ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

FPGA Implementations of Tiny Mersenne Twister

Guoping Wang
Department of Engineering, Indiana University Purdue University Fort Wayne, Fort Wayne, Indiana
wang@ipfw.edu

Abstracts

Random number generators are essential in many computing applications, such as Artificial Intelligence like genetic algorithms and automated opponents, random game content, simulation of complex phenomena such as weather and fire, numerical methods such as Monte-Carlo integration, cryptography algorithms such as RSA use random numbers for key generation, digital signal processing and communications, etc. Pseudo-random Number Generators (PRNGs) generate a sequence of “random” numbers using an algorithm, operating on an internal state, such as Linear Congruential Generator, Truncated Linear Congruential Generator, Linear Feedback Shift Register, Inversive Congruential Generator, Lagged Fibonacci Generator, Cellular Automata, Mersenne Twister, etc. The Mersenne Twister method, which avoided many of the problems with earlier generators and widely used in many applications, was proposed in 1998. In 2011, a tiny version of Mersenne Twister (TinyMT) was proposed. In some applications for example, where the large state size (19937 bits) of Mersenne Twister may be an obstruction for implementation. TinyMT is designed for such situation, with small state size and good randomness for that size of internal state. In this paper, FPGA implementations of four different TinyMT architectures were proposed and realized on Xilinx Virtex-4 FPGAs for the first time. The proposed designs can achieve very high throughput but with relatively very small areas.

Keyword: Pseudo Random Number Generator, Mersenne Twister, FPGA.

Introduction

Random number generators are essential in many computing applications, such as Artificial Intelligence like genetic algorithms and automated opponents, random game content, simulation of complex phenomena such as weather and fire, numerical methods such as Monte-Carlo integration, cryptography algorithms such as RSA use random numbers for key generation, digital signal processing and communications, etc. Pseudo-random Number Generators (PRNGs) generate a sequence of ``random" numbers using an algorithm, operating on an internal state, such as Linear Congruential Generator, Truncated Linear Congruential Generator, Linear Feedback Shift Register, Inversive Congruential Generator, Lagged Fibonacci Generator, Cellular Automata, Mersenne Twister, etc. PRNG algorithms are of active research, for both the quality and performance aspects in the following areas:

- As the execution speed is increasing, it demands a fast random number generation.
- The longer period is always better than the shorter one for PRNG generator. A good PRNG algorithm should have a long length of period to guarantee the randomness of the sequence.
- A fast PRNG should have a small number of internal state as high-speed memory is expensive.
- A fast PRNG algorithm should be able to generate independent multiple sequences concurrently or in parallel.

The Mersenne Twister method, which avoided many of the problems with earlier generators and widely used in many applications, was proposed in 1998 [1]. Two versions, MT11213 and MT19937, were developed with periods $2^{11213}-1$ and $2^{19937}-1$ (approximately 10^{6001}), which represents far more computation than is likely possible in the life of the entire universe. MT19937 uses an internal state of 624-bit longs, or 19968 bits, which is about expected for the huge period. It is (perhaps surprisingly) faster than the Linear Congruential Generators, is equidistributed in up to 623 dimensions, and has become the main RNG used in statistical simulations. The speed comes from only updating a small part of the state for each random number generated, and moving through the state over multiple calls. It is now increasingly becoming the random number generator

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

of choice for statistical simulations and generative modeling.

Hardware implementation on reconfigurable hardware, especially on Field Programmable Gate Array (FPGA)s has attracted a great deal of interest in the past 20 years as they can can offer very high performance of a dedicated hardware but also with the feature of programmability flexibility. There are few published reports on Mersenne Twister Implementation on FPGAs [2],[3],[4], and [5]. In [3], a three-stage of initialization process, generator, and output number extractor with pipelined structure was implemented in Xilinx XCV2000E FPGA. In [5], only the output section was implemented on Xilinx Virtex4 FPGAs.

In 2011, a tiny version of Mersenne Twister (TinyMT) was proposed [6] by Satio and Matsumoto. TinyMT is a variant of Mersenne Twister, specially designed with a small memory footprint. Satio and Matsumoto have shown that TinyMT passed the BigCrush tests of TestU01 [7]. There are two types of TinyMT, TinyMT32 and TinyMT64. TinyMT32 outputs 32-bit unsigned integers and single precision floating point numbers. On the other hand, TinyMT64 outputs 64-bit unsigned integers and double precision floating point numbers. TinyMT is not designed to replace Mersenne Twister. In some applications for example, in embedded system where, the large state size (19937 bits) of Mersenne Twister may be an obstruction for implementation. TinyMT is designed for such situation, with small state size and good randomness for that size of internal state. TinyMT is a small-sized pseudo random number generator, compared with Mersenne Twister or WELL RNG. TinyMT32 uses 16 bytes for its internal state vector and 12 bytes for its parameters, and TinyMT64 uses 16 bytes for its internal state vector and 16 bytes for its parameters. Pseudo random number sequences generated by TinyMT32 has a period of $2^{127}-1$.

In this paper, for the first time, an FPGA implementation of TinyMT32 was realized on Xilinx Virtex4 FPGA which can achieve very high throughput with very small area. Four different implementation architectures are proposed for various applications. The proposed designs are captured using VHDL and simulated to verify the correctness of its functionality using Mentor Graphics Modelsim simulator. The implementation results (Xilinx FPGA slices, block RAMs, maximum working frequency, throughput, etc) are compared to show that they can have a very high throughput with small area footprint compared to the full version of Mersenne Twister and other comparable PRNGs.

The rest of the paper is organized as follows: Section 2 introduces the algorithm of Mersenne Twister and Tiny MT. Section 3 gives the implementation of TinyMT on Xilinx FPGAs. The implementation results are described in Section 4 and Section 5 concludes this paper with the summary and conclusion.

Mersenne Twister and TinyMT Algorithms

In this section, background information of MT19937 and TinyMT algorithms is described. Mersenne Twister(MT) is a pseudo random number generating algorithm developed by Makoto Matsumoto and Takuji Nishimura in 1998 [1]. The MT algorithm was developed with the following merits:

- It is designed with consideration on the flaws of various existing generators.
- The algorithm is coded into a C-source downloadable.
- Far longer period and far higher order of equidistribution than any other implemented generators. It is proved that the period is $2^{19937}-1$, and 623-dimensional equidistribution property is assured.
- Fast generation. Although it depends on the system, it is reported that MT is sometimes faster than the standard ANSI-C library in a system with pipeline and cache memory.
- Efficient use of the memory. The implemented C-code mt19937.c consumes only 624 words of working area.

MT is a variant of Twisted Generalized Feedback Shift Register modification in order to allow a Mersenne prime period. The characteristic polynomial has many terms, and has good distribution up to v bits of accuracy for $32 \geq v \geq 1$. The Mersenne Twister algorithm generates a sequence of word vectors, which are considered to be uniform pseudo random integers between 0 to 2^w-1. Dividing it by 2^w-1, each word vector can be a floating point number in [0,1].

TinyMT is a variant of Mersenne Twister (MT) proposed by Saito and Matsumoto [6] in 2011. It is a small-sized pseudo random number generator, compared with Mersenne Twister or WELL RNG. TinyMT32 uses 16 bytes for its internal state vector and 12 bytes for its parameters, and TinyMT64 uses 16 bytes for its internal state vector and 16 bytes for its parameters. Pseudo random number sequences generated by TinyMT has a period of $2^{127}-1$. TinyMT is specially designed for a small memory footprint. On TinyMT, the users can generate multiple

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

independent sequences when choosing different sequence parameter sets from TinyMT Dynamic Creator (DC) [1]. The seed jumping function, which calculates the internal state of TinyMT after an arbitrary steps of the recursive state transitions, is also provided to make multiple non-overlapping sequences from the same sequence parameter sets. TinyMT is licensed under the BSD License as well as the other MT variants.

TinyMT is a combination of two different functions: the state transition function and the output function. Two different output functions with 32-bit and 64-bit tempering parameters are proposed. The size of TinyMT internal state with generation parameters for the 32-bit tempering parameter is 28 bytes including the 127-bit internal state and three 32-bit generation parameters. The period of each generated number sequences is $2^{127}-1$. Saito and Matsumoto [6] have showed that TinyMT passed the BigCrush tests of TestU01[7], and estimated that the total number of generation parameter sets MAT1, MAT2 and TMAT which could be generated by the TinyMTDC, a variation of MT DC algorithm for TinyMT, is 2^{58} regarding the number of computed irreducible polynomials. TinyMTDC is written in C++ and depends on Number Theory Library (NTL) [7].

There are two types of TinyMT: TinyMT32 and TinyMT64. TinyMT32 outputs 32-bit unsigned integers and single precision floating point numbers. On the other hand, TinyMT64 outputs 64-bit unsigned integers and double precision floating point numbers. TinyMT is not designed for a particular hardware. It will run on many types of hardware, because of its small size. TinyMT generates pseudo random numbers using two functions: a state transition function and an output function. The state transition function is an F2-linear function whose characteristic polynomial is irreducible of degree 127. The output function is not F2-linear, but almost F2-linear. The output function is composed from several F2-linear functions, except for one non F2-linear operation, namely an addition as integers modulo 232 (or mod 264 for TinyMT64) which replaces an F2-vector addition.

TinyMT is not designed to replace Mersenne Twister. In some applications, namely, hardware implementation or highly parallel environment, the large state size (19937 bits) of Mersenne Twister may be an obstruction for implementation. TinyMT is designed for such situation, with small state size and good randomness for that size of internal state.

The C-code implementation of TinyMT main functions can be seen in Figure 1.

```
void TinyMT32_init(TinyMT32_t * random, uint32_t seed) {
random->status[0] = seed;
random->status[1] = random->mat1;
random->status[2] = random->mat2;
random->status[3] = random->tmat;
for (int i = 1; i <MIN_LOOP; i++) {
random->status[i & 3] ^ = i + UINT32_C(1812433253) * (random-
>status[(i - 1) & 3]
^ (random->status[(i - 1) & 3] >>30));
}
period_certification(random);
for (int i = 0; i <PRE_LOOP; i++) {
TinyMT32_next_state(random);
} }

void TinyMT32_next_state(TinyMT32_t * random) {
uint32_t x;
uint32_t y;
y = random->status[3];
x = (random->status[0] & TINYMT32_MASK) ^ random->status[1]
^ random->status[2];
x ^= (x <<TINYMT32_SH0);
y ^= (y >>TINYMT32_SH0) ^ x;
random->status[0] = random->status[1];
random->status[1] = random->status[2];
random->status[2] = x ^ (y <<TINYMT32_SH1);
random->status[3] = y;
random->status[1] ^= -((int32_t)(y & 1)) & random->mat1;
random->status[2] ^= -((int32_t)(y & 1)) & random->mat2;
}

uint32_t TinyMT32_temper(TinyMT32_t * random) {
uint32_t t0, t1;
t0 = random->status[3];
t1 = random->status[0]+ (random->status[2] >>TINYMT32_SH8);
t0 ^= t1; t0 ^= -((int32_t)(t1 & 1)) & random->tmat;
return t0;
}
```

Figure 1. TinyMT Functions in C

This paper mainly focuses on the FPGA implementations of TinyMT32. Various implementation architectures are described for different applications.

FPGA Implementation of TinyMT-32

An analysis of TinyMT algorithm shows that the process of generating Tiny Mersenne Twister number 32-bit can be separated into the following 4 steps:

1) Using TinymtDC, generating the parameters MAT1,MAT2,TMAT required for TinyMT32.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

2) In Tinymt32, based on the input parameters MAT1,MAT2, and TMAT, initialize the four 32-bit TinyMT status words with initial random 32-bit seed. This process consists of MIN_LOOP, Period Certification and Init Next State update.
3) Calling the Next State function, update the TinyMT four 32-bit status words and generating the untempered numbers.
4) Calling the temper function, outputting the generated random 32-bit number.

The initialization process only runs once at the beginning of its generation, and it will not be called again during the generating process. Moreover, the generating and tempering processes only require logic and XOR operating which consume relatively few logic resources on FPGAs. In our implementation, step 1 is performed with software in advance, the generating three 32-bit parameters, that is, MAT1, MAT2 and TMAT can be stored in constant memory (ROM). Depending on the requirements of the project, if only a limited random number sequence is needed, then step 2) can also be computed using software in advance and stored on FPGA memory. Otherwise, step 2) to 4) can be implemented on FPGA hardware. For example, assuming that we can use 1024 Byte parameter ROM to store the parameters generated by TinymtDC and assuming that each group (3 parameters) takes up 3 32-word, then the 1024 byte can store more than 64 distinct group parameters to be used by TinyMT generator. The initialization step will update the TinyMT 4 32-bit status words with the initial 32-bit seed based on these preset parameters. The TinyMT can generate more than 64 distinct 32-bit random number sequences. If the number of these distinct group random number sequence is enough, then in the system, we only need FPGA to implement step 3 and 4. Otherwise, steps 2 to 4) are implemented in the FPGA hardware, in which the initial seed ranges from 0 to $2^{32}-1$ and the TinyMT can generate a very large number sequence of random numbers, as high as 64×2^{32} distinct random number sequence.

In the following, four different FPGA implementations of TinyMT32 are described, that is, TinyMT engine 1 without ROM, TinyMT engine 1 with ROM, TinyMT engine 2 without ROM and TinyMT engine 2 with ROM.

A. Design 1: TinyMT32 Engine 1 without ROM

In this implementation, the parameters generated from TinyMTDC are initialized/updated by software in advance into four 32-bit status words and are loaded into the TinyMT engine. The initialized parameters are not stored on FPGA memory, instead, they are loaded before the generating process. This design takes the smallest area and can achieve the highest throughput. Figure 2 shows the schematic of such a design. However, since the parameters are not stored on FPGA, each time a new random sequence is generated, a new group parameters (the initialized four 32-bit status words and MAT1,MAT2 and TMAT from TinyMTDC) will be loaded externally into the FPGA.

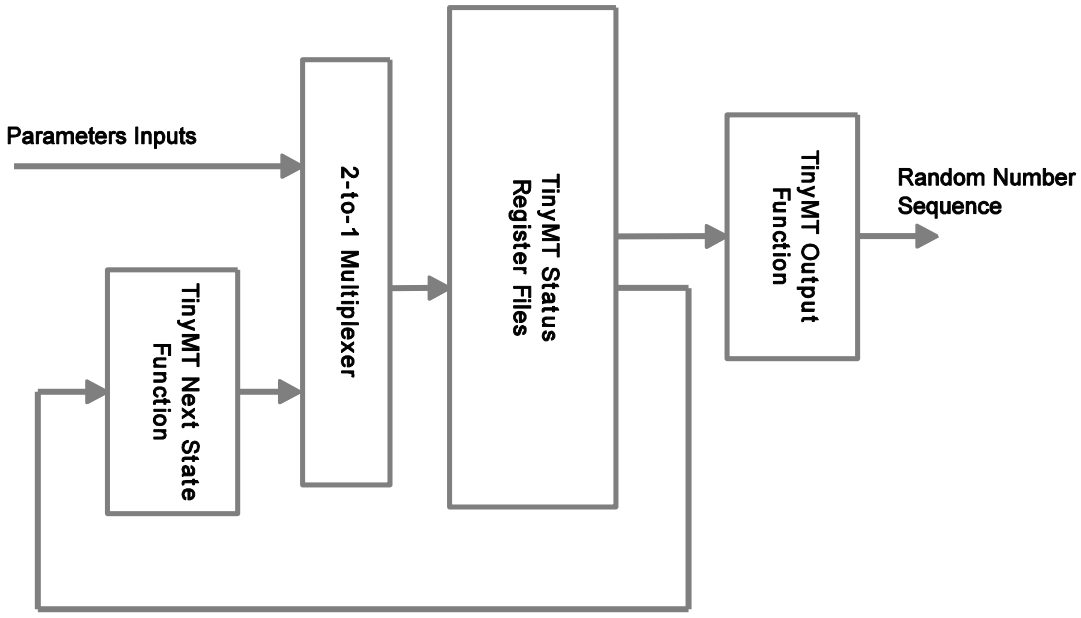

Figure 2. TinyMT Engine 1 without FPGA Memory

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

B. Design 2: TinyMT32 Engine 1 with on-chip ROM

In this implementation, the parameters generated from TinyMTDC are initialized/updated by software in advance into four 32-bit status words. They are stored on FPGA block RAMs. Assuming that 1024 byte on-chip RAM is used to store the precalculated parameters. Each group status word takes up four 32-bit, that is, 16 bytes, and 1K RAM can store 64 distinct group precalculated parameters. Since both the initialized status words (four 32-bit) and MAT1, MAT2 and TMAT from TinyMT DC are required for the generated, totally 7 block RAMs are needed to store these parameters for generation. One advantage compared to implementation 1 is that implementation 2 can generate up to 64 different random number sequences. Figure 3 shows the schematic of implementation 2. This design has the small memory footprint and high throughput, with a little complex controller circuit.

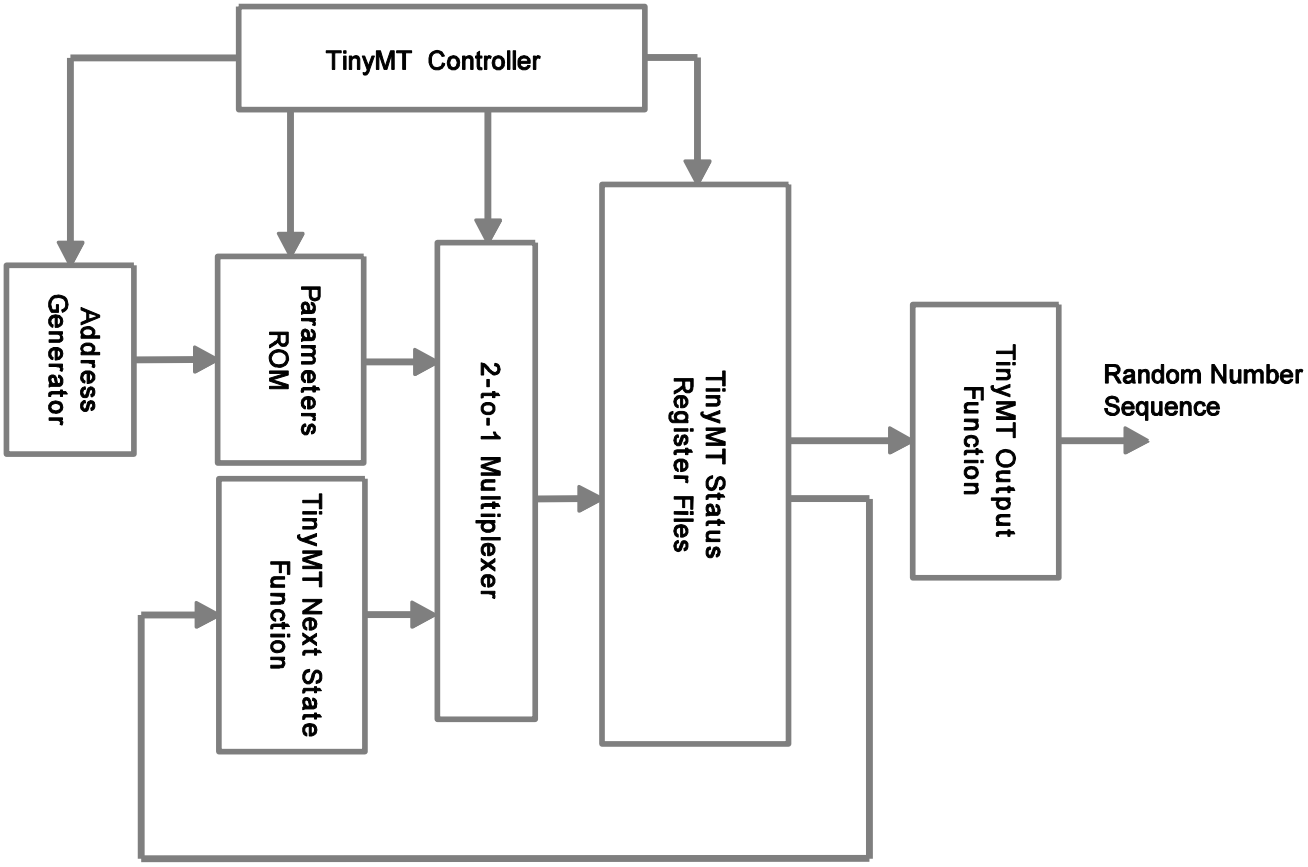

Figure 3. TinyMT Engine 1 with on-chip FPGA Memory

C. Design 3: TinyMT32 Engine 2 without ROM

The difference between designs 1, 2 and designs 3, 4 is that, in designs 3 and 4, the parameters MAT1, MAT2 and TMAT are generated by TinyMTdc in advance using software, then they are loaded into the TinyMT engine 2 for hardware initialization and update. Thus the initialization process is realized on hardware instead of software. In designs 3 and 4, the random initial 32-bit seed can be initialized into a value between *0* and $2^{32}-1$, thus up to 2^{32} different random number sequence can be generated for one group parameter, MAT1, MAT2 and TMAT. If 1024 byte on-chip RAM is accessible on FPGA, then up to 64×2^{32} distinct random number sequence can be created.

In design 3, the parameters MAT1,MAT2, and TMAT are loaded into the TinyMT engine when a new random number sequence is needed with the schematic shown as Figure 4.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

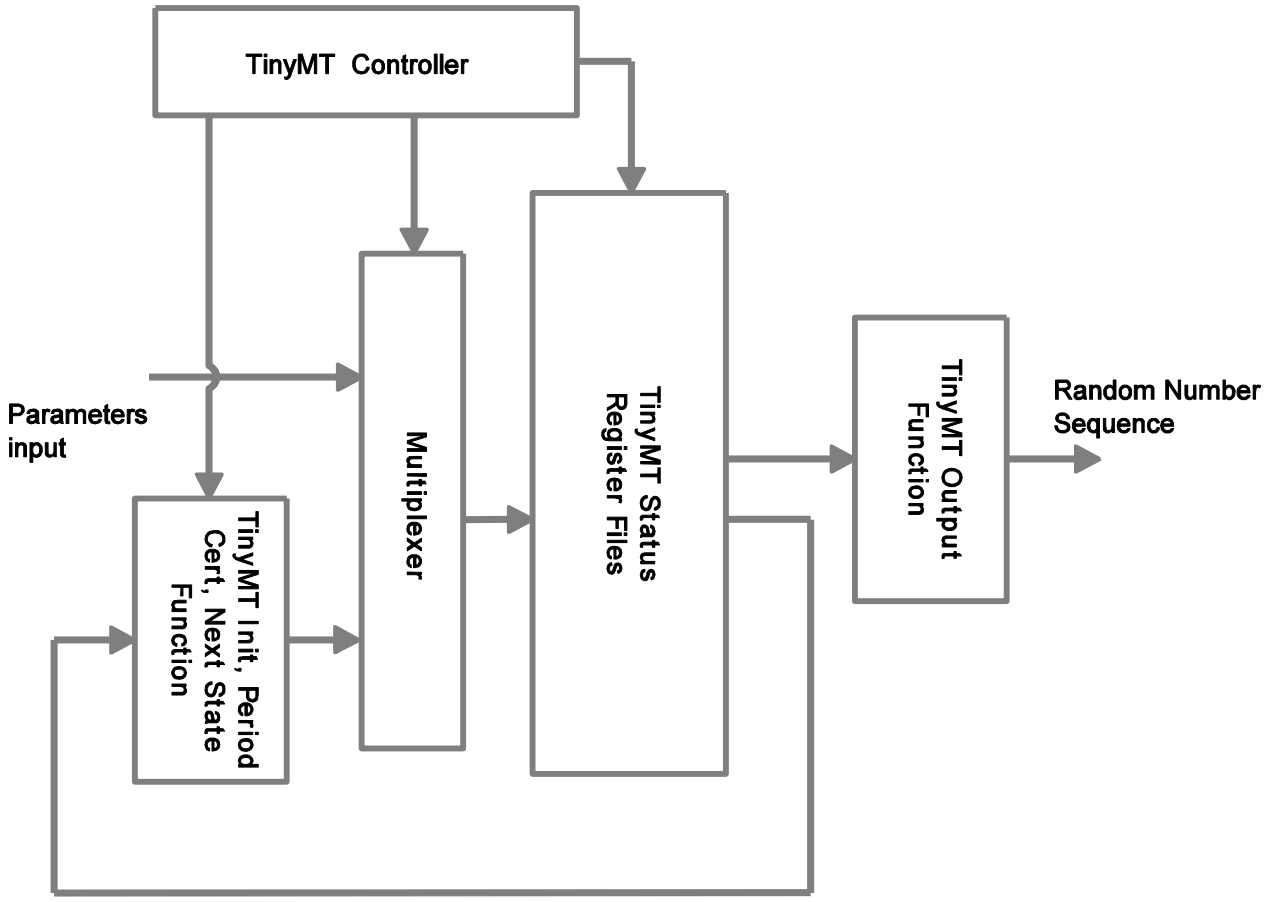

Figure 4. TinyMT Engine 2 without on-chip FPGA Memory

D. Design 4: TinyMT32 Engine 2 with parameters stored in ROM

This is the most flexible TinyMT engine design. Sixty-four distinct group of MAT1,MAT2 and TMAT parameters are stored on FPGA 1024 byte RAM. Each time, when a new random number sequence is needed, the engine loads one group of parameters. When all 64 group parameters are used up, then the random seed is increased by 1, hence, a new random number sequence can be generated. This implementation takes up the most space with the sacrifices of a little more space.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

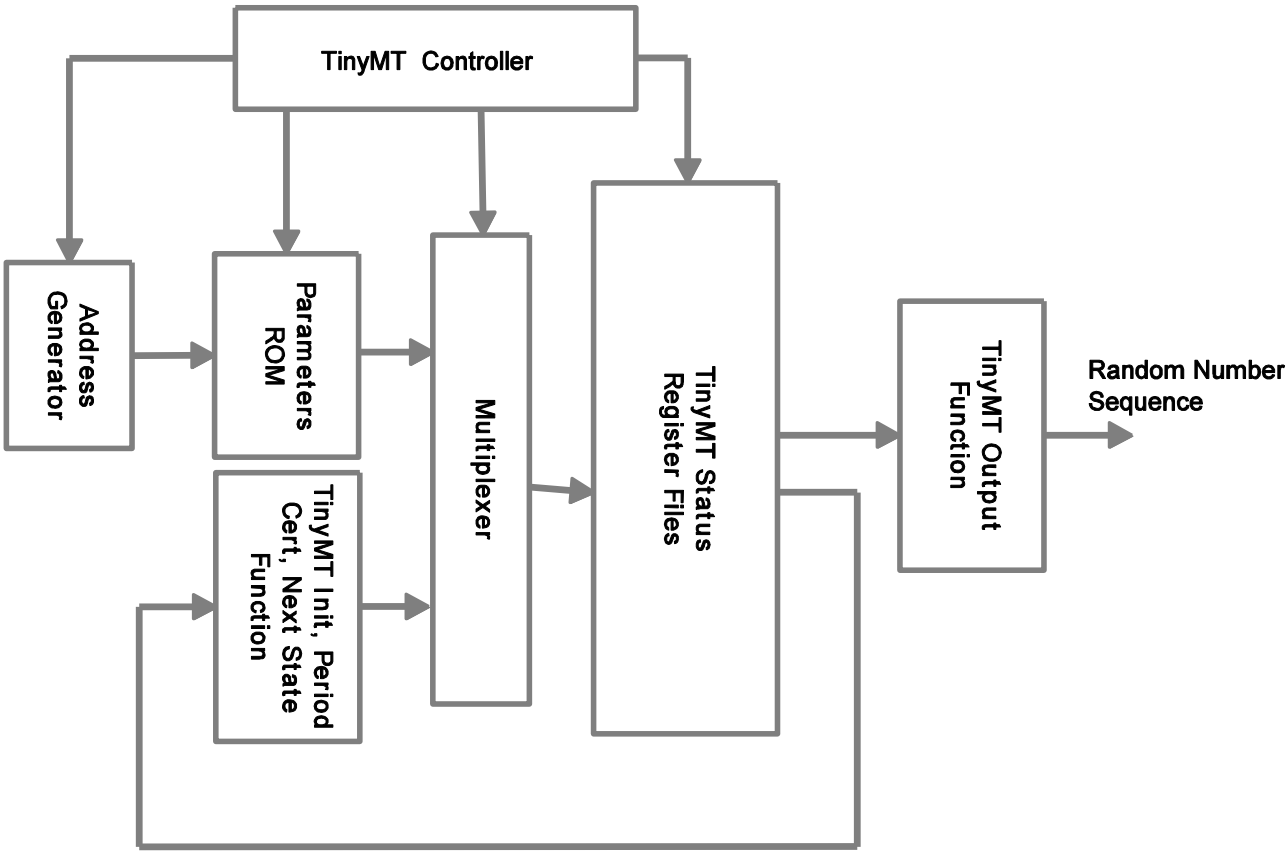

Figure 5. TinyMT Engine 2 without on-chip FPGA Memory

The block diagram can be seen in Figure 5. In this implementation, the data path consists of block RAM (or registers), controller (Finite State Machine), Multiplexers. After reset, the seed is initialized to a predetermined number (for example, 0), then the FSM controls the TinyMT engine into initialization state, in which the MAT1,MAT2,TMAT and SEED are converted and updated. This initialization states consist of three substates, that is, MIN_LOOP, PERIOD_CERTIFRICATION and Init Next State. This initialization state is only executed once in the beginning of the generation. After the initialization state, the FSM controls the TinyMT engine into generating state. In this generating state, it generates a sequence of 32-bit random number every clock until it reaches the required quantity of random number. The generated untempered 32-bit random number is tempered for the output. For design 4, a total number of 64×2^{32} distinct random number sequence can be generated.

1. Results and Comparisons

The proposed Tinymt designs 1, 2 3 and 4 are captured with VHDL using Xilinx ISE 13.2 ISE Suite, simulated using Modelsim SE simulator to verify this functionality and implemented on Xilinx Virtex4 VFX 100 FPGA (xc4vfx100-10ff152).

Table *1* describes the synthesis results with the maximum frequency, number of slices, number of LUTs used, RAM, number of DSP48s and throughput for each design.

Table 1. Performance Analysis Comparison of Tinymt Engines

Design	RAMs	# of slices	# of LUTs	Max Freq. (Mhz)	# of DSP48s	Throughout Gb/s
1	0	206	355	390	0	12.48
2	7	175	335	349	0	11.17
3	0	468	818	87	3	2.78
4	3	444	859	87	3	2.78

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

Table 2. Comparison of Design Parameters with Other PRNGs

Design	FPGA	# of Slices	# of LUTs	# of RAMs	Max Freq (Mhz)
Tinymt Engine 4	Virtex-4	444	859	3	87
[2]	N/A	420	N/A	N/A	N/A
[3]	Virtex E	330	539	2	24.23
[4]	Virtex-5	2028	N/A	N/A	111

From
Table *1*, we can see that, for implementations of TinyMT designs 1 and 2, the number of slices, maximum working frequency, highest throughput are very similar except that 7 64x32-bit RAMs are needed for TinyMT design 2 due to the fact that the parameters from TinyMT DC and initialized parameters are stored on FPGA RAMs. For the implementations of TinyMT designs 3 and 4, the number of slices, maximum working frequency, the number of DSP48s, highest throughput are also very similar except that 3 64x32-bit RAMs are needed for TinyMT design 4 since the parameters MAT, MAT2 and TMAT from TinyMT DC are stored on FPGAs. Designs 1 and 2 have a much higher throughput and maximum working frequency and takes up smaller areas compared to designs 3 and 4, because the initialization process are hardware implemented for TinyMT designs 3 and 4. TinyMT designs 3 and 4 also need three additional DSP48s for the initialization. For applications which need limited random number sequence, TinyMT designs 1 and 2 could be good candidates. If a large distinct number of random number sequence is required, then TinyMT designs 3 and 4 will good choices. If there are additional block RAM available on FPGAs, TinyMT designs 2 and 4 are better choices than designs 1 and 3.

Since no previous FPGA implementation of TinyMT is available for comparison, we only compare the proposed TinyMT design with MT19937 full version and the results can be seen in
Table *2*.
From this table, we can see that the proposed TinyMT implementation can achieve a significantly higher throughput and occupies less area compared to the previous MT1997 on FPGA implementations.

Conclusion and Summary

Random number generators are essential in many computing applications. TinyMT, a variant PRNG of the popular Mersenne Twister, has been implemented on FPGA for the first time in this paper. Four TinyMT engines are proposed for different application areas. The proposed implementations show a significant high throughput and with small areas which are applicable in fields where area is a concern.

Reference

[1]. M. Matsumoto, T. Nishimura, Dynamic creation of pseudorandom number generators, in: In Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, 1998, pp. 56–69. URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/articles.html

[2]. V. Sriram, D. Kearney, An area time efficient field programmable mersenne twister uniform random number generator, in: Proceedings of 17 the International Conference on Engineering of Reconfigurable Systems and Algorithms, 2006.

[3]. S. Chandrasekaran, A. Amira, High performance fpga implementation of the mersenne twister, in: Electronic Design, Test and Applications, 2008. DELTA 008. 4th IEEE International Symposium on, 2008, pp. 482 –485. doi:10.1109/DELTA.2008.113.

[4]. D. Pellerin, E. Trexel, M. Xu, Fpga-based hardware acceleration of c/c++ based applications, in: Impulse Accelerated Technologies, 2007. URL http://www.eetimes.com/design/programmable-logic/4015124/FPGA-based-hardware-acceleration-of-C-C--based-applications--Part-1

[5]. X. Tian, K. Benkrid, Mersenne twister random number generation on fpga, cpu and gpu, in: Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on, 2009, pp. 460 –464. doi:10.1109/AHS.2009.11.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 2.114

[6]. M. Saito, M. M., A high quality pseudo random number generator with small internal state [in japanese], in: Information Processing Society of Japan (IPSJ) SIG Notes, no. 2011-HPC-131(3), 2011, pp. 1–6. URL http://ci.nii.ac.jp/naid/110008620834/en/

[7]. P. L'Ecuyer, R. Simard, Testu01: A c library for empirical testing of random number generators, ACM Trans. Math. Softw. 33 (4). doi:10.1145/1268776.1268777.URL http://doi.acm.org/10.1145/1268776.1268777

[8]. V. Shoup, NTL: A library for doing number theory (2009). URL http://www.shoup.net/ntl/