего ограничения c_external, которые реализовано в программе test.

Листинг 13.17 Класс с внешними ограничениями и их использование

```
// Определение класса
class Packet;
    rand bit [7:0] length;
    rand bit [7:0] payload[];
    constraint c_valid {length > 0;
```

```
            payload.size == length;}
    constraint c_external;
endclass

// Определение внешних ограничений в программе
program test;
    constraint Packet::c_external {length == 1;}
    ...
endprogram
```

13.13. Проблемы псевдослучайной генерации

В примере (листинг 13.18) ограничение total_len указывает, что сумма пары переменных (pkt1_len + pk2_len) не должна превышать 64. При этом корректными значениями будут (32, 32) и (2, 62). Однако этому условию удовлетворяет и пара значений (-64, 128). В таких ситуациях, чтобы избежать появления отрицательных значений, следует использовать беззнаковые типы данных (листинг 13.19).

Листинг 13.18 Знаковые переменные являются причиной проблем псевдослучайной генерации

```
class SignedVars;
    rand byte pkt1_len, pk2_len;
    constraint total_len {
            pkt1_len + pk2_len == 64;
    }
endclass
```

Листинг 13.19 Формирование значений для беззнаковых 32-битовых переменных

```
class Vars32;
    rand logic [31:0] pkt1_len, pk2_len;        // беззнаковый тип
    constraint total_len {
    pkt1_len + pk2_len == 64;
            }
endclass
```

Даже эта версия приводит к ошибкам. Большие значения pkt1_len и pkt2_len, как 32'h80000040 и 32'h80000000, при сложении дадут 32'd64. Следует подумать о сложении другой пары констант для ограничения значений этих двух переменных. Но лучший подход - сделать их меньшей длины и избежать использования 32-битовых переменных в ограничениях (листинг 13.19).

Листинг 13.19 Формирование значений для беззнаковых 8-битовых

```
class Vars8;
    rand logic [7:0] pkt1_len, pk2_len;                    // 8-битов
    constraint total_len {
                    pkt1_len + pk2_len == 8'd64;           // 8-битов
            }
endclass
```

13.14. Ограничения итеративные и массивов

Форму распределения значений сигналов можно менять с помощью оператора foreach и нескольких функций массивов. Простейшим ограничением является функция size.

Применяется для определения количества элементов в динамическом массиве или очереди.

```
class dyn_size;
   rand reg [31:0] d[];
   constraint d_size {d.size inside {[1:10]}; }
endclass
```

В примере использование ограничения inside позволяет задать нижнюю и верхнюю границы массива.

Примеры использования foreach для управления генерированием значений массивов. Ограничение A_size задает размер массива, ограничение A_values определяет возрастающий порядок значений в массиве.

```
class Transaction;
        rand byte A[] ;
        constraint A_size { A.size >1; A.size <=7; }
        constraint A_values {
                foreach ( A[ k ] ) (k < A.size - 1) -> A[k + 1] > A[k]; }
endclass : Transaction
```

Пример с листинга 13.20a представляет использование конструкции foreach сразу в двух ограничениях C1 и C2.

Листинг 13.20a - Использования foreach и управления выбором ограничений

```
class Transaction;
  rand byte A[] ;
 constraint C1 { A.size > 0; A.size < 10;
    foreach ( A [ i ] ) A[i] inside {2,4,8,16}; }
 constraint C2 { A.size > 2; A.size < 20;
    foreach ( A [ j ] ) A[j] == 2 * j; }
endclass

Transaction tr = new;

initial begin
   $display(" RANDOMIZE START");
   repeat (3) begin // Run a few cycles
        tr.C2.constraint_mode(0);          // Отключить ограничение C2
        assert(tr.randomize);              // Генерация транзакции
        $display ("Array size %d", tr.A.size);
        foreach (tr.A[i])begin
                ifc.element <= tr.A[i]; // передача значения массива переменой ifc.element
                #1;
        end
    end
```

```
  tr.constraint_mode(0); // Отключить все ограничения
  $display(" Second constraint");
  repeat (3) begin // Run a few cycles
        tr.C2.constraint_mode(1); // Включить ограничение C2
        assert(tr.randomize);
        $display ("Array size %d", tr.A.size);
        foreach (tr.A[i])begin
              ifc.element <= tr.A[i];
              #1;
        end
  end
  $display(" RANDOMIZE END");
  $finish;
end
```

13.14.1. Сумма элементов.

Случайный массив данных можно использовать его для контролирования потока передаваемой информации. Например, интерфейс генерирует четыре слова данных, которые могут быть посланы последовательно, через такт или несколько тактов. Сигнал strobe обозначает наличие данных на линии. Существует несколько вариантов стробирования для передачи значений в течение десяти циклов (рис 13.2). Реализовать их можно с помощью случайных битовых массивов (листинг 13.20). В константе ограничения c_set_four используется функция суммирования sum, которая указывает, что в массиве может быть только 4 единицы, что соответствует 4 стробам передаваемых данных .

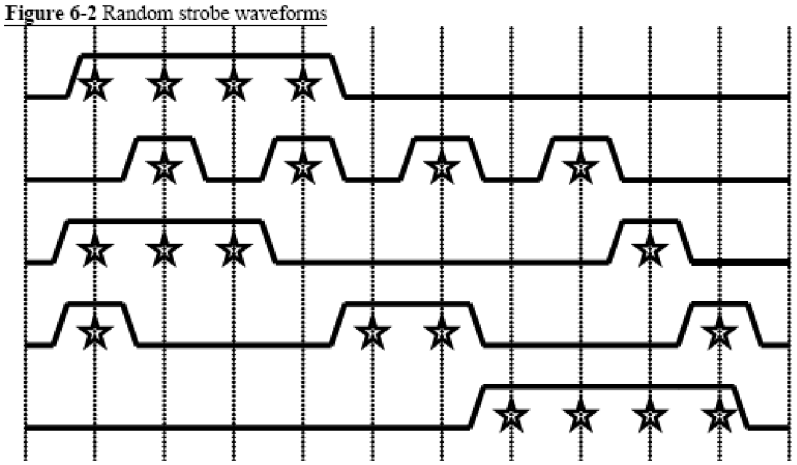

Рис. 13.2 Случайная форма сигналов стробирования

Листинг 13.20 Использование шаблона стробирования

```
// Класс для формирования псевдослучайного шаблона стробирования
parameter MAX_TRANSFER_LEN = 10;
class StrobePat;
   rand bit strobe[MAX_TRANSFER_LEN];
   constraint c_set_four { strobe.sum == 3'h4; }
endclass
```

```
// Использование класса шаблона стробирования
initial begin
    StrobePat sp;
    int count = 0;                  // Индекс в массиве данных
    sp = new();
    assert (sp.randomize);
    foreach (sp.strobe[i]) begin
            bus.cb.strobe = sp.strobe[i];
            // If strobe is enabled, drive out next data word
            if (sp.strobe[i])
                    bus.cb.data = data[count++];
    end
end
```

13.15 Контрольные вопросы и задания

1. Где применяется псевдослучайная генерация значений переменных?
2.