卒業論文

HDL による乱数発生回路
メルセンヌ・ツイスタの設計と実装

氏名　　：土屋 直幸
学籍番号：2210020294-1
指導教員：山崎 勝弘 教授
提出日　：2006 年 2 月 20 日

立命館大学　理工学部　情報学科

内容梗概

本論文では、ハードウェア記述言語 Verilog-HDL による、メルセンヌ・ツイスタ乱数発生回路の実装について述べる。メルセンヌ・ツイスタのリファレンスプログラムを関数ごとに初期化モジュールを 2 つ、乱数生成モジュールを 1 つ、メインモジュールを 1 つの計 4 つにモジュール分割し、実装した。実装したメルセンヌ・ツイスタの回路では、入力値として、符号無し 32bit の乱数の種と生成する乱数の個数を与えると、符号無し 32bit の乱数を生成することができる。

本研究では、メルセンヌ・ツイスタをハードウェア化することにより、擬似乱数ではない真の乱数を生成して、真の乱数生成の高速化をすることが目的である。実際にメルセンヌ・ツイスタをハードウェア化することにより、約 1.77 倍の高速化を確認した。また、パイプライン化を行うと、約 95.88 倍の速度向上が得られることを確認した。乱数生成速度は回路構成の改良により、さらなる速度向上の見込みがある。

目次

図目次

表目次

1. はじめに

乱数は、情報科学やゲーム業界ではもちろん、科学全般、社会学、経済学など、広い分野で利用されている。乱数には擬似乱数と擬似ではない真の乱数が存在するが、世の中で使用されている乱数のほとんどが擬似乱数である。しかし、暗号・セキュリティの分野では擬似乱数をそのまま使用すると、暗号が解読される危険性が増すため、擬似乱数生成アルゴリズムをハードウェア化して得られる、擬似ではない真の乱数が必要となる。この先、超高速の計算が可能な量子コンピュータが開発されると、因数分解のような天文学的な計算量でも、すぐに計算できてしまうため、擬似乱数の暗号では、どんなに周期を大きくしても、組み合わせ全てを試す総当り法で暗号が破られてしまう。真の乱数を得るには、外部回路として、原子核崩壊の雑音や電子回路の熱雑音などから、ランダムな値を得る必要がある。

一方、LSI の小型化により、回路規模は大きく、かつ複雑になってきているため、回路図によって部品の接続関係を記述する従来の回路設計では限界になってきている。そこで開発されたのが、ハードウェア記述言語（HDL）である。HDL の普及により、設計者の作業量は軽減され、設計期間は短縮し、コストは大幅に削減された。HDL で記述した回路は論理合成ソフトを使用して回路に変換される。現在では、ハードウェア設計において、HDL が不可欠である。代表的な言語は Verilog-HDL と VHDL の２つである。Verilog-HDL は C 言語に近い言語で抽象的で記述性が良い。VHDL は世の中で初めて標準化されたハードウェア記述言語で、厳格な文法の言語である。本研究では Verilog-HDL を扱う。

真の乱数を得るために、Verilog-HDL 言語により、乱数発生回路を作成する。乱数を得るためには、乱数生成アルゴリズムが必要だが、現時点で存在しているすべての乱数生成アルゴリズムの中で、乱数性や生成速度などの面で最も優れているといわれているものが、メルセンヌ・ツイスタというアルゴリズムである。メルセンヌ・ツイスタは乱数の周期が非常に長く、生成される乱数のばらつきが良く、線形フィードバックシフトレジスタのような単純な構造なので、生成速度が速い。さらに、メモリ効率は良く、32bit×624 個のワーキングメモリを使用するだけである。しかも、メルセンヌ・ツイスタは計算量が多く、生成される乱数の偏りが大きいといった、従来のアルゴリズムの欠点を考慮して開発されたので、欠点が少ない。メルセンヌ・ツイスタのアルゴリズムは、乱数を使ったシミュレーションに用いられることが多い。本研究では、乱数生成アルゴリズムとして、メルセンヌ・ツイスタを使用する。

本研究では、メルセンヌ・ツイスタのホームページに公開されている、C 言語で記述されたメルセンヌ・ツイスタのプログラムから、メルセンヌ・ツイスタのアルゴリズムのハードウェア化を行う。そのために、まず Verilog-HDL 言語で記述可能となるような、また、ハードウェア化したときに、乱数の生成速度ができる限り速くなるような、乱数発生回路メルセンヌ・ツイスタの構成を考える。考えたメルセンヌ・ツイスタの構成をもとに、関

数ごとに初期化モジュールを 2 つ、乱数生成モジュールを 1 つ、メインモジュールを 1 つの計 4 つにモジュール分割を行い、Verilog-HDL 言語で実装する。すべてのモジュールを接続し、実装が完了すると、検証のためのシミュレーションを行う。シミュレーションでは、メルセンヌ・ツイスタのハードウェア化が正常に行われたかどうかを確認するために、いろいろな値の種を入力値として与え、乱数のばらつき具合を確認する。また、ハードウェア化したときの回路規模の大きさや、動作周波数を確認する。動作周波数と乱数生成に必要なクロック数から、乱数生成速度を求める。そして、C 言語で記述されたメルセンヌ・ツイスタのプログラムを実行したときの実行時間と比較して、ハードウェア化による乱数生成速度の向上を確認する。

本論文では、第 2 章でメルセンヌ・ツイスタの特徴について述べる。第 3 章ではメルセンヌ・ツイスタをハードウェア化したときの利点や意義を述べる。第 4 章ではメルセンヌ・ツイスタの構成と実装について述べる。第 5 章では実行結果の評価をして、考察する。最後に、第 6 章では本研究の成果と今後の課題について述べる。

2.乱数発生回路メルセンヌ・ツイスタ

「メルセンヌ・ツイスタ」とは松本眞氏・西村拓士氏によって、1996 年から 1997 年に渡って開発された疑似乱数生成アルゴリズムである[6]。乱数とは、サイコロなどによる、今までに生成された数から次の数が予測不可能で生成の規則が分からないため、以前の状況の再現が不可能な数列である。しかし、コンピュータは指示された計算を行うだけの機械であるため、コンピュータ上で作る事が可能な乱数は、できるだけ生成される数値の予測を困難に、生成される数値の偏りを小さくする必要がある。コンピュータの計算によって生成された乱数を擬似乱数という。全ての擬似乱数は周期を持ち、1 周期ごとに同じ数列を繰り返し生成する性質を持つ。プログラムを用いて生成可能な乱数は全て擬似乱数になる。

メルセンヌ・ツイスタの長所は以下の点がある。

- $2^{19937}-1$ という非常に長い周期を持つ。
- 最高で 623 次元超立方体の中に 均等に分布する。
- 生成速度もアルゴリズムが複雑なわりに、掛け算や割り算などの命令が少ないので、処理が速い。
- メモリ効率も良く、32bit×624 ワードのワーキングメモリで計算できる。
- 計算量が多く、生成される乱数の偏りが大きいといった、従来の乱数生成アルゴリズムの欠点を考慮して開発された。

メルセンヌ・ツイスタのフローチャートを図 1 に示す。

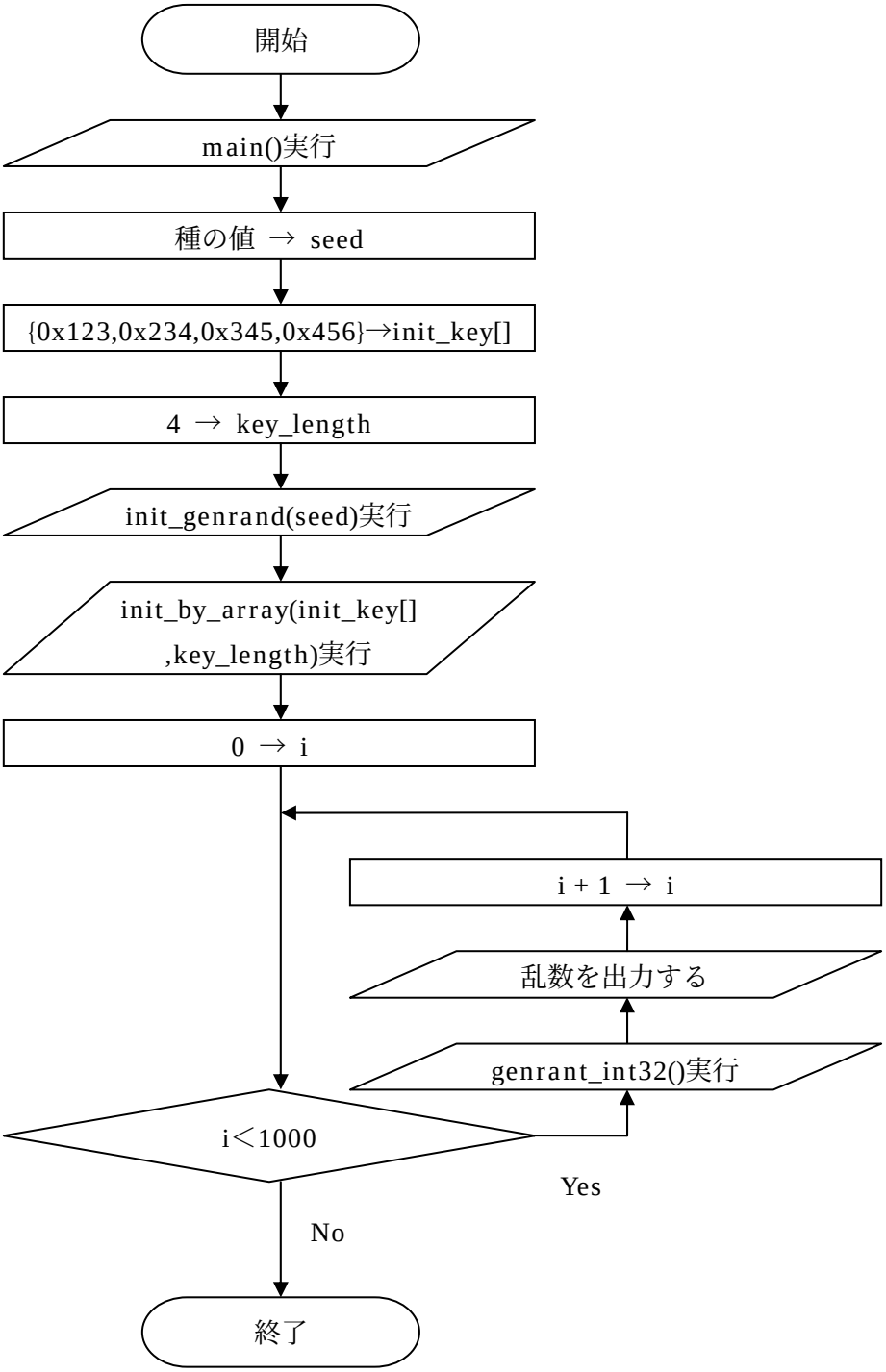

図 1:メルセンヌ・ツイスタのフローチャート

メルセンヌ・ツイスタのアルゴリズムの説明と各関数について説明する。

main 関数

- 各パラメータを設定する。
- init_genrand 関数、init_by_array 関数を実行する。
- 乱数を生成する個数だけ、繰り返し genrand_int32 関数を実行して、生成された乱数を出力する。

init_genrand 関数

- 0 番地に種を代入する。
- i 番地の値を得るために、i-1 番地の値を使用する。シフト、EXOR、乗算、加算の順に演算を行う。
- 623 番地まで計算を繰り返し、値を代入する。
- この時点でメモリのすべての番地に値が入る。

init_by_array 関数

- i 番地の値を得るために、i-1 番地の値を使用する。init_genrand 関数とは異なる値を使用し、シフト、EXOR、乗算、加算の順に演算を行う。
- 1 番地から 623 番地の値が更新される。
- 0 番地は更新されていないので、最後に、0 番地に 0x80000000 を代入する。
- この時点でメモリの初期化が完了する。

genrand_int32 関数

(a)(624n+1)回目の場合(n=0,1,2,…)

- 0 番地から 227 番地まで AND、OR、EXOR 演算を行う。
- 228 番地から 623 番地まで、別の数値を使って AND、OR、EXOR 演算を行う。
- 再び別の数値を使って、AND、OR、EXOR 演算を行い、y に代入する。

(b)毎回共通の部分

- y にシフト、AND、EXOR 演算を行い、生成された乱数を返す。

3.メルセンヌ・ツイスタのハードウェア化による利点と意義

メルセンヌ・ツイスタをハードウェア化することで、外部回路で得た値を乱数の種として使用することが可能となり、全くでたらめな数を使用することができるため、擬似ではない真の乱数を得ることができる。また、ハードウェア化すると、真の乱数の特徴と擬似乱数の特徴の両方の良い面を持った乱数が得られる。

擬似乱数は同じ種からは同じ乱数の数列が生成される。擬似乱数はシミュレーションをするときは最適であるが、暗号を生成する場合、擬似乱数の性質上、簡単に暗号がわかってしまう。ハードウェア化すると、毎回違う乱数を得ることができるため、暗号に使用することが可能である。

真の乱数と擬似乱数の違いを述べる。

(1)真の乱数

・特徴

原子核崩壊時の雑音・大気中の雑音・電子回路の熱雑音などの外的要因を元のデータとして、ビット列を生成する。

・利点

次に出現するビット列の予測が不可能であり、暗号鍵として利用すると安全性が高い。

・欠点

再び同じ系列の乱数列を利用することが不可能であり、再現性に乏しい。また、余分な外部回路が必要となり、コストがかかる。外部回路の雑音に偏りがあると、生成される乱数にも偏りが出る。

・応用範囲

一時的に使用される、再現性の必要がない乱数列に対してはこれらの乱数を使用することが望ましい。

(2)擬似乱数

・特徴

ある特定のアルゴリズムにより次々と計算を行い、ビット列を生成していくことによって発生する擬似的な乱数である。

・利点

ソフトウェアやハードウェアの内部のみで乱数を生成可能であり、余計な外部回路がいらないため、全体構成が簡潔となる。また、動作が高速であることが多い。真の乱数よりも偏りが小さい。

・欠点

特定のアルゴリズムにより、乱数を生成しているため、生成される乱数列には、ある規

則性ができてしまう。そのため、生成アルゴリズムは秘匿にするか、外部から見ても簡単にはわからないように複雑にする必要がある。

・応用範囲

再現性の必要なコンピュータシミュレーションなどに用いられることが多い。

(3)ハードウェア化によって得られる乱数

・特徴

原子核崩壊時の雑音・大気中の雑音・電子回路の熱雑音などの外的要因を乱数の種として使用する。そして、ある特定のアルゴリズムにより次々と計算を行い、ビット列を生成していく。真の乱数と擬似乱数の特徴を併せ持つ。

・利点

ハードウェア化すると、ソフトウェアで乱数生成を行うよりも、乱数生成速度が向上する。パイプライン化すると、1 クロックで実行できるよりになり、乱数生成速度が飛躍的に向上する。次に出現するビット列の予測が不可能であり、暗号鍵として利用すると安全性が高い。

・欠点

余分な外部回路が必要となり、コストがかかる。外部回路の雑音に偏りがあると、種にも偏りが出て、その結果、生成される乱数にも偏りが出る。

・応用範囲

再現性の必要なコンピュータシミュレーションなどに用いられることが多い。暗号鍵としても使用することができる。

4.メルセンヌ・ツイスタの構成と実装

4.1 構成

メルセンヌ・ツイスタの構成を図 2 に示す。

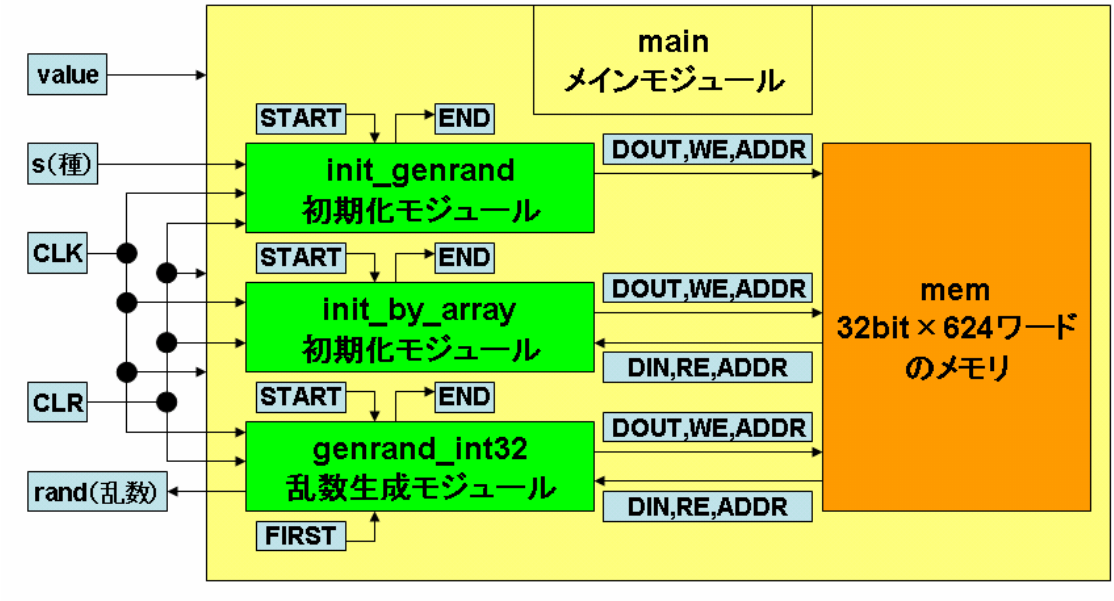

図 2:メルセンヌ・ツイスタの構成図

メルセンヌ・ツイスタの処理の流れ

1、種に適当な値（正の整数）を入力する。value に生成する乱数の個数を入力する。
2、START が立ち上がり、init_genrand の処理（メモリの初期化）を行う。
3、init_genrand の処理（メモリの初期化）が終わり、END を返す。
4、START が立ち上がり、init_by_array の処理（メモリの初期化）を行う。
5、init_by_array の処理（メモリの初期化）が終わり、END を返す。
6、START が立ち上がり、genrand_int32 の処理（乱数生成）を行う。
7、１回目の乱数生成の場合は FIRST を立ち上げる。
8、genrand_int32 の処理（乱数生成）が終わり、END を返す。
9、乱数を出力する。生成した乱数の個数が value の値より小さければ、６へ戻る。
１０、value の値だけ乱数を生成したら終了する。

・CLR が立ち上がるとリセットして 1 番へ戻る。
・WE が 1 のとき、ADDR でアドレスを設定し、DOUT の値をメモリへ格納する。
・RE が 1 のとき、ADDR でアドレスを設定し、メモリから DIN の値を取り出す。
・WE、RE が共に 1 のときは WE が優先される。

乱数を 1 個生成した場合のメルセンヌ・ツイスタの計算量を表 1 に示す。

表 1:メルセンヌ・ツイスタの計算量（乱数を 1 個生成した場合）

単位：演算を行う回数

	init_genrand	init_by_array	genrand_int32	main	計
加算	1246	3119	1873	1	6239
減算	1246	4988	1815	0	8049
乗算	623	1247	0	0	1870
比較	623	1872	625	0	3120
シフト	623	1247	1025	0	2895
AND	0	0	3065	0	3065
OR	0	0	1021	0	1021
EXOR	623	2494	2046	0	5163
計	4984	14967	11470	1	31422

乗算は他の演算に比べて、大幅に時間がかかる処理である。乗算はメモリの初期化をするときのみ行っている。加算、減算は、1 だけ足す、または 1 だけ引くために使われているものが約 80%を占めているので、それほど時間がかかる処理ではない。

乱数を 624 個生成した場合のメルセンヌ・ツイスタの計算量を表 2 に示す

表 2:メルセンヌ・ツイスタの計算量（乱数を 624 個生成した場合）

単位：演算を行う回数

	init_genrand	init_by_array	genrand_int32	main	計
加算	1246	3119	2496	624	7485
減算	1246	4988	1815	0	8049
乗算	623	1247	0	0	1870
比較	623	1872	625	0	3120
シフト	623	1247	3517	0	5387
AND	0	0	4311	0	4311
OR	0	0	1021	0	1021
EXOR	623	2494	4538	0	7655
計	4984	14967	18323	624	38898

乱数を 624 個生成した場合は、乱数を 1 個生成した場合と比べて、表 2 の中で緑で示した部分の値が変化する。生成する乱数を 624 個に増やしても、計算量が増えるのは genrand_int32 モジュールと main モジュールだけである。乗算は計算量が変わっていない。

4.2 実装

実装はハードウェア記述言語の Verilog-HDL を用いて行った。実装の手順は以下の通りである。

(1)init_genrand モジュールを作成する。
(2)main モジュールと init_genrand モジュールを接続する。
(3)init_by_array モジュールの作成する。
(4)main モジュールと init_by_array モジュールを接続する。
(5)genrand_int32 モジュールの作成する。
(6)main モジュールと genrand_int32 モジュールを接続する。

すべてのモジュールは状態遷移機械とデータパスで実装した。

各モジュールの説明

(1) init_genrand モジュール

init_genrand モジュールの構成を図 3 に示す。

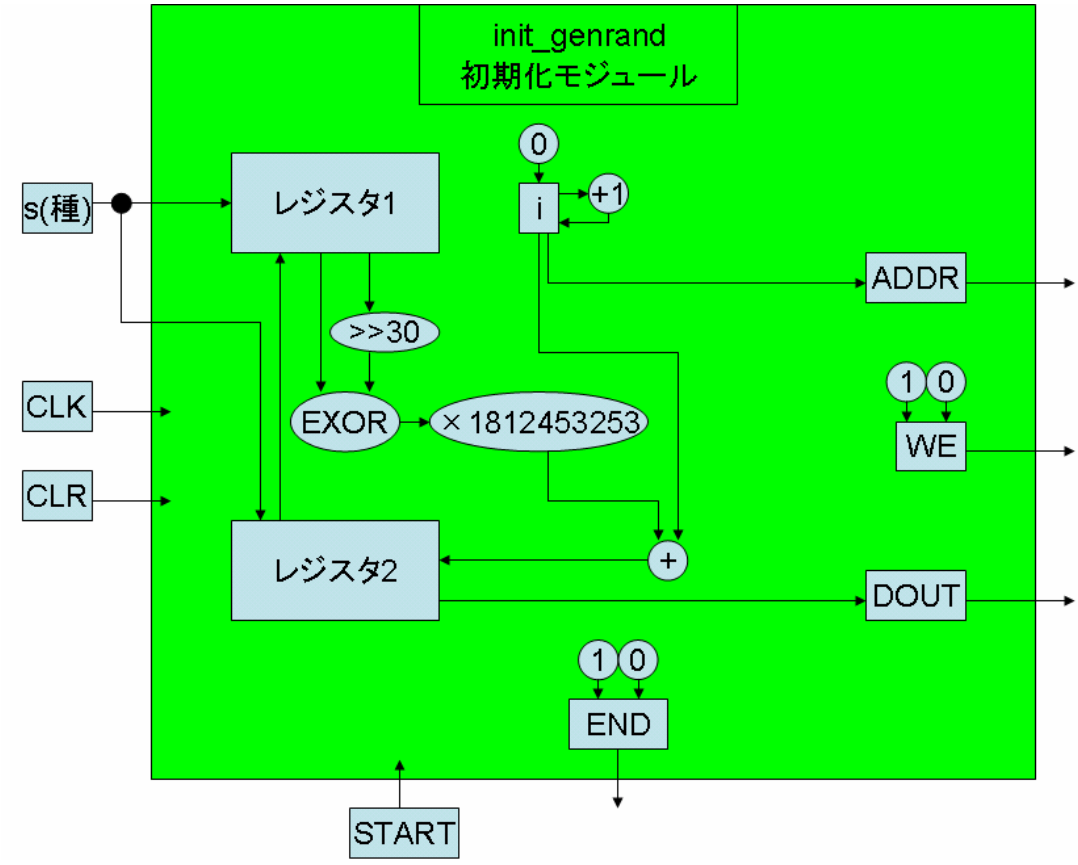

図 3：init_genrand モジュールの構成

init_genrand モジュールの状態遷移表を表 3 に示す。

表 3：init_genrand モジュールの状態遷移表

状態	処理	次の状態
00	END = 0; WE = 0; if(START) goto 01; else goto 00; START が立ち上がるまで状態 00 で待つ。START が立ち上がると状態 01 へ進む。	00,01
01	reg1 = s; reg2 = s; i = 0;	08
02	if(i<624) goto 03; else goto 12;	03,12
03	temp1 = reg2 >> 30;	04
04	temp2 = reg2 ^ temp1;	05
05	temp3 = 1812433253 * temp2;	06
06	reg1 = i + temp3;	07
07	reg2 = reg1;	08
08	ADDR = i; DOUT = reg1;　メモリのアドレス、書き込む値を設定する。	09
09	WE = 1;　メモリへ値を格納する。	10
10	WE = 0;	11
11	i = i + 1;	02
12	END=1;　プログラムを終了する。	00

状態数：13

init_genrand モジュールは、値が格納されていないメモリに、種から演算を行って、値を格納する初期化モジュールである。START=1 になると処理が始まる。END=1 になると処理が終わる。レジスタを 2 つ用意して、片方は i 番地、もう片方は i-1 番地として処理を行った。ADDR にメモリのアドレス、DOUT にメモリに格納する値を設定して、WE=1 になるとメモリに値を書き込む。

(2)init_by_array モジュール

init_by_array モジュールの構成を図 4 に示す。

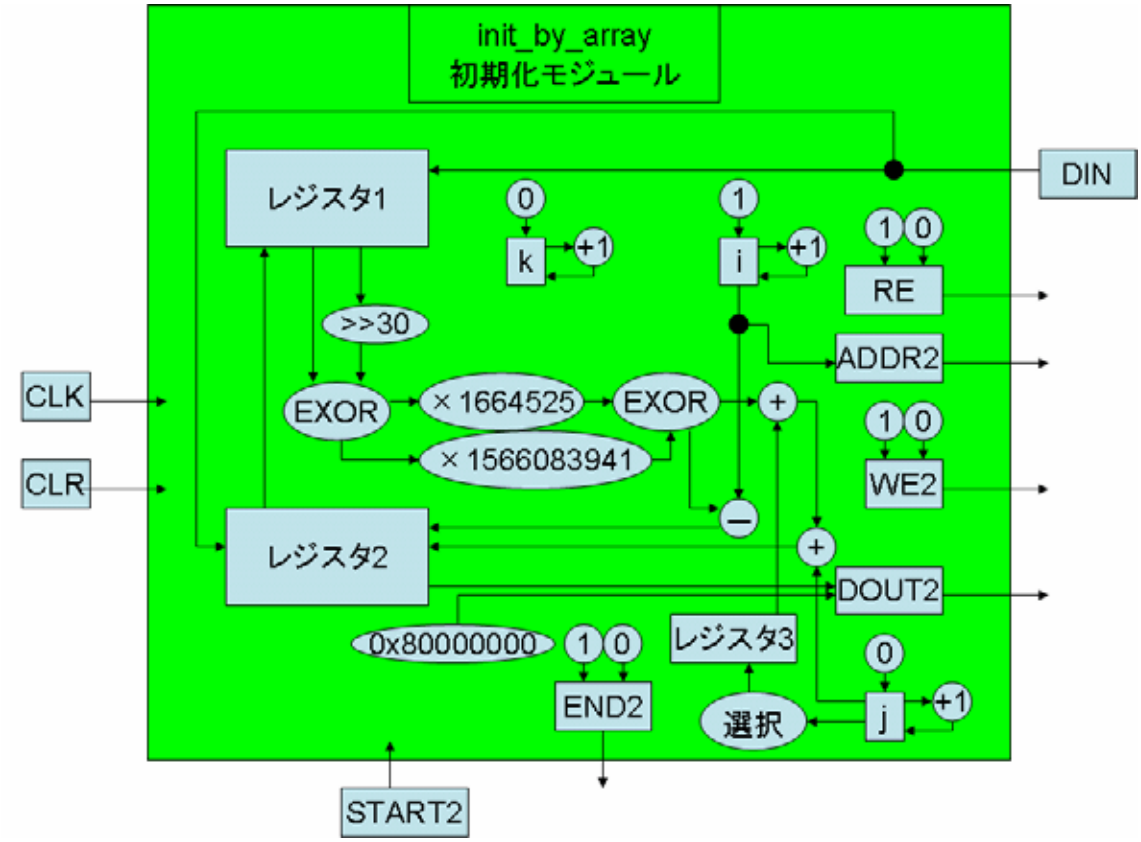

図 4：init_by_array モジュールの構成

init_by_array モジュールの状態遷移表を表 4 に示す。

表 4：init_by_array モジュールの状態遷移表

状態	処理	次の状態
00	END2 = 0; WE2 = 0; RE = 0; if(START2) goto 01; else goto 00; START2 が立ち上がるまで状態 00 で待つ。START2 が立ち上がると状態 01 へ進む。	00,01
01	i = 1; j = 0; k = 0;	02
02	ADDR2 = 0; RE = 1;	03
03	DIN に値を格納するために 1 クロック待つ。	04
04	reg2 = DIN;　DIN の値を取り出す。	05
05	RE = 0;	06
06	if(k<624) goto 07; else goto 37;	07,37
07	ADDR2 = i; RE = 1;	08

08	DIN に値を格納するために 1 クロック待つ。	09
09	reg1 = DIN;	10
10	RE = 0;	11
11	if(j==0) goto 12; else goto 13;	12,13
12	reg3 = 0x123;	13
13	if(j==1) goto 14; else goto 15;	14,15
14	reg3 = 0x234;	15
15	if(j==2) goto 16; else goto 17;	16,17
16	reg3 = 0x345;	17
17	if(j==3) goto 18; else goto 19;	18,19
18	reg3 = 0x456;	19
19	temp1 = reg2 >> 30;	20
20	temp2 = reg2 ^ temp1;	21
21	temp3 = 1664525 * temp2;	22
22	temp4 = reg1 ^ temp3;	23
23	temp5 = reg3 + temp4;	24
24	reg1 = j + temp5;	25
25	ADDR2 = i; DOUT2 = reg1;	26
26	WE2 = 1;	27
27	WE2 = 0;	28
28	reg2 = reg1;	29
29	i = i + 1; j = j + 1; k = k + 1;	30
30	if(i>=624) goto 31 else goto 35;	31,35
31	ADDR2 = 0; DOUT2 = reg2;	32
32	WE2 = 1;	33
33	WE2 = 0;	34
34	i = 1;	35
35	if(j>=4) goto 36 else goto 06;	06,36
36	j = 0;	06
37	k = 0;	38
38	if(k<623) goto 39; else goto 58;	39,58
39	ADDR2 = i; RE = 1;	40
40	DIN に値を格納するために 1 クロック待つ。	41
41	reg1 = DIN;	42
42	RE = 0;	43

43	temp1 = reg2 >> 30;	44
44	temp2 = reg2 ^ temp1;	45
45	temp3 = 1566083941 * temp2;	46
46	temp4 = reg1 ^ temp3;	47
47	reg1 = temp4 - i;	48
48	ADDR2 = i; DOUT2 = reg1;	49
49	WE2 = 1;	50
50	WE2 = 0;	51
51	reg2 = reg1;	52
52	i = i + 1; k = k + 1;	53
53	if(i>=624) goto 54 else goto 38;	38,54
54	ADDR2 = 0; DOUT2 = reg2;	55
55	WE2 = 1;	56
56	WE2 = 0;	57
57	i = 1;	38
58	ADDR2 = 0; DOUT2 = 0x80000000;	59
59	WE2 = 1;	60
60	WE2 = 0;	61
61	END2 = 1;　プログラムを終了する。	00

状態数：62

init_by_array モジュールは、init_genrand モジュールの処理によって格納した値をメモリから取り出して、値を更新して、メモリに格納する初期化モジュールである。START2=1 になると処理が始まる。END2=1 になると処理が終わる。init_by_array モジュールでは、メモリから値を取り出す処理が追加し、ADDR2 にアドレスを設定して、RE=1 になると DIN にメモリの値を取り出す。

(3)genrand_int32 モジュール

genrand_int32 モジュールの構成を図 5 に示す。

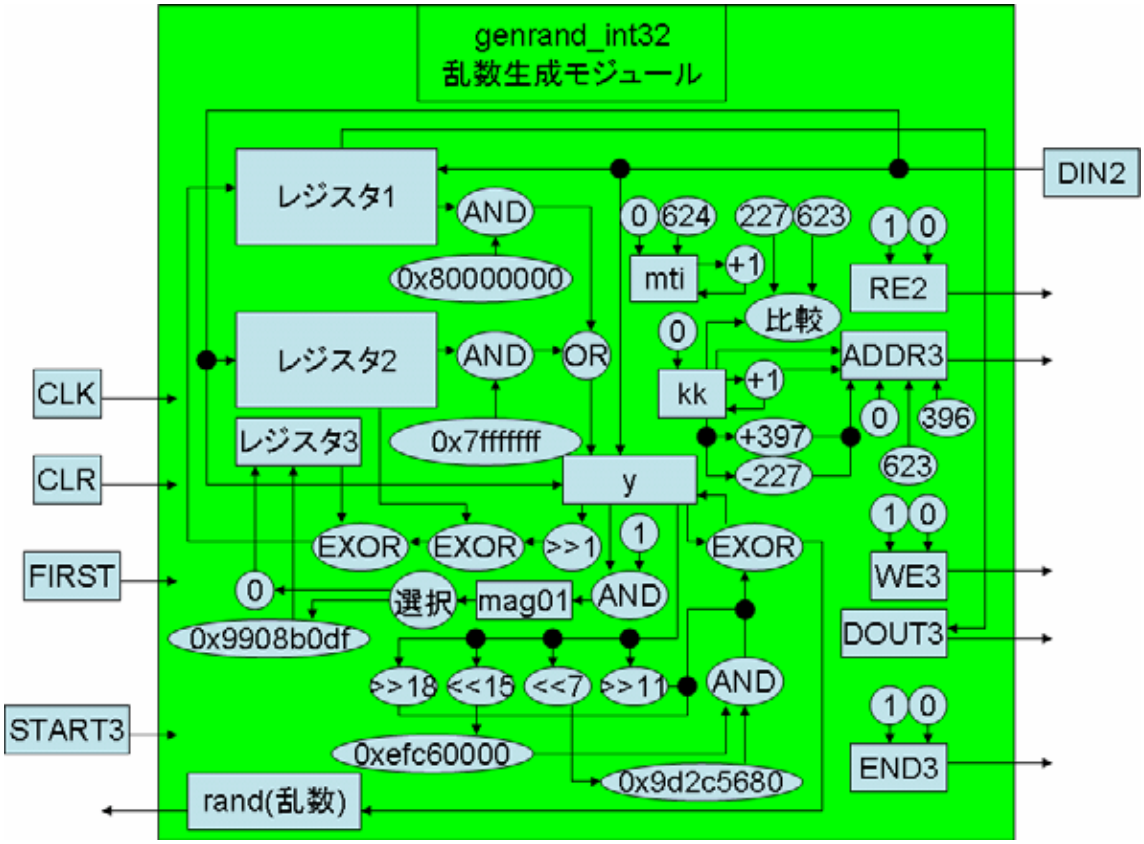

図 5：genrand_int32 モジュールの構成

genrand_int32 モジュールの状態遷移表を表 5 に示す。

表 5：genrand_int32 モジュールの状態遷移表

状態	処理	次の状態
00	END3 = 0; WE3 = 0; RE2 = 0; if(START3) goto 01; else goto 00; START3 が立ち上がるまで状態 00 で待つ。START3 が立ち上がると状態 01 へ進む。	00,01
01	if(FIRST) goto 107; else goto 02;	02,107
02	if(mti>=624) goto 03 else goto 87	03,87
03	kk = 0;	04
04	if(kk<227) goto 05; else goto 32;	05,32
05	ADDR3 = kk; RE2 = 1;	06
06	DIN2 に値を格納するために 1 クロック待つ。	07
07	reg1 = DIN2;	08

08	RE2 = 0;	09
09	ADDR3 = kk + 1; RE2 = 1;	10
10	DIN2 に値を格納するために 1 クロック待つ。	11
11	reg2 = DIN2;	12
12	RE2 = 0;	13
13	temp1 = reg1 & 0x80000000UL;	14
14	temp2 = reg2 & 0x7fffffffUL;	15
15	y = temp1 | temp2;	16
16	ADDR3 = kk + 397; RE2 = 1;	17
17	DIN2 に値を格納するために 1 クロック待つ。	18
18	reg2 = DIN2;	19
19	RE2 = 0; goto 20;	20
20	mag01 = y & 0x1UL;	21
21	if(mag01==0) goto 22; else goto 23;	22,23
22	reg3 = 0x0UL;	23
23	if(mag01==1) goto 24; else goto 25;	24,25
24	reg3 = 0x9908b0dfUL;	25
25	temp1 = y >> 1;	26
26	temp2 = reg2 ^ temp1;	27
27	reg1 = temp2 ^ reg3;	28
28	ADDR3 = kk; DOUT3 = reg1;	29
29	WE3 = 1;	30
30	WE3 = 0;	31
31	kk = kk + 1;	04
32	if(kk<623) goto 33; else goto 60;	33,60
33	ADDR3 = kk; RE2 = 1;	34
34	DIN2 に値を格納するために 1 クロック待つ。	35
35	reg1 = DIN2;	36
36	RE2 = 0;	37
37	ADDR3 = kk + 1; RE2 = 1;	38
38	DIN2 に値を格納するために 1 クロック待つ。	39
39	reg2 = DIN2;	40
40	RE2 = 0;	41
41	temp1 = reg1 & 0x80000000UL;	42
42	temp2 = reg2 & 0x7fffffffUL;	43

43	y = temp1 | temp2;	44
44	ADDR3 = kk - 227; RE2 = 1;	45
45	DIN2 に値を格納するために 1 クロック待つ。	46
46	reg2 = DIN2;	47
47	RE2 = 0;	48
48	mag01 = y & 0x1UL;	49
49	if(mag01==0) goto 50; else goto 51;	50
50	reg3 = 0x0UL;	51
51	if(mag01==1) goto 52; else goto 53;	52,53
52	reg3 = 0x9908b0dfUL;	53
53	temp1 = y >> 1;	54
54	temp2 = reg2 ^ temp1;	55
55	reg1 = temp2 ^ reg3;	56
56	ADDR3 = kk; DOUT3 = reg1;	57
57	WE3 = 1;	58
58	WE3 = 0;	59
59	kk = kk + 1;	32
60	ADDR3 = 623; RE2 = 1;	61
61	DIN2 に値を格納するために 1 クロック待つ。	62
62	reg1 = DIN2;	63
63	RE2 = 0;	64
64	ADDR3 = 0; RE2 = 1;	65
65	DIN2 に値を格納するために 1 クロック待つ。	66
66	reg2 = DIN2;	67
67	RE2 = 0;	68
68	temp1 = reg1 & 0x80000000UL;	69
69	temp2 = reg2 & 0x7fffffffUL;	70
70	y = temp1 | temp2;	71
71	ADDR3 = 396; RE2 = 1;	72
72	DIN2 に値を格納するために 1 クロック待つ。	73
73	reg2 = DIN2;	74
74	RE2 = 0;	75
75	mag01 = y & 0x1UL;	76
76	if(mag01==0) goto 77; else goto 78;	77,78
77	reg3 = 0x0UL;	78

78	if(mag01==1) goto 79; else goto 80;	79,80
79	reg3 = 0x9908b0dfUL;	80
80	temp1 = y >> 1;	81
81	temp2 = reg2 ^ temp1;	82
82	reg1 = temp2 ^ reg3;	83
83	ADDR3 = 623; DOUT3 = reg1;	84
84	WE3 = 1;	85
85	WE3 = 0;	86
86	mti = 0;	87
87	ADDR3 = mti; RE2 = 1;	88
88	DIN2 に値を格納するために 1 クロック待つ。	89
89	y = DIN2;	90
90	RE2 = 0;	91
91	mti = mti + 1;	92
92	temp1 = y >>11;	93
93	temp2 = y ^ temp1;	94
94	y = temp2;	95
95	temp1 = y << 7;	96
96	temp2 = temp1 & 0x9d2c5680UL;	97
97	temp3 = y ^ temp2;	98
98	y = temp3;	99
99	temp1 = y << 15;	100
100	temp2 = temp1 & 0xefc60000UL;	101
101	temp3 = y ^ temp2;	102
102	y = temp3;	103
103	temp1 = y >> 18;	104
104	temp2 = y ^ temp1;	105
105	rand = temp2;　生成された乱数を rand に格納する。	106
106	END3 = 1;　プログラムを終了する。	00
107	mti = 624;	02

状態数：108

genrand_int32 モジュールは、init_genrand モジュール、init_by_array モジュールによって初期化されたメモリの値を使って、乱数を生成するモジュールである。START3=1 に

なると処理が始まる。END3=1 になると処理が終わる。genrand_int32 の 1 回目の実行時は FIRST=1 となり、長い処理となる。2 回目からは FIRST=0 となり、簡単な処理となる。624n+1 回目の実行時は長い処理となる。生成された乱数は、rand に格納して出力される。

(4)main モジュール

main モジュールの構成は本論文の pp.8 の図 2 に示した。

main モジュールの状態遷移表を表 6 に示す。

表 6：main モジュールの状態遷移表

状態	処理	次の状態
00	START = 0; START2 = 0; START3 = 0; FIRST = 0; i = 0;	01
01	START = 1;　init_genrand を実行する。	02
02	START = 0;	03
03	if(END) goto 04; else goto 03;　init_genrand が終了すると、状態 04 へ進む。	03,04
04	START2 = 1;　init_by_array を実行する。	05
05	START2 = 0;	06
06	if(END2) goto 07; else goto 06;　 init_by_array が終了すると、状態 07 へ進む。	06,07
07	if(i<value) goto 08; else goto 14;　生成する乱数の個数だけ、乱数を生成する。	08,14
08	if(i==0) goto 09; else goto 10;　genrand_int32 の 1 回目の実行時、状態 09 へ進む。	09,10
09	FIRST = 1;　genrand_int32 の 1 回目の実行時は FIRST=1 とする。	10
10	START3 = 1;　genrand_int32 を実行する。	11
11	START3 = 0;	12
12	if(END3) goto 13; else goto 12;　genrand_int32 が終了すると、状態 13 へ進む。	12,13
13	FIRST = 0; i = i + 1;	07
14	全ての処理が終わり、状態 14 で待つ。	14

状態数：15

main モジュールは 32bit×624 ワードのメモリを持ち、init_genrand モジュール、init_by_array モジュール、genrand_int32 モジュールの順に処理を実行するモジュールである。init_genrand モジュール、init_by_array モジュールは 1 回ずつ、genrand_int32 モジュールは生成する乱数の個数だけ実行する。s に乱数の種を、value に生成する乱数の個数を入力値として与えると、rand に生成された乱数を出力する。

5.実行結果の評価と考察

5.1 評価

乱数の種を変化させて、各種ごとに 25 個の乱数を発生させた。生成される乱数の範囲は 0 から 4294967295 までである。

A、種が 19650218（デフォルトの値）のとき

3719404591 3933347911 1509182462 2945625608 2987898953
856356961 1210396626 2812813535 593820460 1666542277
3217560683 2642253538 350072595 3250968231 493759411
2490417198 166020071 2405163095 58166945 3943481778
2117096552 1327657321 3662661197 497441585 1953634444

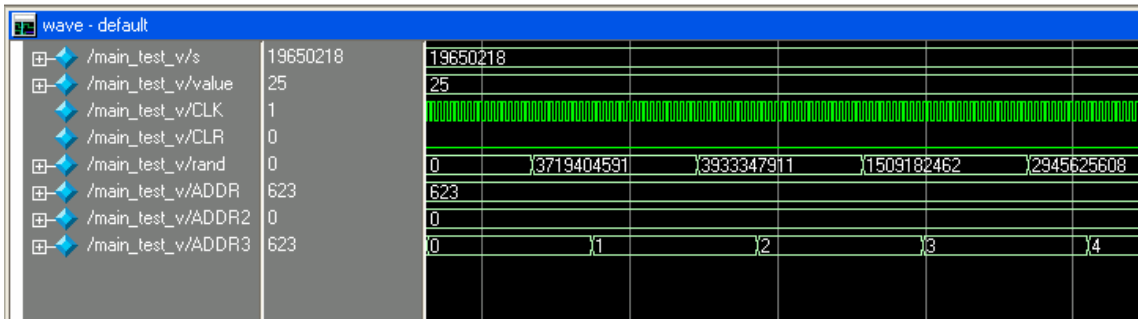

図 6：種が 19650218（デフォルトの値）のときの波形

B、種が 5489 のとき

3873800684 1976736196 2022329381 520302692 2734537017
2131974794 57083552 1426254555 3048006859 683696486
3120873552 3949251043 160467037 4054796416 3376326592
2450056627 4230119705 3347126086 10590972 936376226
3123420508 3880932014 732727278 4039099683 2195777575

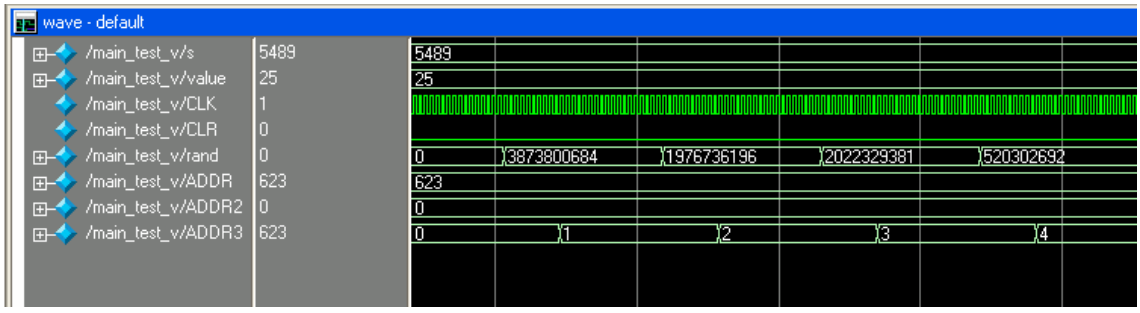

図 7：種が 5489 のときの波形

C、種が 0 のとき

4011681375　735068510 3178372001　 83836341 1249567707
1344675858　977085352 1015638510 3596196901 3392425593
4244669923 1982871838 3844575328　459019370 3092065126
4216244285　261255700 4230236973　264057851 1069332137
2273120747 1800560196 3020199887 2757054657 3358532645

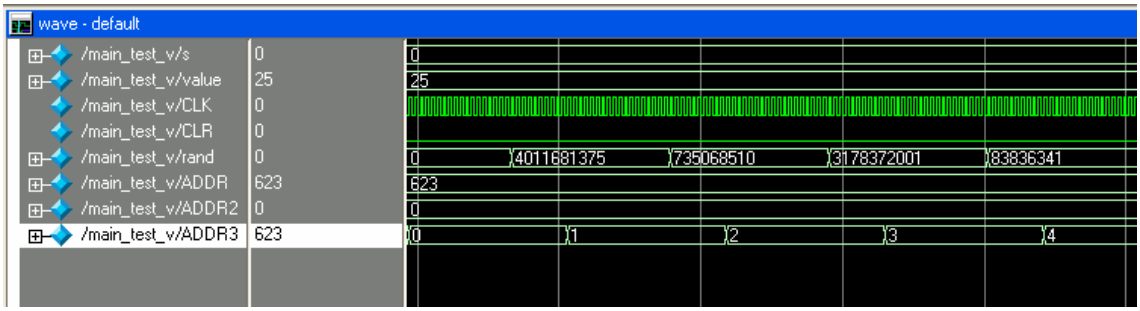

図 8：種が 0 のときの波形

D、種が 3333333333 のとき

3795680350 2849545632 2657704608 3433189321 1675155795
3173731471 1289388607 1037275824 4118733263 1374367153
1850576113　305502557 2737933495 1752096561 2133681703
 299146173 2345778161　154702915 1781665924 4193416359
2462335907　166439489 4224364926 3742014043　964576640

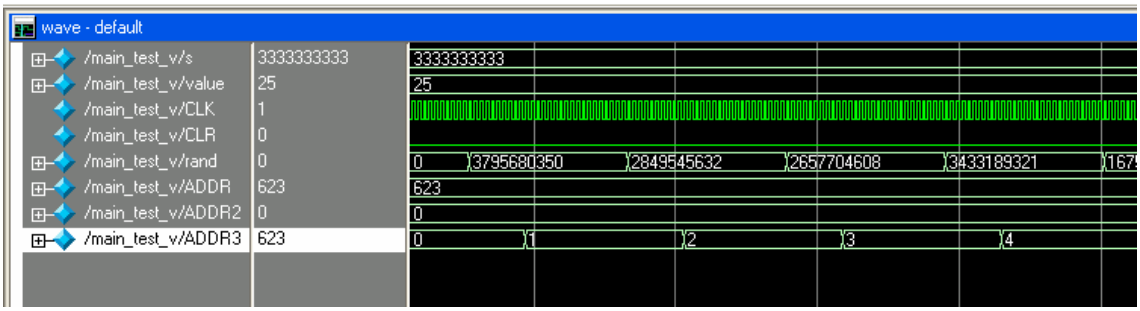

図 9：種が 3333333333 のときの波形

種を変えると、生成される乱数が変わった。計 100 個の乱数を発生させて、最小値は 10590972、最大値は 4244669923 とバランスよく乱数が生成された。また、種が 0 のときでも自然なばらつきの乱数が生成できた。1 個目の乱数が大きい値になっているのが気になるが、他の種で試したところ、小さい値も出ている。

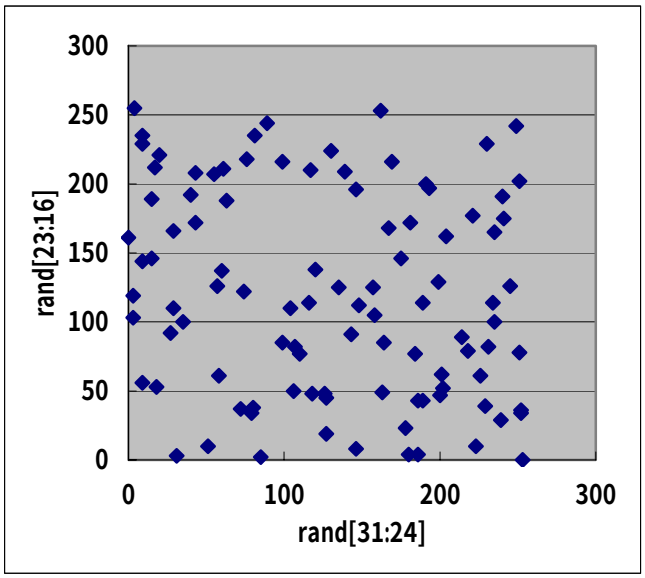

図 10：上位 16bit の乱数分布図

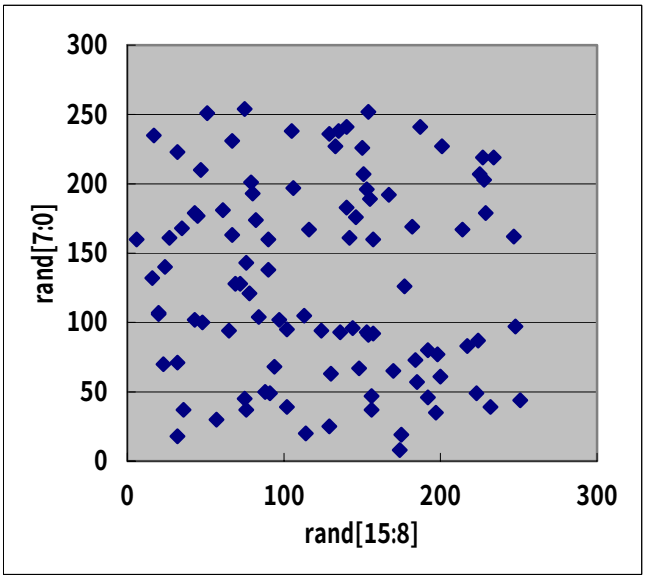

図 11：下位 16bit の乱数分布図

4 種類の種から 25 個ずつ生成された、計 100 個の乱数を 8bit ごとに分けて、分布図にした。31bit から 24bit を縦軸に、23bit から 16bit を横軸に取った分布図を図 10 に示す。15bit から 8bit を縦軸に 7bit から 0bit を横軸に取った分布図を図 11 に示す。図 10、図 11 の分布図より、上位 bit、下位 bit ともに、極端な偏りがないことがわかる。

回路規模の検証に使用したデバイスを表 7 に示す。Virtex4 を使用して、スライス数、フリップフロップ数、最大周波数の検証を行った。ハードウェア化されたメルセンヌ・ツイスタの回路規模を表 8 に示す。アルゴリズムはまったく同じでも、プログラムの改良により、スライス使用率や最大周波数が変わった。表 8 には最も性能が良かったプログラムでのデータを示した。

genrand_int32 モジュールの最大周波数が他のモジュールに比べて、2 倍以上の高い値が出ているが、原因は不明である。genrand_int32 モジュールのプログラムの量や状態数に対して、スライス数が少ないことが気になる。

main モジュールのスライス数の使用率は 10.8%で、今回使用したシステムの環境では、メルセンヌ･ツイスタ 9 個分の回路を作成可能である。フリップフロップ数の使用率は 1.6%で、まだ十分な余裕がある。最大周波数は 90.605MHz であった。

乱数生成部の波形から、1 個の乱数を生成するのに必要なクロック数が 56 であることがわかった。よって、1 個の乱数を生成するために必要な時間は、$(1／(90.605\times10^{6}))\times56$ 6.18×10^{7}=618nsである。また、パイプライン化すると、処理を 1 クロックで実行することができる。パイプライン化を行った場合、1 個の乱数を生成するために必要な時間は、$(1／(90.605\times10^{6}))$ 1.14×10^{-6} =11.4nsである。

表 7：使用デバイス

Device Family	Virtex4
Device	xc4vlx60
Package	ff668
Speed Grade	-10

表 8：ハードウェア化されたメルセンヌ・ツイスタの回路規模

モジュール	スライス数(個)	フリップフロップ数(個)	最大周波数(MHz)	スライス使用率(%)	フリップフロップ使用率(%)
init_genrand	162	188	93.027	0.6	0.4
init_by_array	400	281	90.605	1.5	0.5
genrand_int32	514	353	200.863	1.9	0.7
main	2883	848	90.605	10.8	1.6

一方、C 言語によるメルセンヌ・ツイスタのプログラムを実行した。公平にするために、ハードウェア版では使わなかった関数は削除し、アルゴリズムを同じにした。C 言語によるメルセンヌ・ツイスタの実行時間を表 9 に示す。C 言語による実行結果を図 12 に示す。メルセンヌ・ツイスタの処理の前後に、時間を出力する関数を挿入して実行し、その時間差より実行時間を求めた。

1 個の乱数を生成するために必要な時間は、乱数 1 個あたりの生成時間の平均より、約 1093ns である。

表 9：C 言語によるメルセンヌ・ツイスタの実行時間

生成した乱数の個数	1 回目(秒)	2 回目(秒)	3 回目(秒)	平均実行時間(秒)	乱数 1 個あたりの生成時間
100000	0.11	0.11	0.11	0.110	1100ns
1000000	1.05	1.10	1.04	1.063	1063ns
10000000	10.93	10.93	10.88	10.913	1091ns
100000000	109.64	109.91	115.56	111.703	1117ns

メルセンヌ・ツイスタのハードウェア化により、約 1.77 倍の高速化ができた。
さらに、パイプライン化すると、約 95.88 倍の速度向上が得られる。

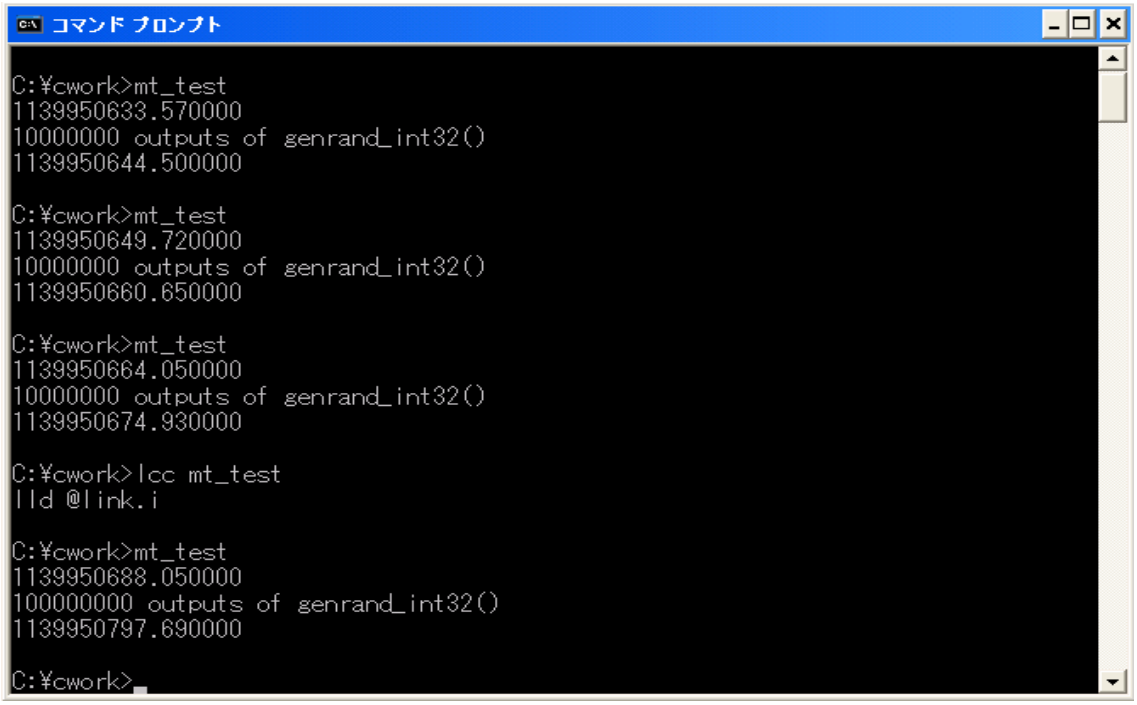

図 12：ソフトウェア版メルセンヌ・ツイスタの実行結果

5.2 考察

ハードウェア化されたメルセンヌ・ツイスタの規模を見ると、スライス数は 2883 で、使用率は 10.8%であった。全体で 800 行を超える、大きなプログラムなので、スライス数、使用率は納得のいく結果である。フリップフロップ数は 868 で、使用率は 1.6%であった。ハードウェア化されたメルセンヌ・ツイスタはクロックにより動作する回路で、フリップフロップは多く使用していると思ったが、意外に使用率は低かった。そして、最大周波数は 90.605MHz であった。最大周波数が大きくなれば、乱数生成速度が向上するので、最大周波数は最も重要な要素である。最大周波数は、プログラムの改良を行っても、他の要素と比べてあまり変化が見られなかったので、最大周波数は限界に近い値が出ていると思われる。乱数生成速度を向上させるには、クロック数を抑えると効果的である。ハードウェア化されたメルセンヌ・ツイスタの回路では、乱数を生成するまでのクロック数が約 95000 である。そして、1 個の乱数を生成するのに必要なクロック数が 56 である。これは Verilog-HDL のプログラムを改良すれば、半分以下のクロック数に抑えることが可能だと思われる。クロック数を抑えるには、状態数を出来る限り抑えることが必要である。クロック数を抑えると、乱数生成速度が飛躍的に向上する。

乱数のばらつきについては理想的である。どの桁を見ても、目立つ偏りは見当たらない。種が 0 であっても、他の種と同等のばらつきの乱数を生成することができた。生成された乱数はアルゴリズムにより処理を行っているだけなので、擬似乱数となる。同じ種からは同じ値の乱数列があらわれた。真の乱数を得るには、外部回路が必要で、外部回路で得られた値を種に代入すると、真の乱数が得られる。

6.おわりに

本研究では乱数発生回路のメルセンヌ・ツイスタを設計し、Xilinx 社の提供する ISE 7.1i を用いて実装した。メルセンヌ・ツイスタの検証はメンター・グラフィックス社の提供する ModelSim XE Ⅲ 6.0a を用いて行った。

本研究を行ったことにより、メルセンヌ・ツイスタの乱数生成速度が、C 言語で実行した場合と比較して、約 1.77 倍の向上が得られた。また、パイプライン化すると、約 95.88 倍の速度向上が得られることを確認した。

今後の課題として、Verilog-HDL で記述したプログラムを改良して、さらに乱数生成速度を向上させることが挙げられる。そして、メルセンヌ・ツイスタを FPGA ボードの上に実装することが挙げられる。次に、剰余回路を作成して、乱数の範囲を指定して出力することができる回路を作成し、乱数の実用化を目指す。また、雑音などでランダムな値の種を得て、擬似ではない真の乱数を発生させることが挙げられる。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導をいただきました山崎勝弘教授、小柳滋教授に深く感謝いたします。

また、本研究に関して貴重なご意見をいただきました、中谷氏、難波氏、及びいろいろな面で貴重な助言や励ましを下さった高性能計算研究室の皆様に心より感謝いたします。

参考文献

[1]David A.Patterson/John L.Hennessy:コンピュータの構成と設計(上),日経 BP 社,1999.
[2]David A.Patterson/John L.Hennessy:コンピュータの構成と設計(下),日経 BP 社,1999.
[3]堀 桂太郎:図解 ModelSim 実習,森北出版,2005.
[4]田丸 啓吉:論理回路の基礎(改訂版),工学図書,1989.
[5]月刊 C MAGAZINE 12 月号 Vol.17 pp.108-115,ソフトバンククリエイティブ,2005
[6]MersenneTwisterHomePage
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html
[7]ザイリンクス ホームページ http://www.xilinx.co.jp/
[8]Verilog-HDL による回路設計 http://www.tokuma.org/WebPack/