Mersenne Twister Pseudo Random Number Generator

Overview

Introduction

この IP は Mersenne Twister 法による疑似乱数生成回路です。
こちらを参考に書いてみました。Mersenne Twister HomePage
(http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html)
mt19937ar を元にしています。

Features

・Mersenne Twister 法による疑似乱数生成回路です。
・状態テーブルの数(N)は 624 です。
・VHDL で記述しています。
・論理合成可能です。Xilinx 社の Vivado、Altera 社の QuartusII で確認済み。
・１クロックで 1、2、3、8、16 ワード(1 ワードは 32bit)の乱数を生成します。
・ジェネリック変数で SEED 値を設定できます。
・状態テーブルを書き換えることが可能です。

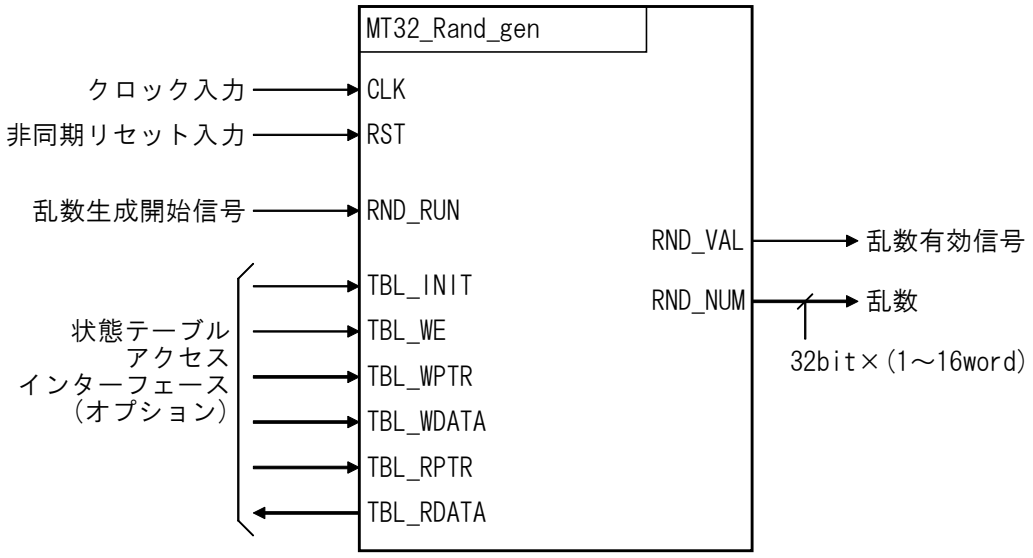

Fig.1 Top-Level Signaling Interface

Licensing

二条項 BSD ライセンス (2-clause BSD license) で公開しています。

Specification

Parameter Descriptions

Table.1 Parameter Descriptions

Name	TYPE	Default	Description
L	Integer	1	1 クロックで生成する乱数の数を指定します このパラメータで指定できる数は、1、2、4、8、16 のいずれかです
SEED	Integer	123	乱数のシード値を指定します このシード値を元に状態テーブルが初期化されます

Port Descriptions

Table.2 Port Descriptions

Name	Type	Width	I/O	Description
CLK	STD_LOGIC	1	in	クロック信号
RST	STD_LOGIC	1	in	非同期リセット信号 注)状態テーブルの内容はリセットしません
RND_RUN	STD_LOGIC	1	in	乱数生成開始信号 この信号が'1'になってから 3 クロック後に乱数を出力します TBL_INIT が'1'の時、この信号を'1'にしてはいけません
RND_VAL	STD_LOGIC	1	out	乱数有効信号 RND_NUM より生成された乱数が有効であることをしめす信号 RND_RUN が'1'になってから 3 クロック後に'1'になります
RND_NUM	STD_LOGIC_VECTOR	32*L	out	乱数出力信号 生成された乱数を出力する信号 RND_RUN が'1'になってから 3 クロック後に乱数を出力します

TBL_INIT	STD_LOGIC	1	in	状態テーブル・初期化信号 状態テーブルを初期化することを示します この信号が'1'の時のみ、TBL_*信号は有効です この信号を'1'にすると、内部のカウンタがリセットされます
TBL_WE	STD_LOGIC_VECTOR	1*L	in	状態テーブル・ライト信号 状態テーブルへのライトを示す信号 ライトはワード(32bit)単位で行います
TBL_WPTR	STD_LOGIC_VECTOR	16	in	状態テーブル・ライトアドレス 状態テーブルへのライトアドレスをワード(32bit)単位で示します 例えば、LANE=2 の場合下位 1 ビットは無視されます
TBL_WDATA	STD_LOGIC_VECTOR	32*L	in	状態テーブル・ライトデータ 状態テーブルへのライトデータを LSB で入力します
TBL_RPTR	STD_LOGIC_VECTOR	16	in	状態テーブル・リードアドレス 状態テーブルからのリードアドレスをワード(32bit)単位で示します 例えば、LANE=2 の場合下位 1 ビットは無視されます
TBL_RDATA	STD_LOGIC_VECTOR	32*L	out	状態テーブル・リードデータ 状態テーブルからのリードデータ TBL_RPTR の入力に対して 1 クロック後に TBL_RPTR で示したアドレスの値を出力します

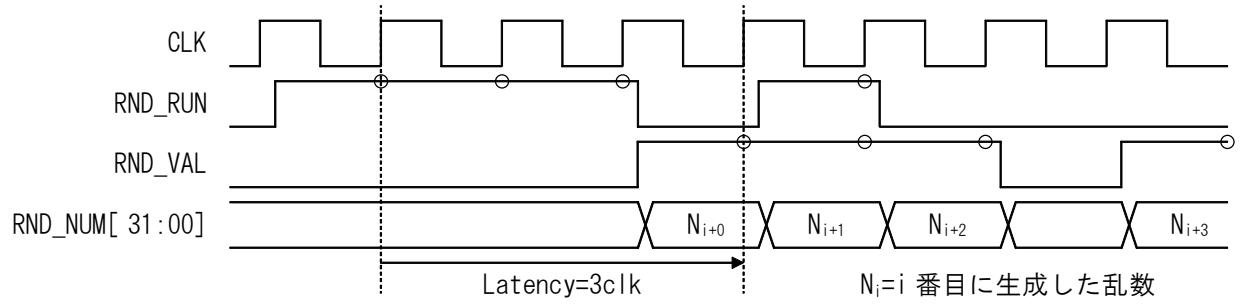

Fig.2 Generate Timing (L=1)

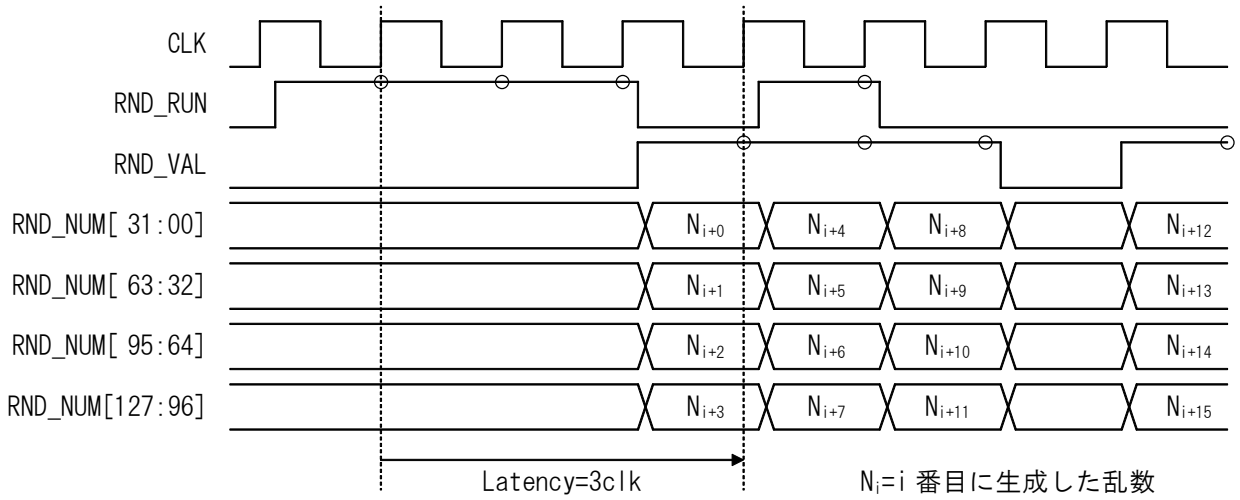

Fig.3 Generate Timing (L=4)

Resouces and Performance

Xilinx 社の FPGA で実装してみた結果を以下に示します。

Parameter の RAM は、内部メモリに BRAM(Block RAM)を使うか、LUT を使うかを指定しています。Performance の生成速度は 1 秒間に生成できるワード数(1 ワードは 32bit)の理論値です。

Table.3 Resouces and Performance(Xilinx)

Device		Parameter		Resouces		Performance	
Family	Speed	L	RAM	Slices	RAMB	Fmax	Generate word/sec
Artix-7	3	1	BRAM	112	2	250[MHz]	250[Mword/sec]
			LUT	1815	0	250[MHz]	250[Mword/sec]
		2	BRAM	221	4	250[MHz]	500[Mword/sec]
			LUT	1911	0	250[MHz]	500[Mword/sec]
		4	BRAM	443	8	250[MHz]	1000[Mword/sec]
			LUT	2101	0	250[MHz]	1000[Mword/sec]
		8	BRAM	894	16	250[MHz]	2000[Mword/sec]
			LUT	2784	0	250[MHz]	2000[Mword/sec]
		16	BRAM	1844	32	238[MHz]	3808[Mword/sec]
			LUT	3098	0	250[MHz]	4000[Mword/sec]

Architecture

Block Diagram

下図は L=1 の時のブロック図です。

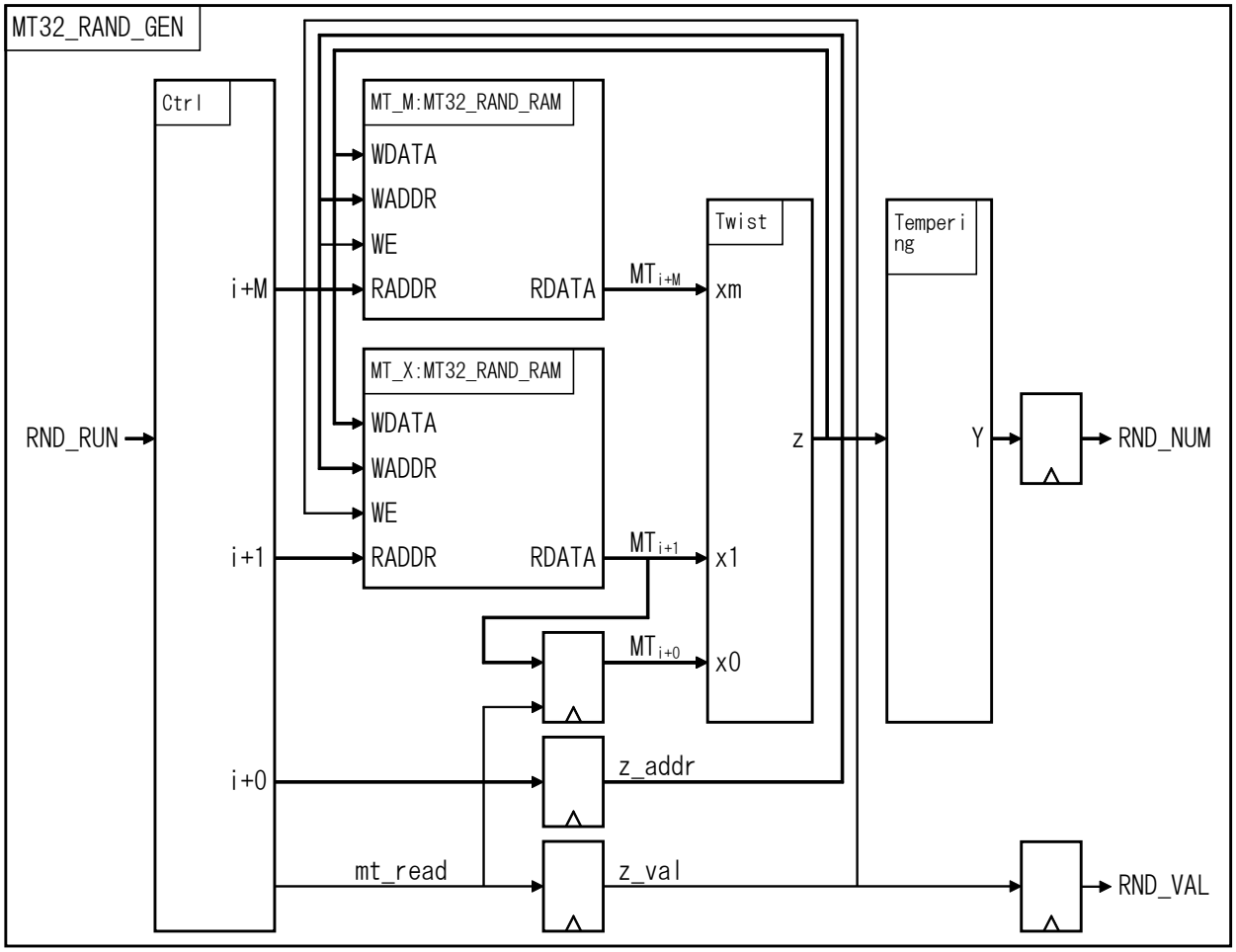

Fig.4 Block Diagram(L=1)

RAM の構成(L=1 の場合)

MT32_Rand_Gen は 1 クロックで 1 ～L の乱数を生成します。ところが Mersenne Twister のアルゴリズムでは、１つの乱数を生成するためには次のように状態テーブル(mt)の i、(i+1) mod N、(i+M) mod N の位置の値が必要です。

mt19937ar.vhd

```
    procedure generate_word(
        variable generator  : inout PSEUDO_RANDOM_NUMBER_GENERATOR_TYPE;
        variable result     : out   RANDOM_NUMBER_TYPE
    ) is
        alias    mt         :       RANDOM_NUMBER_VECTOR(0 to generator.table'length-1)
is generator.table;
        variable i          :       integer range mt'low to mt'high;
        variable x0,x1,xm   :       RANDOM_NUMBER_TYPE;
        variable y          :       RANDOM_NUMBER_TYPE;
        variable z          :       RANDOM_NUMBER_TYPE;
        constant mag01      :       RANDOM_NUMBER_VECTOR(0 to 1) := (0 => X"00000000", 1
=> MATRIX_A);
    begin
        i  := generator.index;
        x0 := mt(i);
        x1 := mt((i+1) mod mt'length);
        xm := mt((i+M) mod mt'length);
        y  := (x0 and UPPER_MASK) or (x1 and LOWER_MASK);
        z  := xm xor (y srl 1) xor mag01(to_integer(y mod mag01'length));
        mt(i) := z;
        generator.index := (i+1) mod mt'length;
        y  := z;
        y  := y xor ((y srl 11));
        y  := y xor ((y sll  7) and X"9d2c5680");
        y  := y xor ((y sll 15) and X"efc60000");
        y  := y xor ((y srl 18));
        result := y;
    end procedure;
```

１クロックで乱数を生成するには、状態テーブルから３つのアドレスのデータを同時に読み出す必要があります。通常ならライト１ポート、リード３ポートの RAM が必要ですが、基本的に i はインクリメントされるため、一つ前の乱数生成時に使用した MT[i+1]をレジスタに保存しておき、次の乱数生成時に MT[i]として使用すれば、ライト１ポート、リード２ポートの RAM で実装することが出来ます。MT32_Rand_Gen では、ライト１ポート、リード２ポートの RAM はライト１ポート、リード１ポートの RAM を二つ並べることで実装しています。

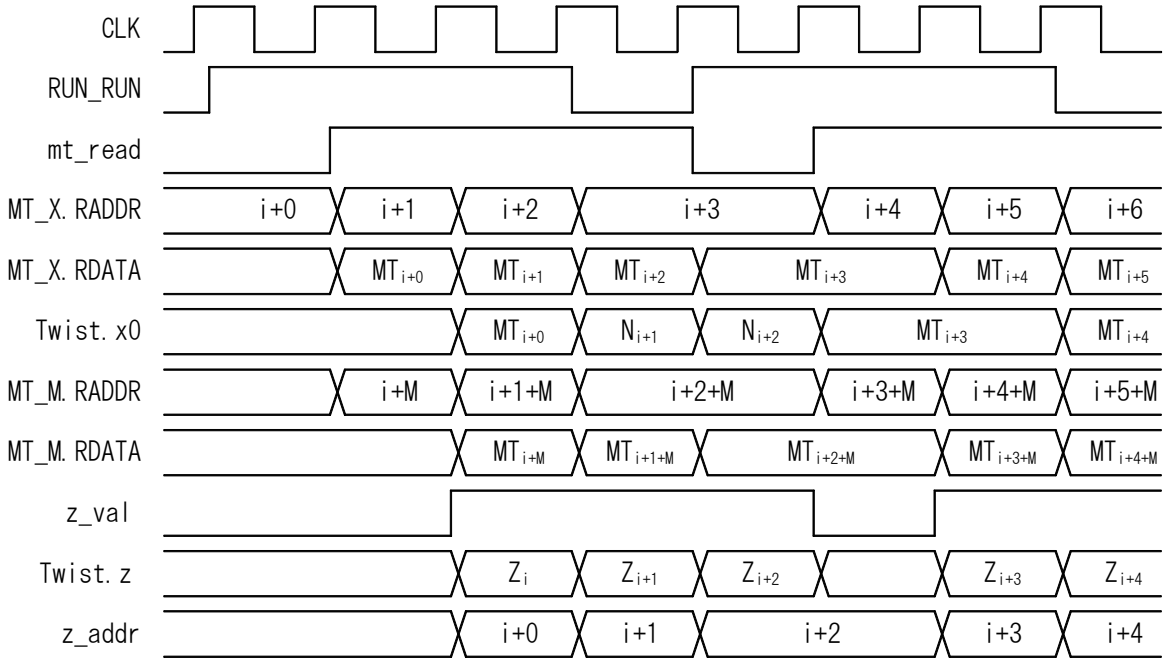

Fig.5 RAM Read and Twist Timing Chart (L=1)

RAM の構成(L>1 の場合)

MT32_Rand_Gen では L に 2、4、8、16 を指定することで、1 クロックでそれぞれ 2 ワード、4 ワード、8 ワード、16 ワードの乱数を同時に生成することが出来ます。

そのためには、例えば L=4 の場合、次の 9 箇所の状態テーブルの値を 1 クロックで読む必要があります。

1. i
2. (i+1) mod N
3. (i+2) mod N
4. (i+3) mod N
5. (i+4) mod N
6. (i+0+M) mod N
7. (i+1+M) mod N
8. (i+2+M) mod N
9. (i+3+M) mod N

MT32_Rand_gen では RAM の構成を工夫することで、状態テーブルの値を同時に読んでいます。

まずは、MT[i]、MT[(i+1) mod N]、MT[(i+2) mod N]、MT[(i+3) mod N]、MT[(i+4) mod N]の RAM の構成を次図に示します。

L=1 の場合と同様に、MT[(i+4) mod N] の値をレジスタに保持しておき、次の MT[i]として使用します。

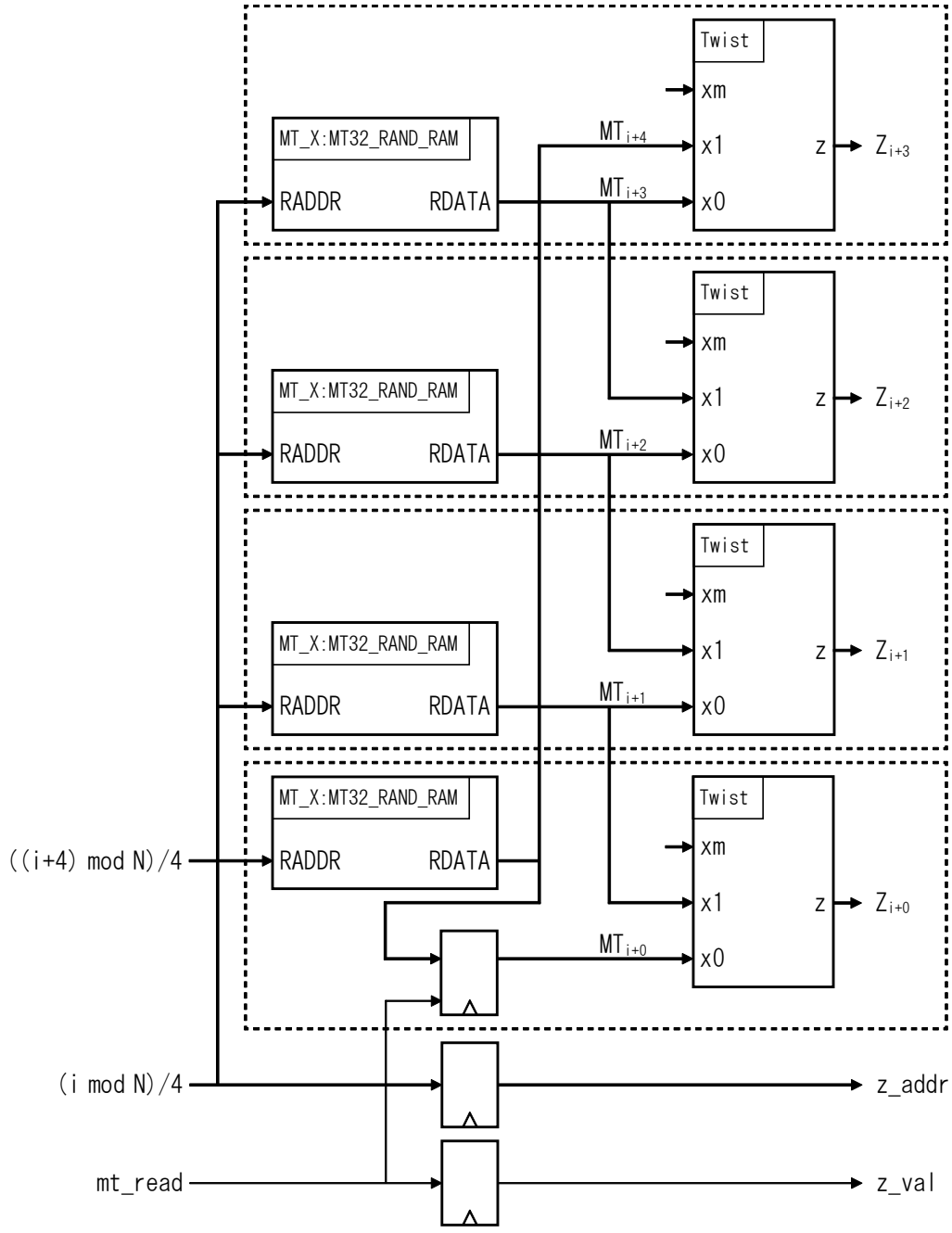

Fig.6 Block Diagram(L=4)-1

次に MT[(i+0+M) mod N]、MT[(i+1+M) mod N]、MT[(i+2+M) mod N]、MT[(i+3+M) mod N]の RAM 周りの構成を次図に示します。

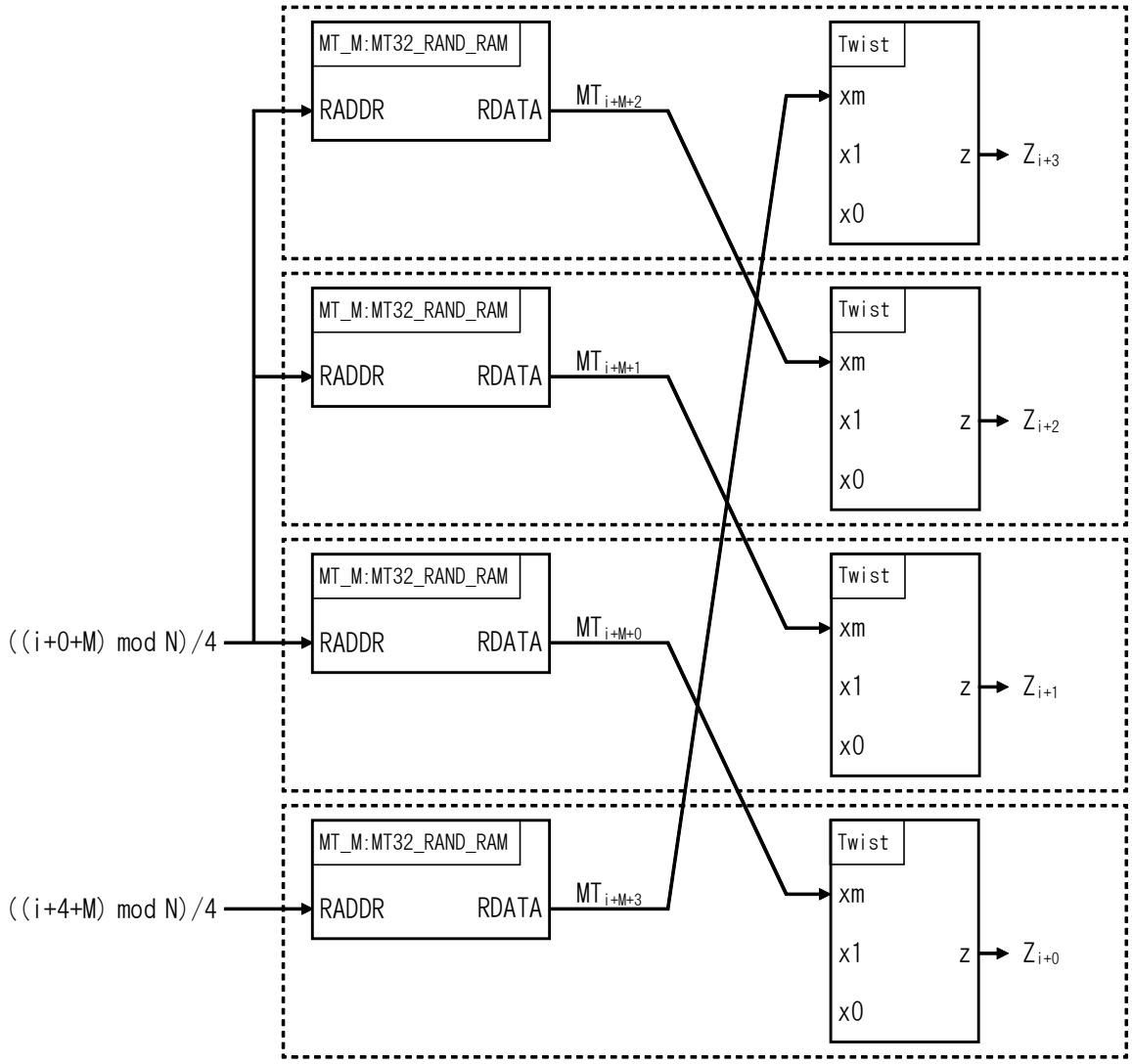

Fig.7 Block Diagram(L=4)-2

注意点は、MT32_Rand_Gen では M=397 なので、例えば MT[i+0+M]は Z[i+1]を書き込んだ RAM に格納されていることです。そのため、各 RAM の出力は 1 だけローテーションして各 Twist の xm に接続する必要があります。

RAM のアドレス

RAM のアドレスのタイプは次のように定義しています。

mt32_rand_gen.vhd

```
    function  CALC_MT_PTR_LOW return integer is
        variable retval : integer;
    begin
        retval := 0;
        while (2**retval < L) loop
            retval := retval + 1;
        end loop;
        return retval;
    end function;
    function  CALC_MT_PTR_HIGH return integer is
        variable retval : integer;
    begin
        retval := 0;
        while (2**(retval+1) < N) loop
            retval := retval + 1;
        end loop;
        return retval;
    end function;
    constant  MT_PTR_LOW           : integer := CALC_MT_PTR_LOW;
    constant  MT_PTR_HIGH          : integer := CALC_MT_PTR_HIGH;
    subtype   MT_PTR_TYPE         is std_logic_vector(MT_PTR_HIGH downto MT_PTR_LOW);
```

また i mod N などいちいち剰余を計算するのは面倒なので、i は常にインクリメントされることを利用して、次のようにしています。

mt32_rand_gen.vhd

```
    function  TO_MT_PTR(arg:integer) return MT_PTR_TYPE is
        variable u : unsigned(MT_PTR_HIGH downto 0);
    begin
        u := to_unsigned(arg,u'length);
        return std_logic_vector(u(MT_PTR_TYPE'range));
    end function;
    function  INC_MT_PTR(ptr:MT_PTR_TYPE) return MT_PTR_TYPE is
        variable retval : MT_PTR_TYPE;
    begin
        if (unsigned(ptr) >= unsigned(TO_MT_PTR(N-1))) then
            retval := (others => '0');
        else
            retval := std_logic_vector(unsigned(ptr)+1);
        end if;
        return retval;
    end function;
```

さらに次のように各 RAM へのアドレスを生成しています。

ここで各信号の意味は次の通り。

- x_curr_index = ((i) mod N)/L
- x_next_index = ((i+L) mod N)/L
- m_curr_index = ((i+M) mod N)/L
- m_next_index = ((i+L+M) mod N)/L

mt32_rand_gen.vhd

```
CTRL: process(CLK,RST) begin
       if (RST = '1') then
               x_curr_index <= TO_MT_PTR(0);
               x_next_index <= TO_MT_PTR(0);
               m_curr_index <= TO_MT_PTR(M);
               m_next_index <= TO_MT_PTR(M);
               z_curr_index <= TO_MT_PTR(0);
               mt_read      <= '0';
               z_val        <= '0';
       elsif (CLK'event and CLK = '1') then
           if    (TBL_INIT='1') then
               x_curr_index <= TO_MT_PTR(0);
               x_next_index <= TO_MT_PTR(0);
               m_curr_index <= TO_MT_PTR(M);
               m_next_index <= TO_MT_PTR(M);
               z_curr_index <= TO_MT_PTR(0);
               mt_read      <= '0';
               z_val        <= '0';
           else
               if (RND_RUN = '1')then
                   x_curr_index <= x_next_index;
                   x_next_index <= INC_MT_PTR(x_next_index);
                   m_curr_index <= m_next_index;
                   m_next_index <= INC_MT_PTR(m_next_index);
                   mt_read      <= '1';
               else
                   mt_read      <= '0';
               end if;
               if (mt_read = '1') then
                   z_curr_index <= x_curr_index;
                   z_val        <= '1';
               else
                   z_val        <= '0';
               end if;
           end if;
       end if;
   end process;
```

Simulation

GHDL によるシミュレーション

GHDL のバージョン

GHDL のバージョンは 0.29 です。
GHDL のホームページはこちら(http://ghdl.free.fr)

Makefile

シミュレーション用に Makefile を用意しています。

Makefile

```
GHDL=ghdl
GHDLFLAGS=--mb-comments
WORK=work

TEST_BENCH = test_bench ¥
             $(END_LIST)

all: $(TEST_BENCH)

clean:
        rm -f *.o *.cf $(TEST_BENCH)

test_bench: mt19937ar.o test_bench.o mt32_rand_gen.o mt32_rand_ram.o mt32_rand_ram_auto.o
         $(GHDL) -e $(GHDLFLAGS) $@
        -$(GHDL) -r $(GHDLRUNFLAGS) $@

test_bench.o    :  ../../src/test/vhdl/test_bench.vhd mt32_rand_gen.o
        $(GHDL) -a $(GHDLFLAGS) --work=work $<

mt32_rand_gen.o :  ../../src/main/vhdl/mt32_rand_gen.vhd
        $(GHDL) -a $(GHDLFLAGS) --work=mt32_rand_gen $<

mt32_rand_ram.o:  ../../src/main/vhdl/mt32_rand_ram.vhd
        $(GHDL) -a $(GHDLFLAGS) --work=mt32_rand_gen $<

mt32_rand_ram_auto.o:  ../../src/main/vhdl/mt32_rand_ram_auto.vhd mt32_rand_ram.o
        $(GHDL) -a $(GHDLFLAGS) --work=mt32_rand_gen $<

mt19937ar.o    :  ../../src/main/vhdl/mt19937ar.vhd
        $(GHDL) -a $(GHDLFLAGS) --work=mt32_rand_gen $<
```

シミュレーションの実行

```
$ cd sim/ghdl
$ cd make
ghdl -a --mb-comments --work=mt32_rand_gen ../../src/main/vhdl/mt19937ar.vhd
ghdl -a --mb-comments --work=mt32_rand_gen ../../src/main/vhdl/mt32_rand_gen.vhd
ghdl -a --mb-comments --work=work ../../src/test/vhdl/test_bench.vhd
ghdl -a --mb-comments --work=mt32_rand_gen ../../src/main/vhdl/mt32_rand_ram.vhd
ghdl -a --mb-comments --work=mt32_rand_gen ../../src/main/vhdl/mt32_rand_ram_auto.vhd
ghdl -e --mb-comments test_bench
ghdl -r  test_bench
 check prgn_1.table
 1000 outputs of genrand_int32()
2991312382 3062119789 1228959102 1840268610  974319580
2967327842 2367878886 3088727057 3090095699 2109339754
1817228411 3350193721 4212350166 1764906721 2941321312
  :
  (中略)
  :
../../src/test/vhdl/test_bench.vhd:297:9:@51680ns:(assertion failure):   Run complete...
./test_bench:error: assertion failed
./test_bench:error: simulation failed
ghdl: compilation error
```

最後にエラーが出てるように見えますが、これは assert(FALSE)でシミュレーションを終了しているためです。

Vivado によるシミュレーション

Vivado のバージョン

Vivado 2015.1

Vivado プロジェクトの作成

すでにプロジェクトを作っている場合は省略してください。
プロジェクトを生成するための Tcl スクリプトを用意しています。

sim/vivado/create_project.tcl

上記の Tcl スクリプトを Vivado で実行するとプロジェクトが作成されます。

Vivado > Tools > Run Tcl Script > create_project.tcl

シミュレーションを実行

Vivado > Open Project > mt32_rand_gen.xpr
Flow Navigator > Run Simulation > Run Behavioral Simulation

Synthesis and Implementation

Vivado による論理合成

Vivado のバージョン

Vivado 2015.1

Vivado プロジェクトの作成

すでにプロジェクトを作っている場合は省略してください。
プロジェクトを生成するための Tcl スクリプトを用意しています。

fpga/xilinx/vivado2015.1/mt32_rand_gen/create_project.tcl

上記の Tcl スクリプトを Vivado で実行するとプロジェクトが作成されます。

Vivado > Tools > Run Tcl Script > create_project.tcl

デバイスはとりあえず xc7a15tcsg324-3 を指定しますが、変更したい場合は、create_project.tcl を修正してください。

Synthesis

Flow Navigator > Run Synthesis

Implementation

Flow Navigator > Run Implementation

QuartusII による論理合成

QuartusII のバージョン

QuartusII 13.1sp1

QuartusII 用のプロジェクト

fpga/altera/13.1sp1/mt32_rand_gen.qpf

Acknowledgments

このような貴重なアルゴリズムを惜しげもなく公開してくださった方々にはひたすら感謝です。