Gerador de Nmeros Aleatrios Weaver

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article contains the implementation of several random number generator algorithms in literary programming and is intended to be used with Weaver Game Engine. However, it can also be used as an independent API in other projects. All different algorithms are encapsulated in the same API letting programmers to easily change the algorithm using macro definitions. We also describe here how the algorithms perform in some statistical tests and in benchmarks running in Linux, OpenBSD, Windows and Web Assembly.*

1. Introduction

1.1. Random Number Generators in Games

A random number generator is useful in the following scenarios in a game:

a) Simulation: This applies to natural phenomenon that can be random but also applies to social phenomenon. This is commonly used in games that are simulators. But also can simulate changes in the weather, wind or the mood of characters in practically any kind of game.

b) Sampling: A game can have different number of opponents that could be generated. By the generation of different characteristics chosen randomly we are sampling one between all possible characters that could be generated. For example in a Pokmon game each species of the creatures can have lots of variation in different attributes depending of the genetics and also the attacks that the creature knows can change. Generating a random creature, we are sampling one individual between other possible ones. We could also have a game with a procedurally generated world, where the entire world is sampled from the set of all possible worlds.

c) Programming: During the game development, the programmer can use a random number generator to simulate different choices that a player could do just to test if he can find some combinations that bring problems. And also can generate random levels to test if some bug involving the game engine can be spotted.

d) Aesthetics: Adding some randomness in image patterns that otherwise would be regular images can increase the aesthetic appeal for them. Some randomness also can make screen transitions more interesting.

e) Entertainment: The fun of a game could be entirely in the randomness. This is the case for games involving dices, roulettes, shuffling cards.

With so many different use cases for random number generators in games, we really need a general purpose random number generator. Never a generatos specialized only in some of the abose use cases.

Usually we have a greater tollerance when a random number generator produces predictable results in a game. Lots of old games used in the past generators that consisted in a sequence of fixed numbers, and the bias in such games was not apparent. But there are cases when the consequence of a bad generator are very severe. For example, a simulator that aims for realism can produce a bad training for players that use the game to trains for real

activities. A bad generator can be exploited in games involving bets or serious competition.

Our objective here will be define a random number generator API and different options for random number generators to instantiate it. A user can choose which algorithm will be used, but if no choice is presented, the defined library will make its own choice about the algorithm.

1.2. Literary Programming and Notation

This article uses the technique of "Literary Programming" to develop our random number generator API. This technique was presented at [Knuth, 1984] and have as objective develop software in a way that a computer program to be compiled is exactly equal a document written for human beings detailing and explaining the code. This document is not independent of the source code, it is the project source code. Automated tools are used to extract the code from this document, sort it in the right order and produce the code that is passed to a compiler.

For example, in this article we will define two different files: `random.c` and `random.h`. Both of them can be inserted statically in any project or compiled as a shared library. The content of `random.h` is:

File: `src/random.h`**:**

```
#ifndef WEAVER_RANDOM
#define WEAVER_RANDOM
#ifdef __cplusplus
extern "C" {
#endif
#include <stdint.h>
#include <stdbool.h>
#if defined(__unix__) || defined(__APPLE__)
#include <pthread.h>
#endif
                <Section to be inserted: Choose Default RNG Algorithm>
                    <Section to be inserted: RNG Structs>
                <Section to be inserted: RNG Declarations>
#ifdef __cplusplus
}
#endif
#endif
```

The two first lines and the last one are macros that ensure that the function declaration and variables from this file will always be included at most once in a compiling unit. We also put macros to check if we are compiling this as C or C++. If we are in C++, we assure the compiler that all our functions will be in C-style. We never will modify them with operator overload, for example. This makes the code became more compact.

The red part in the above code shows that some code will be inserted there in the future. In "RNG Declarations", for example, we will put there function declarations.

Each piece of code have a title, that in the above example is `random.h`. The title shows where the code will be placed. In the case above, the code will directly to a file. In future pieces of code, we will have a title "RNG Declarations" with the code that will be inserted in the place of the red text in the code above.

1.3. API functions that will be defined

Our RNG API will have three different functions.

The first one generates and initializes a random number generator returning a poiter

for it. To initialize it, we will require a memory allocation function (can be `malloc` from standard library, or a custom memory allocator). Next we need a seed represented as an array size and an array of 64-bit integers that we will assume being chosen uniformly at random.

This way, our initialization function is generic enough that the user can use a personalized ammount of entropy for the generator. In a shuffling card game, for example, if we want to shuffle 54 cards, we would want in the initialization a number of n random bits such that 2^n is not much smaller than $54!$. Othewise, we surely will not be able to generate most of the possible shuffling. However, in some machines and environment, getting randomness could be difficult and we should be able to pass a smaller number of random bits during initialization.

Section: RNG Declarations:

```
struct _Wrng *_Wcreate_rng(void *(*alloc)(size_t), size_t size, uint64_t *seed);
```

However, some algorithms can ignore additional randomness passed if the number is greater than recommended. And some algorithms can give no guarantee of quality if you pass less randomness that recommended. The API should inform the programmer the minimum recommended size and maximum recommended size for the array in the initialization. We will define these values during the description of different algorithms:

Section: RNG Declarations:

<Section to be inserted: Warn About Ideal Seed Size>

The second function in our API is the one that returns a new 64-bit random number after each invocation:

Section: RNG Declarations (continuation):

```
uint64_t _Wrand(struct _Wrng *);
```

And the last function will finalize the random number generator. It receives a pointer for the disallocation function or NULL. If the function gets NULL as first argument, the random number generator will not be disallocated (some memory managers could use some kind of garbage collection instead of a `free` function). But even if not disallocated, the random number generator should not be used after being finalized with this function:

Section: RNG Declarations (continuation):

```
bool _Wdestroy_rng(void (*free)(void *), struct _Wrng *);
```

1.4. Suporte Threads

The code in this section will ensure that more than one thread can use our random number generator after its initialization. First we need the headers for thread support where this is supported:

Section: Including Headers (continuation):

```
#if defined(__unix__) || defined(__APPLE__)
#include <pthread.h>
#endif
#if defined(_WIN32)
#include <windows.h>
#endif
```

Each random number struct will have a mutex (inside `struct _Wrng`):

Section: Mutex Declaration:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_t mutex;
#endif
```

```
#if defined(_WIN32)
CRITICAL_SECTION mutex;
#endif
```

If we have a pointer for `struct _Wrng` called `rng`, we can initialize the mutex with:

Section: Initializing Mutex:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_init(&(rng -> mutex), NULL);
#endif
#if defined(_WIN32)
InitializeCriticalSection(&(rng -> mutex));
#endif
```

Before entering the critical section in code with this mutex, we do the following:

Section: Mutex:WAIT:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_lock(&(rng -> mutex));
#endif
#if defined(_WIN32)
EnterCriticalSection(&(rng -> mutex));
#endif
```

And after the critical section, we release the mutex with:

Section: Mutex:SIGNAL:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_unlock(&(rng -> mutex));
#endif
#if defined(_WIN32)
LeaveCriticalSection(&(rng -> mutex));
#endif
```

The main reason for a function to finalize the generator, in our case, the function `_Wdestroy_rng`, is the need to finalize the mutex. And free allocated memory if needed. As there is no other reason, here we already know the necessary to describe the generator finalization. And we can define the finalization function:

Section: Definition: _Wdestroy_rng:

```
bool _Wdestroy_rng(void (*free)(void *), struct _Wrng *rng){
  bool ret;
#if defined(__unix__) || defined(__APPLE__)
  ret = pthread_mutex_destroy(&(rng -> mutex));
#elif defined(_WIN32)
  DeleteCriticalSection(&(rng -> mutex));
  ret = true; // According with the documentation, this always succeeds
#endif
  if(free != NULL)
    free(rng);
  return ret;
}
```

2. Random Number Generator Algorithms

2.1. Linear Congruent Generator (LGC)

A Linear Congruent Generator (LCG) is a generator of sequences $(x_0, x_1, \ldots))$ such that for given integer values a, c and m:

$$x_{i+1} = ax_i + c \ (mod\ m)$$

The choice for the parameters a, c and m determines the quality of the generator. As we want a generator for 64-bit numbers, we will choose $m = 2^{64}$. This way, we do not need to use explicitly the modulo operation.

About the multiplier a, we will use the value 0xfa346cbfd5890825 because this is one of the values suggested in the article [Steele, 2021] that examined rhe quality of different parameters for this kind of generator.

About the value c, we need to ensure that it will be an odd integer (because it need to be prime in relation to m). Other than this, we can choose any possible odd number. The initial value also need to be an odd integer.

To ensure this, we can use the first 64 bits of our seed to initialize x_0 after force the least significative bit to 1. If we have more 64-bit numbers in the seed, we can use it as our c, also setting to 1 the least significative bit. Otherwise, we just use $c = 1$.

Ths means that the recommended size for our seed is 1 or 2 numbers with 64 bits:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_LCG
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  2
#endif
```

The struct for our generator is very simple. We need to store only the last generated value and the constant c:

Section: RNG Structs:

```
#ifdef W_RNG_LCG
struct _Wrng{
  uint64_t last_value, c;
                    <Section to be inserted: Mutex Declaration>
};
#endif
```

And we initialize this with the following code:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_LCG
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    // If have a seed, use it, otherwise pick this constant chosen at random:
    rng -> last_value = ((size > 0)?(seed[0]):(0x1c3b9d10b1d41adc));
    rng -> c = ((size > 1)?(seed[1]):(1));
    rng -> last_value |= (1u);
    rng -> c |= (1u);
                      <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

And the function that returns new random values is:

Section: Definition: _Wrand:

```
#ifdef W_RNG_LCG
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                    <Section to be inserted: Mutex:WAIT>
  rng -> last_value = 0xfa346cbfd5890825 * rng -> last_value + rng -> c;
  ret = rng -> last_value;
                    <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
#endif
```

The expected period for this generator is 2^{64}. After this, the values will repeat themselves.

2.2. SFMT

The generator SFMT is the "SIMD-Oriented Fast Mersenne Twister", proposed first at [Saito, 2006]. Its name is because this generator period are very big mersenne primes (with the form $2^n - 1$). The most used implementation of this algorithm have periods of $2^{199937} - 1$ different values without repetitions. This is more than enough (and even an exaggeration) for any use case. And "SIMD" is an instruction set for vectorizing 128-bit integers that more recet CPUs support. These instructions are expected to make the algorithm run faster, if the compiler is smart enough to use it, or if we use them explicitly in assembly.

The generator works using the following recursion where each value x_i have 128 bits:

$$X_{i+n} = g(X_i, \ldots, X_{i+n-1})$$

In our case, we use $n = 156$. This means that we always must store and keep in the memory the last 156 values of the sequence, each one with 128 bits. This explains why this generator needs much more memory when compared to others.

The function g works in this case concatenating all the input values in a binary vector with 156×128 values. Then, it multiplies a binary matrix by this binary vector. The binary matrix has 156×128 columns and 128 lines. Except that instead of working with the usual field of rational or real numbers, we are working with the finite field GF(2) composed only by two numbers: 0 and 1 (the vector and the matrix are binaries). What we means by "multiplication" in this context is the logical AND and the addition is the logical XOR. And we choose a convenient matrix such that we do not need to store and multiply it explicitly. We can represent the multiplication of the matrix by a vector using a compact number of operations.

In the struct for our generator, we need to store all the previous neccessary values and also an offset, that stores the position of the next 64-bit value to be returned. If the offset is even, we assume that we neet to generate a new value. In the even case, we generate a new 128 bit value and return the first 64 bits. In the odd case, we still have the remaining 64 bits of the last value and do not need to generate a new one.

Section: RNG Structs:

```
#ifdef W_RNG_MERSENNE_TWISTER
struct _Wrng{
  char w[128 * 156 / 8]; // Todos N valores gerados, cada um com _W bits
  int offset;          // Index for the next 64-bit value to be returned
                    <Section to be inserted: Mutex Declaration>
};
#endif
```

The language C, following the ISO standards, do not support variables that always have 128 bits. However, some compilers like GCC and Clang support this as an extension. Using

the extension, we can compute the function g to obtain the next element from our random sequence using the following operations:

Section: SFMT: Compute next element:

```
#ifdef __SIZEOF_INT128__
unsigned __int128 result, tmp;
uint32_t aux[4];
int i, index = rng -> offset / 2;
result = ((unsigned __int128 *) (rng -> w))[index];
result = result << 8;
result = result ^ ((unsigned __int128 *) (rng -> w))[index];
i = (index + 122) % 156;
aux[0] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 0);
aux[1] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 1);
aux[2] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 2);
aux[3] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 3);
aux[0] = (aux[0] >> 11) & 0xDFFFFFEF; // 0xBFFFFFF6
aux[1] = (aux[1] >> 11) & 0xDDFECB7F; // 0xBFFAFFFF
aux[2] = (aux[2] >> 11) & 0xBFFAFFFF; // 0xDDFECB7F
aux[3] = (aux[3] >> 11) & 0xBFFFFFF6; // 0xDFFFFFEF
memcpy(&tmp, aux, 16);
result = result ^ tmp;
i = (index + 156 - 2) % 156;
result = result ^ (((unsigned __int128 *) (rng -> w))[i] >> 8);
i = (index + 156 - 1) % 156;
aux[0] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 0);
aux[1] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 1);
aux[2] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 2);
aux[3] = *(((uint32_t *) &(((unsigned __int128 *) (rng -> w))[i])) + 3);
aux[0] = (aux[0] << 18);
aux[1] = (aux[1] << 18);
aux[2] = (aux[2] << 18);
aux[3] = (aux[3] << 18);
memcpy(&tmp, aux, 16);
result = result ^ tmp;
((unsigned __int128 *) (rng -> w))[index] = result;
#else
#error "Mersenne Twister unsupported without 128 bit integer support."
#endif
```

Notice that if we are using a compiler without support for fixed-size variables with 128 bits, we return an error and refuse to compile.

The code above represents the multiplication between a matrix and a vector. Notice that the represented matrix is very sparse. If we are generating value x_i, we compute it using only the previous values x_{i-1}, x_{i-2}, x_{i-156} and x_{i-34}.

Knowing how to generate the next x_i with 128 bits, the code of our function that returns the next 64-bit pseudo-random value is:

Section: Definition: _Wrand:

```
#ifdef W_RNG_MERSENNE_TWISTER
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
```

<**Section to be inserted: Mutex:WAIT**>

```
  if(rng -> offset % 2 == 0){
                <Section to be inserted: SFMT: Compute next element>
  }
  ret = ((uint64_t *) (rng -> w))[rng -> offset];
  rng -> offset = (rng -> offset + 1) % (128 * 156 / 64);
                        <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
#endif
```

To initialize this generator we need to fill our previous 156 values obtaining it from our seed. Even with a smaller seed, we stretch it with a mini-random number generator just to fill the initial vector of previous values. The first two components of this mini-pseudo-random number generator are these functions:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_MERSENNE_TWISTER
static uint32_t f1(uint32_t x){
  return (x ^ (x >> 27)) * (uint32_t) 1664525UL;
}
static uint32_t f2(uint32_t x){
  return (x ^ (x >> 27)) * (uint32_t) 1566083941UL;
}
#endif
```

How we will initialize the struct is defined by the reference implementation. Therefore, like in the reference, we use a mini-generator based on 32-bit numbers to fill our initial state:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_MERSENNE_TWISTER
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    uint32_t *dst = (uint32_t *) (rng -> w), *origin = (uint32_t *) seed;
    size_t size_dst = 128 * 156 / 32, size_origin = size * 2;
    int count, r, i, j, mid = 306, lag = 11;
    // Preenchemos inicialmente tudo com 0x8b:
    memset(rng -> w, 0x8b, 128 * 156 / 8);
    count = ((size_origin + 1 >= size_dst)?(size_origin + 1):(size_dst));
    r = f1(dst[0] ^ dst[mid] ^ dst[size_dst - 1]);
    dst[mid] += r;
    r += size * 2;
    dst[mid + 11] += r;
    dst[0] = r;
    count --;
    for(i = 1, j = 0; j < count && j  < size_origin; j ++){
      r = f1(dst[i] ^  dst[(i + mid) % size_dst] ^
             dst[(size_dst - 1 + i) % size_dst]);
      dst[(i + mid) % size_dst] += r;
      r += origin[j] + i;
      dst[(i + mid + 11) % size_dst] += r;
      dst[i] = r;
      i = (i + 1) % size_dst;
```

```
    }
    for(; j < count; j++){
      r = f1(dst[i] ^ dst[(i + mid) % size_dst] ^
             dst[(i + size_dst - 1) % size_dst]);
      dst[(i + mid) % size_dst] += r;
      r += i;
      dst[(i + mid + 11) % size_dst] += r;
      dst[i] = r;
      i = (i + 1) % size_dst;
    }
    for (j = 0; j < size_dst; j++) {
      r = f2(dst[i] + dst[(i + mid) % size_dst] +
             dst[(i + size_dst - 1) % size_dst]);
      dst[(i + mid) % size_dst] ^= r;
      r -= i;
      dst[(i + mid + lag) % size_dst] ^= r;
      dst[i] = r;
      i = (i + 1) % size_dst;
    }
    rng -> offset = 0;
                    <Section to be inserted: SFMT: Ensure Period>
                      <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

Notice how before returning (and initializing the mutex) we check if the generator will have the desired period of $2^{19937} - 1$. We ensure this with a parity check. If parity is wrong, we change a single bit to ensure the correct period:

Section: SFMT: Ensure Period:

```
{
  // Bit mask to be checked
  uint32_t parity = (dst[0] & 0x00000001U) ^ (dst[3] & 0xc98e126aU);
  // Parity check:
  for (i = 16; i > 0; i >>= 1)
    parity ^= parity >> i;
  parity = parity & 1;
  if(parity != 1)
    dst[0] = dst[0] ^ 1;
}
```

Finally, we need to document the recommended seed size for this algorithm. We can perfectly deal with a seed of just 64 bits. In fact, in the reference implementation, it is possible to initialize with just 32 bits. If we pass a bigger seed, in the initialization we use the additional bits to compute the initial state. However, as we need to fill a vector with 19968 bits, providing more than 312 numbers with 64 bits will be redundant:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_MERSENNE_TWISTER
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  312
#endif
```

2.3. SplitMix64

SplitMix is a random number generator designed to be splittable. This means that exist an algorithm, that given the RNG structure, produce a new RNG structure such that both structures will generate unrelated random numbers with the same quality and without needing to provide a new seed.

But the real reason for providing the SplitMix algorithm here is not this property. We are not planning to support in our API splittable generators. Our real interest is in the different nature of this generator compared with others and the fact that it can be initialized with only 64-bit seeds.

Recall that in the end of Mersenne Twister section, we needed to consider the case of how to fill our initial stat with 312 bits if we get fewer random bits in the seed. For Mersenne Twister we used exactly the method suggested in the reference implementation, which consists in using another specific RNG just to fill the initial state. In these cases, we need a RNG that works with shorter seeds and papers like [Matsumoto, 2007] shows the importance of choosing radically different RNGs for this purpose, to avoid correlation of sequences generated with similar seeds.

Ignoring the splitting mechanism, SplitMix is very simple. It was created from the idea of having a state with 64 bits, and each time we need a new value, we update the state to a new value and return the result of a hash function over the RNG state.

Except that for performance reasons, instead of using a full hash function, we use a bit mixer: a component that is part of some hash functions and is responsible for avoid correlations between similar input values. The bit mixer used for SplitMix64 is the one defined originally for a hash function caled MurmurHash3:

Section: SplitMix: Bit Mixer:

```
{
  uint64_t tmp = *state;
  tmp = (tmp ^ (tmp >> 33)) * 0xff51afd7ed558ccdl;
  tmp = (tmp ^ (tmp >> 33)) * 0xc4ceb9fe1a85ec53l;
  ret = tmp ^ (tmp >> 33);
}
```

And before computing the above bit mixer, we also need to update the RNG state with the following line of code where `gamma` is an odd constant:

Section: SplitMix: Update State:

```
{
  *state +=  gamma;
}
```

The final construction of the function that returns a new pseudo-random number is:

Section: Auxiliary Functions:

```
#if defined(W_RNG_SPLITMIX) || defined(W_RNG_XOSHIRO) || defined(W_RNG_PCG) || \
  defined(W_RNG_CHACHA20)
static uint64_t splitmix_next(uint64_t *state, uint64_t gamma){
  uint64_t ret;
                    <Section to be inserted: SplitMix: Bit Mixer>
                   <Section to be inserted: SplitMix: Update State>
  return ret;
}
#endif
```

If we are using SplitMix not to initialize another RNG state, but as our real RNG, we will store this state and gamma value in the RNG struct:

Section: RNG Structs:

```
#ifdef W_RNG_SPLITMIX
struct _Wrng{
  uint64_t state, gamma;
                    <Section to be inserted: Mutex Declaration>
};
#endif
```

And the API function to generate a new pseudo-random number is:

Section: Definition: _Wrand:

```
#ifdef W_RNG_SPLITMIX
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                        <Section to be inserted: Mutex:WAIT>
  ret = splitmix_next(&(rng -> state), rng -> gamma);
                        <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
#endif
```

We can initialize the RNG structure using directly the seed to choose the initial state and, if possible, the gamma value. We just need to ensure that the gamma value is odd to have the expected period of 2^{64}:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_SPLITMIX
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  int i;
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size < 1)
      rng -> state = 0x32147198b5436569;
    else
      rng -> state = seed[0];
    if(size < 2)
      rng -> gamma = 0x9e3779b97f4a7c15;
    else
      rng -> gamma = (seed[1] | 1);
                        <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

This means that when using the SplitMix generator, we need a seed with 1 or 2 64-bit numbers:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_SPLITMIX
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  2
#endif
```

2.4. Xoshiro256**

Xoshiro and Mersenne Twister have some similarities. Both use past outputs directly to produce new inputs in a matrix multiplication. But contrary to Mersenne Twister, Xoshiro computes its sequences as 256-bit numbers, not 128-bit numbers. And instead of generating the next input storing all previous 156 outputs, Xoshiro stores only the last 256-number generated, requiring much less memory.

In the RNG struct we store the 256-number as a sequence of four 64-bit number:

Section: RNG Structs:

```
#ifdef W_RNG_XOSHIRO
struct _Wrng{
  uint64_t w[4];   // Valores de estado
                    <Section to be inserted: Mutex Declaration>
};
#endif
```

Like Mersenne Twister, to produce the next value, we multiply it by a matrix and use operations in GF(2): we use the operator AND in the bits instead of multiplication and operator XOR instead of sum. And we also use a matrix that at the same time produce good results and is convenient enough to not be explicitly stored. Instead, the matrix multiplication can be written using only a few fast operations, like combining AND, XOR and SHIFT.

In fact, the matrix multiplication to update the state is represented using only this code with 7 lines:

Section: Xoshiro: Matrix Multiplication:

```
{
  uint64_t t = rng -> w[1] << 17;
  rng -> w[2] ^= rng -> w[0];
  rng -> w[3] ^= rng -> w[1];
  rng -> w[1] ^= rng -> w[2];
  rng -> w[0] ^= rng -> w[3];
  rng -> w[2] ^= t;
  rng -> w[3] = ((rng -> w[3] << 45) | (rng -> w[3] >> 19));
}
```

But as we generate pseudo-random numbers using only a linear transformation and as our state is very small (contrary to Mersenne Twister), we need to use an additional trick to avoid failing in statistical tests. Instead of returning the real 256-bit number produced by our generator, we pass part of this value in a scrambler to produce the value that will be returned, masking the relationship between this returned value and the previous one:

Section: Xoshiro: Scrambler:

```
{
  uint64_t tmp = rng -> w[1] * 5;
  ret = ((tmp << 7) | (tmp >> 57)) * 9;
}
```

And finally, our function that returns the next pseudo-random value first uses the scrambler to compute the returned value `ret` using the current state, and then generates the next state using the matrix multiplication:

Section: Definition: _Wrand:

```
#ifdef W_RNG_XOSHIRO
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                        <Section to be inserted: Mutex:WAIT>
                      <Section to be inserted: Xoshiro: Scrambler>
```

```
                <Section to be inserted: Xoshiro: Matrix Multiplication>
                <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
#endif
```

Finally, we need to initialize Xoshiro**. If the seed have 256 bits or more, we copy the bits from the seed to the RNG state. If we have less, we copywhat we have and use the last 64 bits to initialize SplitMix64 state using the default gamma value and produce the remaining bits using the SplitMix generator:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_XOSHIRO
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  int i;
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size >= 4){
      for(i = 0; i < 4; i ++)
        rng -> w[i] = seed[i];
    }
    else{
      uint64_t state = 0x32147198b5436569, gamma = 0x9e3779b97f4a7c15;
      for(i = 0; i < size - 1; i ++)
        rng -> w[i] = seed[i];
      if(size > 1)
        state = seed[i];
      for(; i < 4; i ++)
        rng -> w[i] = splitmix_next(&state, gamma);
    }
                        <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

Because of this, we recommend that Xoshiro gets a seed with between one and four 64-bit elements:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_XOSHIRO
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  1
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  4
#endif
```

2.5. PCG (*Permuted Congruential Generator*)

The PCG random number generator is similar to SplitMix. It uses a simple operation to change the internal state. But instead of returning the internal state as the next number in the sequence, it applies a more complicated function to the internal state to produce the output.

The simple transformation in the state, in SplitMix means adding a constant to the internal state. But in PCG, we update the internal state applying a linear congruent generrator (LCG) like the one from subsection 2.1.

However, that linear congruent generator was defined using 64 bits. The internal state for PCG have 128 bits and therefore, we use a 128-bit LCG generator to update the internal state.

Our RNG struct is defined as below:

Section: RNG Structs:

```
#ifdef W_RNG_PCG
#ifdef __SIZEOF_INT128__
struct _Wrng{
  unsigned __int128 state;
  unsigned __int128 increment; // Almost an odd number, like in any LCG
                    <Section to be inserted: Mutex Declaration>
};
#else
#error "PCG unsupported without 128 bit integer support."
#endif
#endif
```

We update the internal state performing the LCG operation of multiplying by a constant multiplier carefully chosen and adding our odd increment to the result:

Section: PCG: Update State:

```
{
  unsigned __int128 multiplier;
  multiplier = 2549297995355413924ULL;
  multiplier = multiplier << 64;
  multiplier += 4865540595714422341ULL;
  rng -> state = rng -> state * multiplier + rng -> increment;
}
```

But like in SplitMix, instead of returning this state as the next number in the sequence, we pass this to some sort of hash function or bit scrambler. But in PCG case, what we do is combine the 128 bits in 64 bits using a XOR operation an then we apply a permutation over the result:

Section: PCG: Permutation:

```
{
  uint64_t xorshifted, rot;
  xorshifted = (((uint64_t)(rng -> state >> 64u)) ^ ((uint64_t) rng -> state));
  rot = rng -> state >> 122u;
  ret = (xorshifted >> rot) | (xorshifted << ((-rot) & 63));
}
```

And finally, the complete function to generate the next random number combines these operations as below:

Section: Definition: _Wrand:

```
#ifdef W_RNG_PCG
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                        <Section to be inserted: Mutex:WAIT>
                    <Section to be inserted: PCG: Update State>
                     <Section to be inserted: PCG: Permutation>
                       <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
```

```
#endif
```

What remains to be done is initialize the initial state. If we pass a big seed, with 256 bits or more, we could use it directly to fill the internal state. Instead, we mix them in a more complicated way to match the reference implementation that tries to deal with not so random initializations.

If we pass a seed with at least 128 bits, we use it directly to initialize the initial state and we set the increment of our internal LCG as 1. If we have 192 bits, we can use them to initialize to diffferent values in our increment. And if we have less than 128 bits, we use SplitMix to stretch our initial random bits to 128. The initialization function is defined as below:

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_PCG
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    if(size >= 4){
      unsigned __int128 multiplier;
      multiplier = 2549297995355413924ULL;
      multiplier = multiplier << 64;
      multiplier += 4865540595714422341ULL;
      unsigned __int128 initstate = seed[0], initseq;
      initstate = initstate << 64;
      initstate += seed[1];
      initseq = seed[2];
      initseq = initseq << 64;
      initseq += seed[3];
      rng->state = 0U;
      rng -> increment = (initseq << 1) | 1;
      rng->state = rng->state * multiplier + rng -> increment;
      rng -> state += initstate;
      rng->state = rng->state * multiplier + rng -> increment;
    }
    else if(size >= 2){
      rng -> state = seed[0];
      rng -> state = (rng -> state << 64);
      rng -> state += seed[1];
      if(size == 3){
        rng -> increment = seed[2];
        rng -> increment = (rng -> increment) | 1;
      }
      else
        rng -> increment = 1;
    }
    else{
      uint64_t state, gamma = 0x9e3779b97f4a7c15;
      if(size > 0)
        state = seed[0];
      else
        state = 0x32147198b5436569;
      rng -> state = splitmix_next(&state, gamma);
```

```
      rng -> state = (rng -> state << 64);
      rng -> state += splitmix_next(&state, gamma);
      rng -> increment = 1;
    }
                    <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

This means that our recommended seed size is a number between 2 and 4 to avoid needing to stretch the initial randomness with SplitMix:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_PCG
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  2
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  4
#endif
```

2.6. ChaCha20

ChaCha20 is a criptographically secure random number generator, but this affirmation is only true if it is used correctly: the seed must have a sufficiently large size, it should be chosen in a completely random and uniform way and it never should be reutilized again after the use. Our API does not guarantee these things, here we are not concerned with cryptographically secure random number generators. For us, ChaCha20 will be only another RNG like the previous ones. The difference is that it is expected to have a greater quality at the cost of being slower.

This algorithm has as first component a padding function that gets as input 384 bits that will be treated as a sequence of 64-bit numbers. The padding function outputs 512 bits that should be interpreted as a sequence of 32-bit numbers that form a 4×4 matrix. Even if we do not store them as a matrix.

The padding function is:

Section: ChaCha20: Padding:

```
#ifdef W_RNG_CHACHA20
static void chacha_padding(uint64_t input[6], uint32_t output[16]){
  int i, j;
  output[0] = ('e' << 24) + ('x' << 16) + ('p' << 8) + 'a';
  output[1] = ('n' << 24) + ('d' << 16) + (' ' << 8) + '3';
  output[2] = ('2' << 24) + ('-' << 16) + ('b' << 8) + 'y';
  output[3] = ('t' << 24) + ('e' << 16) + (' ' << 8) + 'k';
  for(j=4, i = 0; i < 6; i ++, j += 2){
    output[j] = (input[i] / 429496729611u);
    output[j+1] = input[i] % 429496729611u;
  }
}
#endif
```

In the input for this padding function, the first 4 values are taken from the initial seed, The next value is a counter initialized as 0 and incremented each time the generator produces a new set of numbers. The last value is a nounce, a number that should be used only once.

The next component is the function below that gets 4 values as 32-bit numbers and modify them doing a sequence of operations:

Section: ChaCha20: QuarterRound:

```
#ifdef W_RNG_CHACHA20
void quarter_round(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d){
  *a = *a + *b;
  *d = *d ^ *a;
  *d = (*d << 16) | (*d >> 16);
  *c = *c + *d;
  *b = *b ^ *c;
  *b = (*b << 12) | (*b >> 20);
  *a = *a + *b;
  *d = *d ^ *a;
  *d = (*d << 8) | (*d >> 24);
  *c = *c + *d;
  *b = *b ^ *c;
  *b = (*b << 7) | (*b >> 25);
}
#endif
```

The previous operation is used in the next permutation function that gets as input 16 numbers with 32 bits and transform them. Notice that in each iteration, the entire input is completely changed twice. As there are 10 iterations, we make 20 transformations in our input.

Section: ChaCha20: Permutao:

```
#ifdef W_RNG_CHACHA20
void chacha_permutation(uint32_t elements[16]){
  int i;
  for(i = 0; i < 10; i ++){
    quarter_round(&elements[0], &elements[4], &elements[8], &elements[12]);
    quarter_round(&elements[1], &elements[5], &elements[9], &elements[13]);
    quarter_round(&elements[2], &elements[6], &elements[10], &elements[14]);
    quarter_round(&elements[3], &elements[7], &elements[11], &elements[15]);
    quarter_round(&elements[0], &elements[5], &elements[10], &elements[15]);
    quarter_round(&elements[1], &elements[6], &elements[11], &elements[12]);
    quarter_round(&elements[2], &elements[7], &elements[8], &elements[13]);
    quarter_round(&elements[3], &elements[4], &elements[9], &elements[14]);
  }
}
#endif
```

With these prerequisites we can begin the definition of our RNG struct and functions for our API. Our RNG should store a vector with 6 elements, each one a 64-bit number (that will be used in the padding function to produce the next numbers), the last 512 bits generated by the RNG (if they exist) and an index to know which previously generated value should be returned. Our struct is:

Section: RNG Structs:

```
#ifdef W_RNG_CHACHA20
struct _Wrng{
  uint64_t array[6];
  uint32_t generated_values[16];
  int index;
                    <Section to be inserted: Mutex Declaration>
};
```

```
#endif
```

When this struct is allocated, the first 4 values in our 6-element array are copied from the seed, the next one is filled with 0 and the last one also is taken from the seed, but if it do not exist, we can use 0 (this is the nounce that identify the produced sequence for a given initial seed). This means that our seed in our API should have between 4 and 5 values:

Section: Warn About Ideal Seed Size:

```
#ifdef W_RNG_CHACHA20
#define _W_RNG_MINIMUM_RECOMMENDED_SEED_SIZE  4
#define _W_RNG_MAXIMUM_RECOMMENDED_SEED_SIZE  5
#endif
```

If we have less values we will use the SplitMix algorithm to produce the remaining values, except for the nounce that we can keep with 0. Our initialization function is given below.

Section: Definition: _Wcreate_rng:

```
#ifdef W_RNG_CHACHA20
struct _Wrng *_Wcreate_rng(void *(*allocator)(size_t), size_t size,
                           uint64_t *seed){
  int i, j;
  struct _Wrng *rng = (struct _Wrng *) allocator(sizeof(struct _Wrng));
  if(rng != NULL){
    for(i = 0; i < 4; i ++){
      if(i < size - 1 || size >= 4)
        rng -> array[i] = seed[i];
      else{
        // Using SplitMix to fill the remaining values
        uint64_t state, gamma = 0x9e3779b97f4a7c15;
        if(size > 0)
          state = seed[i];
        else
          state = 0x32147198b5436569;
        for(j = i; j < 4; j ++)
          rng -> array[j] = splitmix_next(&state, gamma);
        break;
      }
    }
    rng -> array[4] = 0;
    if(size > 4)
      rng -> array[5] = seed[4];
    else
      rng -> array[5] = 0;
    rng -> index = 0;
                        <Section to be inserted: Initializing Mutex>
  }
  return rng;
}
#endif
```

To produce new values, first we need to check if we still have generated bits that still were not returned. Each time ChaCHa20 produces new values, it produces 16 numbers with 32 bits. This means that we can use it as 8 numbers with 64 bits. When we do not need to produce new values, we proceed as below:

Section: Definition: _Wrand:

```
#ifdef W_RNG_CHACHA20
                    <Section to be inserted: ChaCha20: Padding>
                  <Section to be inserted: ChaCha20: QuarterRound>
                    <Section to be inserted: ChaCha20: Permutao>
uint64_t _Wrand(struct _Wrng *rng){
  uint64_t ret;
                          <Section to be inserted: Mutex:WAIT>
  if(rng -> index % 16 == 0){
    rng -> index = 0;
                <Section to be inserted: ChaCha20: Generating new values>
    rng -> array[4] ++;
  }
  ret = rng -> generated_values[rng -> index];
  ret = ret << 32;
  ret += rng -> generated_values[rng -> index + 1];
  rng -> index += 2;
                          <Section to be inserted: Mutex:SIGNAL>
  return ret;
}
#endif
```

And finally, when we really need to produce new values, we use the following code:

Section: ChaCha20: Generating new values:

```
{
  int i;
  uint32_t padded_array[16], copied_array[16];
  chacha_padding(rng -> array, padded_array);
  for(i = 0; i < 16; i ++)
    copied_array[i] = padded_array[i];
  chacha_permutation(padded_array);
  for(i = 0; i < 16; i ++)
    rng -> generated_values[i] = padded_array[i] + copied_array[i];
}
```

2.6. Estrutura Final do Arquivo

The file with th source code of our functions will have the following structure:

File: `src/random.c`:

```
                          <Section to be inserted: Including Headers>
#include "random.h"
#include <string.h> // memcpy
                          <Section to be inserted: Auxiliary Functions>
                          <Section to be inserted: Definition: “˙Wrand>
                        <Section to be inserted: Definition: “˙Wcreate“˙rng>
                       <Section to be inserted: Definition: “˙Wdestroy“˙rng>
```

3. Quality Tests

To test the quality of all algorithms defined in this work and compare them, we implemented the empirical tests proposed in [Knuth, 1998]. They are our initial set of tests, but more tests could be implemented and added to this work later.

A caveat is that lots of tests seggested by Knuth assume that we are generating random numbers between 0 and n or real numbers between 0 and 1. But for many tests this gives more weight to the more significative bits of a number and we want to give the same weight for all bits. Because of this, we treat the sequence generated by our RNG as a stream of bits, choosing different lengths of bits in each test to generate our numbers.

In most tests we apply the chi-square test as suggested in [Knuth, 1998] to test the quality and randomness of our results. We repeat each chi-square test three times and consider the test a failure if in any of them we obtain a very improbable result with a chance lesseR than 1% of happening. If we get two times a slightly improbable result (probability of happening: 5%) we also consider the test a failure. We repeat this triple test 1000 times. The expected result is a success of about 92.34% in all tests.

For all tests we use exactly the same seed with randomly chosen numbers. The numbers are: (`32147198b5436569`, `260287febfeb34e9`, `0b6cc94a91a265e4`, `c6a109c50dd52f1b`, `8298497f3992d73a`).

3.1. Equidistribution Test

In the most basic test we check the equidistributionjs of bits 0 and 1. The success for each algorithm is:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	91%
PCG	93%	`LCG`	93%
ChaCha20	91%		

3.2. Serial Test

In the serial test we generate sequence of d bits and check if each possible combination happens in a proportion of $1/2^n$. It is analogous to the equidistribution, but using more bits. Here we chosen $d = 15$. Our result is:

Gerador	Sucesso	Gerador	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	93%	`LCG`	91%
ChaCha20	93%		

3.3. Gap Test

This is a gap test for bits. We interpret the sequence generated by the RNG as asequence of bits and count the number of sequences of bits 0 followed by a 1. We treat them as "gaps".

For example, in the sequence "100110010100001" we get the following gaps: "1" (size 0), "001" (size 2), "1" (size 0), "001" (size 2), "01" (size 1) e "00001" (size 4). FOr each RNG we obtain a sample of $5 \cdot 2^{20}$ gaps and analyze the number of gaps for each size. We compare this with the expected result using the chi-square test. Our result is:

Generator	Sucesso	Generator	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	91%	`LCG`	86%
ChaCha20	91%		

3.4. Poker Test

Here we interpret our RNG sequence as sequences of numbers between 0 and 15. We generate a lot of tuples with 5 elements. Then, we count the occurence of the following events: all values in the tuple are different, we found 4 equal values, we found 3 equal values, we found a triple and a pair of equal values, we found two pairs of equal values, we found

a single pair of equal values. We measure how the number of each of these events compare with the expected numbers.

Our result is:

Generator	Sucesso	Generator	Sucesso
SFMT	92%	Xoshiro**	93%
PCG	91%	LCG	92%
ChaCha20	92%		

3.5. Collector Test

Like in the previous test we measure the equidistributivity of a hand of poker, here we measure how random is the order in which the cards appear. We interpret our RNG sequence as a sequence of numbers between 0 and 15. We keep generating values until we get all 16 different values finishing the collection. Next we count the number of generated values that were necessary to finish the collection. We reapeat this 4408394 times. And check in how many of these tests we managed to finish the collection generating only 16 numbers, only 17 numbers, and so on until 116+ generated values. We compare these results with what is expected in a real random sequence.

The result is:

Generator	Sucesso	Generator	Sucesso
SFMT	91%	Xoshiro**	92%
PCG	92%	LCG	00%
ChaCha20	92%		

This is the first test in which one of the algorithms fail. The LCG produces results that are too idealized to be considered random.

3.6. Teste da Permutao

Here we generate numbers between 0 and 7, discarding repeated values. In the end we produce a list with 8 values in some order. We repeat the test and count the number of times that each permutation of 8 numbers is produced, comparing the result with what is expected from a random sequence.

The result is:

Generator	Sucesso	Generator	Sucesso
SFMT	93%	Xoshiro**	92%
PCG	93%	LCG	02%
ChaCha20	92%		

3.7. Runs Up Test

Here, like in the previous test, we generate a permutation of elements. But instead of cunting each possible permutation, we check the number of ascending sequences in which each element is greater than the previous one.

FOr this, we generated permutations with numbers between 0 and 8192. And we counted the number of ascending sequences with size 1, 2, 3, 4, 5, 6+ elements.

The result is:

Generator	Sucesso	Generator	Sucesso
SFMT	92%	Xoshiro**	91%
PCG	90%	LCG	90%
ChaCha20	90%		

3.8. Maximum of t Test

Here we produce 3 numbers between 0 and 63 and choosed always the greatest value. Next we count how many times each number is chosen when we repeat the test and compare the result with what is expected.

The result is:

Generator	Sucesso	Generator	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	91%	LCG	86%
ChaCha20	92%		

3.9. Collision Test

Here we do not use the chi-square test because we are searching with elements that have a probability too small to be measured by this test. We measure the probability of finding a collision if we treat our RNG as a hash function.

For this test me assumed a 20-bit output for the RNG and produced 2^{14} different outputs. Next, we counted the number of collisions checking if this value is too greater or too smaller than expected. We repeat this test 3000 times. The number of expected collisions was chosen to produce a success rate near to 92% like in the previous tests.

The result is:

Generator	Sucesso	Generator	Sucesso
SFMT	94%	Xoshiro**	93%
PCG	94%	LCG	92%
ChaCha20	93%		

3.10. Birthday Spacing Test

Here we generate 512 different values, each one with 25 bits. Next we order them and measure the difference between succcessive numbers counting the number of equal differences found. The result is compared with what is expected from random sequences.

This test was originally suggested by George Marsaglia and at the time found problems in many generators that passed in other tests.

Our results are:

Generator	Sucesso	Generator	Sucesso
SFMT	92%	Xoshiro**	92%
PCG	92%	LCG	92%
ChaCha20	91%		

3.11. Teste da Correlao Serial

In this test we produce 1000 numbers with 64 bits that we convert to numbers between 0 and 1 dividing each number by $2^{64} - 1$ casting to floating points numbers with double precision. We compute the serial correlation coefficient between each number compared to the previous one, and measure if the result is in an acceptable range.

Next we repeat the same test not between each number and its successor, but between each number and the number two positions ahead in the sequence. Next with 3 positions ahead, and so on. We try to find the worst serial correlation coefficient.

We choose an acceptable range as one in which we expect to succeed in this test with probability about 93% given a random sequence.

Our result is:

Generator	Success	Generator	Success
SFMT	93%	Xoshiro**	93%
PCG	93%	LCG	93%
ChaCha20	93%		

3.12. Generating all 32-bit values

This is not a test like the previous ones. Instead of a percentage, we are interested if our random number generator can really produce all possible 32-bit numbers when executed for sufficient time. And if so, how many 32-bit numbers need to be generated. We assume that a generator that do not produce all the numbers fail.

Our result is:

Generator	Success	Generator	Success
SFMT	98852849277	Xoshiro**	92842427748
PCG	100979563727	LCG	8589934581
ChaCha20	99686167785		

All RNG algorithms were able to produce all possible 32-bit numbers and the number of tries were similar, except for LCG. The number of generated numbers for LCG was too low, half the produced values were new numbers, not repeated values. Like the collector test, this shows the known bias in LCG generators about not repeating previously generated values.

3.13. Test Conclusion

We managed to produce tests that find problems for the LCG algorithm. The result indicate a bias in that RNG against producing repeated values.

4. Benchmarks

To measure the performance for each algorithm, we produced 100 millions 64-bit numbers. To avoid compiler optimizations, we used each result in a sum. We tested the time spent in 4 different environments and 2 different computers.

The following test (with result in seconds) is the result of running the test in OpenBSD 6.7 using computer A (Intel Pentium B980 dual core, 2,40 GHz, 4 GB RAM) and compiler Clang 8.0.1:

Generator	Time (s)	Generator	Time (s)
SFMT	9.571183	Xoshiro**	7.811493
PCG	8.133946	LCG	7.411282
ChaCha20	10.498551	SplitMix64	7.529885

Repeating these tests using the same computer, but Windows 10 and compiling with Visual Studio (which does not support 128-bit variables, and cannot compile our implementation for PCG and Mersenne Twister), we get the following result:

Generator	Time (s)	Generator	Time (s)
SFMT	-	Xoshiro**	4.570623
PCG	-	LCG	3.615481
ChaCha20	8.019865	SplitMix64	4.340963

Repeating the test in a second newer computer, a Intel i5-3210M quad core with 2,50GHz and 4 GB RAM, running Ubuntu 20.04.2 and compiling with GCC 9.3.0, our result is:

Generator	Time (s)	Generator	Time (s)
SFMT	2.848333	Xoshiro**	2.565788
PCG	2.292101	LCG	2.378609
ChaCha20	7.085438	SplitMix64	2.437079

The most surprising result is PCG being faster than a linear congruential generator. This is very unexpected, as PCG uses internally a LCG. This is also the only case in which PCG run faster than Xoshiro** and SplitMix64. The result is not much different when compiling with Clang 10.0 in the same system:

Generator	Time (s)	Generator	Time (s)
SFMT	2.601277	Xoshiro**	2.449881
PCG	2.222556	LCG	2.348797
ChaCha20	4.620730	SplitMix64	2.346077

Using the same computer, but compiling using Emscripten 2.0.14 to produce Web Assembly and and executing the code in the browser Firefox 86.0, the result is:

Generator	Time (s)	Generator	Time (s)
SFMT	1.318	Xoshiro**	0.460
PCG	1.324	LCG	0.349
ChaCha20	3.687	SplitMix64	0.393

The surprising result is how faster this runs when compared with native code. Other than that, implementations which use 128-bit numbers appear to have a worse performance in this environment.

5. Conclusion: Choosing the Default Algorithm

When the user do not choose which algorithm our API will use, we should make for him an informed choice.

Our first criterion, of course, is if the algorithm is supported in the current architecture and environment. Algorithms PCG and Mersenne Twister will not compile if the chosen compiler do not support 128-bit numbers.

The second criterion is if the algorithm passed in our tests. Here only LCG is excluded, all other algorithms passed. Therefore, we will not consider it in our third criteria.

The third criterion is the speed. Here in all our tests SplitMix64 was the faster one, except when we tested in a Linux environment. It is possible that this happened because the default compilers were more recent and optimized better an implementation with 128-bit variables.

Combining these criteria, we end using SplitMix64 as the default random number generator, except if we are in Linux and we have a compiler that supports 128-bit variables:

Section: Choose Default RNG Algorithm:

```
#if !defined(W_RNG_MERSENNE_TWISTER) && !defined(W_RNG_XOSHIRO) && \
!defined(W_RNG_PCG) && !defined(W_RNG_LCG) && !defined(W_RNG_CHACHA20)
#if defined(__SIZEOF_INT128__) && defined(__linux__)
#define W_RNG_PCG
#else
#define W_RNG_SPLITMIX
#endif
#endif
```

References

Knuth, D. E. (1984) “Literate Programming”, The Computer Journal, volume 27, second edition, p. 97–111

Knuth, D. (1998) “The Art of Computer Programming, v. 2: Seminumerical Algorithms”, Addison-Wesley Professional, third edition.

Saito, M.; Matsumoto M. (2006) “SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator”, Monte Carlo and Quasi-Monte Carlo Methods, Springer, p. 607–622.

Steele, G.; Vigna S. (2021) “Computationally Easy, Spectrally Good Multipliers for Congruential Pseudorandom Number Generators”, arXiv preprint arXiv:2001.-05304.

Matsumoto, M; Wada I.; Kuramoto A.; Ashihara, H. (2007) “Common Defects in the Initialization of Pseudrandom Number Generators”, ACM Trans. Model. Comput. Simul, 17, 4.