卒　業　論　文

ハード／ソフト・コデザインによる
乱数発生と差集合巡回符号エラー訂正

氏　　名：牧 岡　　幸 一
学籍番号：　　2210980212-7
指導教員：山崎　勝弘　教授
提 出 日：2002 年 2 月 18 日

立命館大学理工学部情報学科

内容梗概

近年のＩＣ，ＬＳＩチップの集積度の急速な向上により、大規模なチップの設計が困難となってきた。そこでハード／ソフト・コデザインという手法が考案された。この手法は大規模かつ複雑なシステムの設計をハードウェアとソフトェアの両面から考慮した上で設計を進めていくというものである。こういったハードウェアとソフトウェアのトレード・オフを考慮することが考えられるようになった背景には、ＦＰＧＡのようなチップの登場により、システム開発する環境が整えやすくなってきたことがあげられる。

本研究では、ハードウェアとソフトウェアのトレードについて考慮するにあたり、ある特定の機能に特化したシステムの設計を行い、それぞれのコストや速度に焦点を当てたメリット／デメリットについて検証することを目的とする。

ハードウェア設計を行うにあたり、設計手順として TopDown 設計を用いた。この設計手法は仕様を決め、HDL により記述する。そして論理合成と配置配線を行った後、実機にダウンロードするというものである。そして実際にM系列による乱数発生と差集合巡回符号エラー訂正の回路を作成し、実機にて動作検証する。またソフトウェアの検証も行う必要があるため、本研究ではＣＰＵに KUE-CHIP2 プロセッサ相当のチップを使用した。そして自作のアセンブリプログラムをこのチップにて動作させ、その実行時の波形を観測する。

本研究では、ハードウェアとソフトウェアの両方で同じ機能を実現し、そのために必要とされたロジック数，最大動作周波数，一連の工程を実行するためにかかる時間などについて考察する。

目次

図目次

表目次

1 はじめに

ここわずか２，３年で一般家庭にパソコンが急激に浸透してきた。この背景にはWindowsなど初心者にもわかり易く使いやすいＯＳの進化と共に、それを支えるハードウェアの高速化があげられると考えられる。パソコンに使われているＣＰＵなどは家庭に浸透してきたここ数年の間に、1000万トランジスタにも満たなかったＬＳＩが4000万を突破するほどに集積度が急激に上がると同時に、動作周波数も200MHzから2GHz以上と急激に成長した。また命令セットの追加などによって仮に同クロックで動作したとしても、より高速化が図れるようにもなってきた。しかしその反面、パソコンなどのコンピュータに使用されているＣＰＵをはじめとするＬＳＩは消費電力が急激に増し、そして発熱量も確実に無視できない存在となってきた。つい９０年代初頭にはＣＰＵにはヒートシンクのみで対応できていたものが、最近ではファンを付けていても注意が必要となってきている。

しかしノートパソコンなどでは膨大な発熱があっても、それを排熱するだけの設備を押し込むわけにはいかず、システム自体の存続が危険であるため、低消費電力や低発熱なＬＳＩチップの登場が待たれている。デスクトップパソコンにおいても、最近では低消費電力や低騒音というのは機能のひとつとして注目されるようになってきた。このように、近年では高速化の一途をたどるだけが高性能とはいえないという時代になってきている。情報を処理する早さももちろん性能の指標だが、同時に少ない電力で正確にそして静かな排熱機構で処理するというのも性能を測る指標として確立してきている。

またパソコンのような汎用機だけではなく、携帯電話や家庭用電化製品に組み込まれているマイコンなどでも同様のことが考えられる。消費電力を低く抑えることができれば、携帯電話やモバイル機器などはバッテリーによる長時間使用も可能となり、またより高機能な処理を可能とすることができるようになってきている。こういったモバイル機器はコスト削減も大きな課題となっているため、大規模な集積回路を使用するわけにもいかない。しかしソフトウェアに頼りすぎるとあまり高度なことや高速な処理は望めないというように微妙な相互関係となっている。

今後もまだまだ同様の課題には研究の余地があるが、こういった課題に対応していくためにはより開発環境が身近になる必要があるとともに、即座に回路としての検証が行えることが望ましい。そしてどのような処理をハードウェアにより実現し、どのような処理ならソフトウェアによる処理でも問題ないのかという点を見つける必要がある。そのためにハードウェアとソフトウェア、双方からの開発が必須となる。そういった設計手法を「ハード／ソフト・コデザイン」と呼ぶ。

本研究はハード／ソフト・コデザインの基礎的な実験として、M系列乱数発生と差

集合巡回符号エラー訂正を対象として、ハード，ソフトそれぞれ同様のアルゴリズムを用いることにより機能を実現し、ハードとソフトのトレードについて考察する。またハードウェアで実現する場合とソフトウェアで実現する場合のコスト（トランジスタ数），および速度について検証することを目的とする。ソフトウェアに関してはＫＵＥ－ＣＨＩＰ２マイクロプロセッサのアセンブリ言語により実現する。

またコンピュータがこれほど人間の生活に浸透してきていると、そのコンピュータの計算ミスによる影響は絶大なものである。そのためコンピュータには信頼性が強く求められる。しかしデータを転送した際、ノイズなどにより信号を誤って受信してしまうことがある。それによりこの転送したデータが使い物にならないようでは、通信は実用的であるとはいえない。しかし有線・無線を問わず、なんらかのエラー訂正技術を使うことによりその信頼性を向上させている。もっとも単純で一般的であるエラー発見技術にパリティビットを付加することによりビットエラーを発見する、パリティチェックというものがある。これはある一定量のデータの各ビットをすべて排他的論理和することによって得られた１ビットを最後に付加するというものである。これによって、受信したときにエラーを発見することができるというものである。しかし１ビット増えてしまうということと、パリティビットもエラーとなり得る、またエラーを検出できてもそれを修復する機能はないことがデメリットとなる。また偶数ビットがエラーとなっていた場合はパリティビットによるエラー検出は不可能である。

そこでエラーを検出しても、再送信ではなく実際に転送するビット数をある程度増やすだけで、受信後エラーを検出・修復することができるという技術が必須となる。この差集合巡回符号エラー訂正もそういったことを可能とする技術の１つである。本研究はこの技術を使って、転送したい１１ビットのデータに１０ビットのエラー訂正用ビットを付加して、２１ビットを１セットとして転送する。これにより通信時のエラー訂正について検証する。

設計はＶＨＤＬで記述する。そして論理合成しネットリストを生成し、配置配線ツールでピンの割付を行い、ＨＤＬ－ＢＯＡＲＤⅢにダウンロードする。そして実機によって動作させ、ＭＵ２００－ＶＩＫＴ－ＩＦで波形観測する。

本論文では、まず第２章で開発環境について述べる。第３章ではハードウェアとソフトウェア両方によって実現したM系列法による乱数発生の工程について述べる。そして第４章では差集合巡回符号エラー訂正回路について述べる。最後の第５章では総まとめとして本論文の考察と、今後の課題について述べる。

2 ハード／ソフト・コデザインにおける開発環境

2.1 ハードウェア記述言語

ハードウェア記述言語(HDL: Hardware Description Language)とは、プログラミングをするかのような記述方式を用いることにより回路を比較的簡単に設計できるようにするための言語である。これにより設計者は論理素子をひとつひとつ並べて回路を設計する必要がなくなる。そのため、かなりの労力を削減することができると同時に、回路設計期間を大幅に短縮することができる。

この言語が用いられるようになった背景には、「ムーアの法則」に沿ったかたちでマイクロチップの微細化が急速に進んだということがあげられる。これはチップの集積度が増すということだが、回路そのものは人間が設計する必要がある。しかしより集積度が増すということは人間の労力も増えるということにつながる。また設計する回路が複雑になればなるほど人間による設計は効率が悪くなると同時に、ミスがでるようになる。そのため必ずしも効率のいいチップが設計できるとは限らないし、相当の期間を要することとなる。

このハードウェア記述言語には、VHDL(Very High Speed Integrated Circuit HDL)や Verilog-HDL などがある。この他には AHDL(Altera HDL)など対応チップや機能などに特化したものはいくつかあったが前者２つが一般的に普及している。本研究では VHDL により回路設計を行っている。

本研究の回路設計の工程としてはトップダウン設計をとっている。（図１参照）トップダウン設計とは仕様を決定の後、HDL によって記述し、それをまず機能シミュレータにかける。そして機能レベルで仕様を満たしていた場合、それを論理合成ツールにかける。そしてゲートレベルによるシミュレータにかけ、遅延なども考慮に入れた上で問題ない場合はそれで完成となる。しかし本研究では FPGA(Field Programmable Gate Array)チップを使用した実機による動作検証を行っているため、図１における点線以下の部分も検証している。

本研究で使用した各工程でのツールを次に記載しておく。

機能シミュレーション：ModelSim（MentorGraphics 社）
論理合成：FPGAExpress（Synopsys 社）
：Leonardo Spectrum（Exemplar 社）
FPGA レイアウト：Max＋PlusⅡ（ALTERA 社）
実機：HDL-BOARDⅢと MU200-VIKT-IF（凸版印刷株式会社）

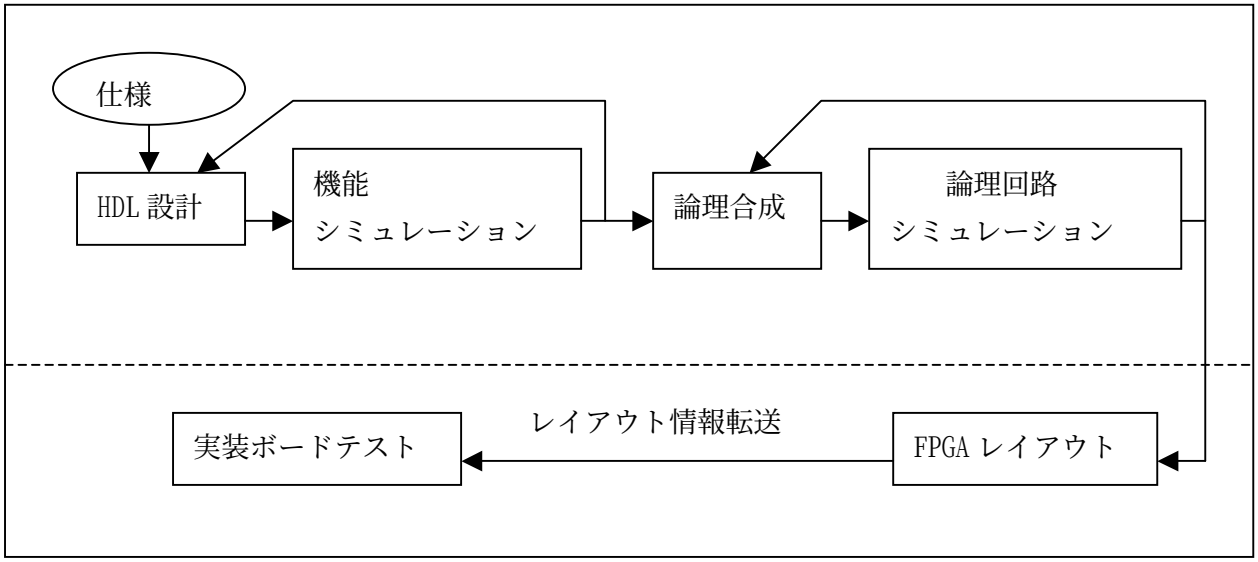

図 1：トップダウン設計

2.2 実機ボードと観測用ボード

本研究の実機動作検証において、HDL-BOARDⅢ[14]と MU200-VIKT-IF[7]の２枚を専用コネクタにより接続したボードを使用した。HDL-BOARDⅢは凸版印刷株式会社エレクトロニクス事業部により開発されたボードであり、FPGA チップを２つ搭載している。そしてＰＣ上の MaxPlus2 によりケーブルを通じて、ＦＰＧＡレイアウト情報をダウンロードし、回路を実機に実現することができる。

MU200-VIKT-IF は三菱電機マイコン機器ソフトウェア株式会社により開発された観測用ボードであり、専用ツールである VIKT-Reflector(for Windows)とセットで使用する。これは前述した HDL-BOARDⅢ上の FPGA チップの外部に出ているピン情報を実際に観測することが可能である。

それぞれのボードの仕様は次の表１の通りであり実機の写真を図２に記載する。

表 1：実機の仕様

HDL-BOARDⅢ		MU200-VIKT-IF	
FPGA チップ	FLEX10K(EPF10K10QC208-4)×２	チップ	EPF10K100ARC240-3
ROM	M27C256	クロック	1～20MHz（ボード上、またはソフトウェアから設定可）
入力装置	PUSH ボタン６コ(RESET,SS を含む)		
観測用表示装置	LCD モジュール（１６桁×１行）	その他	観測用ボードであるため、ＰＣ上に観測データを転送して波形観測が可能
クロック	78.125k～20MHz までをロータリーＳＷにより切替可		
図２において赤線より下		図２において赤線より上	

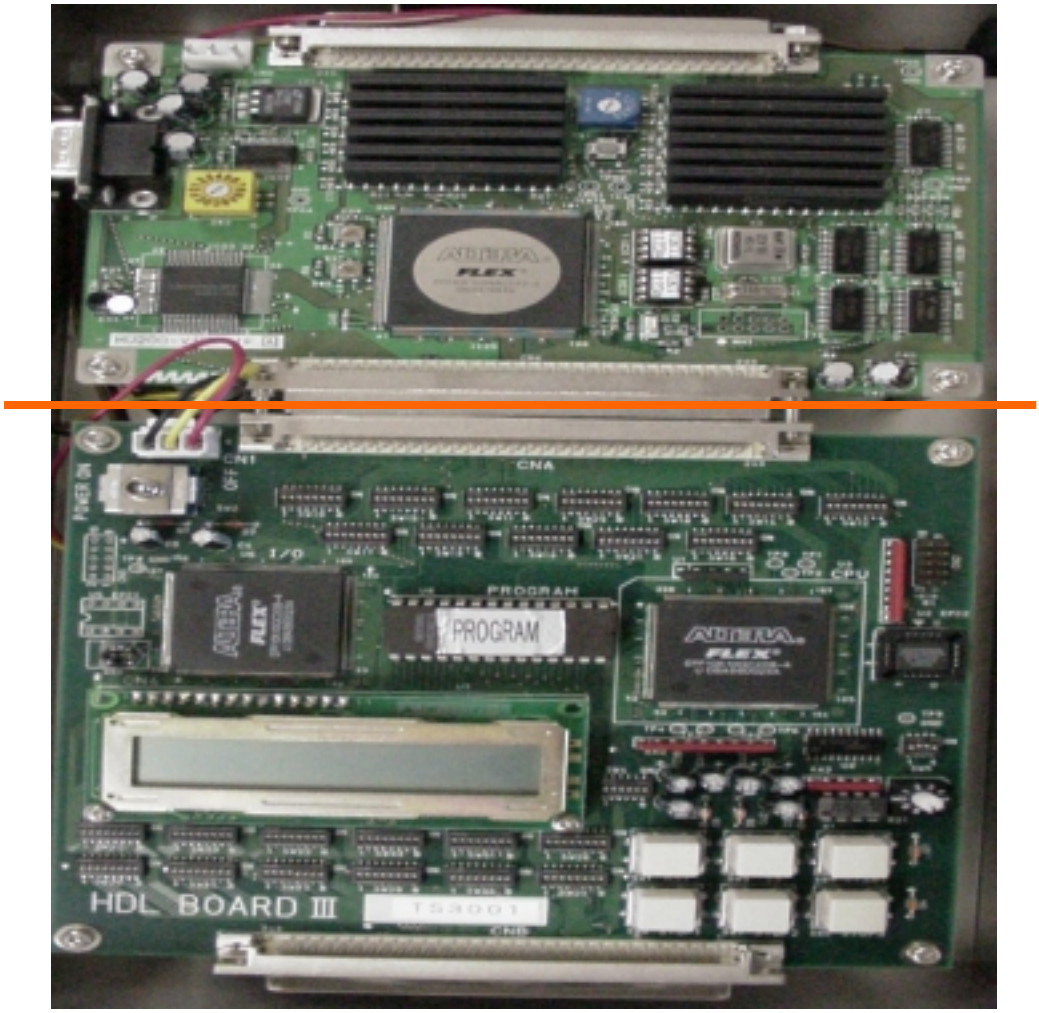

図 2：実機ボードと観測用ボード

2.3 各種ツールと観測ツール

2.3.1 機能シミュレータ

本研究ではModelSimを使用した。このツールはＨＤＬにより記述された回路設計ファイルと、テストベンチファイルを入力としシミュレーションするものである。特徴としては、ＶＨＤＬ，Ｖｅｒｉｌｏｇ－ＨＤＬ両方に対応しているだけでなく、両方が混在した回路設計データであってもシミュレート可能である。また数百万ゲートのＡＳＩＣやＦＰＧＡなど幅広いニーズに対応することができる。

2.3.2 論理合成ツール

本研究ではFPGA Expressと Leonardo Spectrumを使用した。前者はSynopsys社を開発元とする論理合成ツールである。HDLファイルを入力とし、ネットリスト（ＥＤＩＦ形式）を出力する。本研究では、凸版印刷からの「TopDown設計Training Kit」に沿ってTopDown設計を体験する工程において使用した。後者はExemplar社が開発したツールであり、簡単でわずらわしい手順を踏まなくても論理合成が行えるQuickモードと、上級者向けに細かな設定もできるAdvancedモードが選択できる。ネットリストを出力するだけではなく、配置配線ツールから実遅延情報を取得し、その情報からスタティックタイミング解析を行うことができる。このように配置配線ツールと相互にデータをやり取りすることで、論理合成ツールと配置配線ツールとの間で生じるタイミングについての差を、早く収束させることができます。この機能はTimeCloser テクノロジと呼ばれ、このツールの特徴となっている。

2.3.3 配置配線ツール

本研究ではMax+PlusⅡを使用した。これはＡＬＴＥＲＡ社製のＦＰＧＡチップにＦＰＧＡレイアウト情報をダウンロードするための配置配線ツールである。HDL-BOARDⅢとByteBlasterという専用ケーブルで接続する。そしてネットリストを入力データとし、実機に搭載されているＦＰＧＡチップにフィットした配置配線をする。この際入出力ピンは設定可能である。

2.3.4 ＦＰＧＡチップ観測ツール

本研究では VIKT-Reflector を使用した。これは PC と MU200-VIKT-IF をシリアルケーブルによって接続し、MU200-VIKT-IF を操作, 使用するための波形観測ツールである。観測モードとしてはロジアナモードとリアルタイムモードが使用可能である。ロジアナモードは実際に取得した波形情報をまずボード上のバッファに格納し、観測停止となるかトリガ条件が成立した時点でバッファリングを終了し、PC 上の Reflector に取得情報を転送する。またリアルタイムモードは動作している FPGA チップのピン波形情報をリアルタイムで取得し、Reflector にリアルタイム表示していくためクロックは 128Hz を上限としている。

2.4 ＫＵＥ－ＣＨＩＰ２教育用マイクロプロセッサ

KUE-CHIP2(Kyoto University Education Chip2)は、京都大学, 京都高度技術研究所（ＡＳＴＥＭ）, および本学で開発された教育用の８ビット・マイクロプロセッサである。また本学理工学部情報学科の情報学実験では実際に使用されている教育用ボードである。命令は１９種, 命令数は４０個, アドレス指定方式は４種類のプロセッサである。この KUE-CHIP2 の命令セットを表２に示す[14]。

表 2：KUE-CHIP2 命令表

CALL	01010---	◎	サブルーチンコール
RET	01011---	×	サブルーチンリターン
OUT	00010---	×	※使用不可
IN	00011---	×	※使用不可
RCF	00100---	×	0→CF
SCF	00101---	×	1→CF
Bcc	0011 cc	◎	条件が成立すればB'→PC
Ssm	0100A0sm	×	(A)→shift, rotate→A
Rsm	0100A1sm	×	はみだしたビット→CF
LD	0110 AB	○	(B)→A
ST	0111 AB	◎	(A)→B
SBC	1000 AB	○	(A)-(B)-CF→A
ADC	1001 AB	○	(A)+(B)+CF→A
SUB	1010 AB	○	(A)-(B)→A
ADD	1011 AB	○	(A)+(B)→A
EOR	1100 AB	○	(A)xor(B)→A
OR	1101 AB	○	(A)or(B)→A
AND	1110 AB	○	(A)and(B)→A
CMP	1111 AB	○	(A)-(B)

B'(2語目) ×:不要, ○:不要or必要, ◎:必要

Shift Mode(sm)表			
RL	10	RA	00
LL	11	LA	01

NZ	0001	≠0 ZF=0
Z	1001	=0 ZF=1
ZP	0010	≧0 NF=0
NZ	1010	<0 NF=1
P	0011	>0 (NF∨ZF)=0
ZN	1011	≦0(NF∨ZF)=0
NI	0100	※使用不可
NO	1100	※使用不可
NC	0101	CF=0
C	1101	CF=1
GE	0110	≧0 (VF xor NF)=0
LT	1110	<0 (VF xor NF)=1
GT	0111	>0 ((VF xor NF)∨ZF)=0
LE	1111	≦0 ((VF xor NF)∨ZF)=1

A=0: ACC,A=1:IX		
B=000:ACC		
B=001:IX		
B=01-:d	即値アドレス	
B=100:[d]	絶対アドレス	プログラム領域
B=101:(d)	絶対アドレス	データ領域
B=110:[IX+d]	インデックス修飾アドレス	プログラム領域
B=111:(IX+d)	インデックス修飾アドレス	データ領域

3 ハードウェア／ソフトウェアによる乱数発生

3.1 M系列乱数発生アルゴリズム

０または１のC_kを係数とする漸化式

$$x_n \equiv c_1 x_{n-1} + c_2 x_{n-2} + \cdots + c_p x_{n-p} \pmod 2$$

によって０と１からなる数列$\{x_n\}$をつくる。ただし$C_p = 1$とする。この漸化式の特性多項式$f(x) = 1 + c_1 x + \cdots + c_p x^p$が有限体ＧＦ（２）上の原始多項式のとき、$\{x_n\}$をM系列という。一定の刻みで$\{x_n\}$を標本化することにより、区間(0,1)上の２進法ｐ桁の一様分布列を構成できる。このアルゴリズムによりM系列法は成り立っている。M系列法は、１次元だけでなく多次元の一様分布の乱数発生法としても優れている[15]。

3.2 ハードウェアとソフトウェアによる乱数発生の比較

乱数発生の手順としては、まずReg[7]とReg[4]の排他的論理和を次の乱数の最下位ビットとする。それと同時にクロックの立ち上がりごとに、レジスタの値を１ビット左シフトする。このように１つ前の数を利用して、乱数を生成していく。本研究では８ビットの乱数を生成する。これはハードウェアもソフトウェアも同様の工程とする。

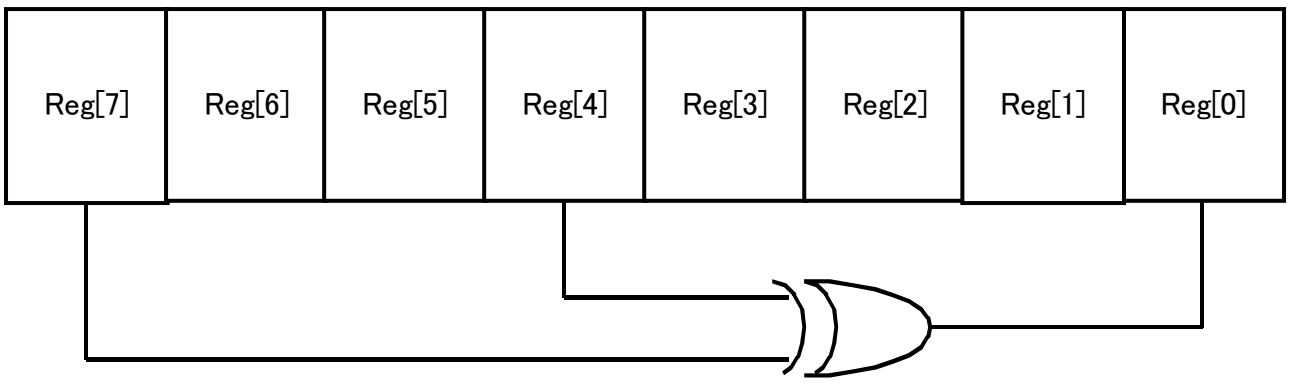

図 3：M系列乱数発生

実機にダウンロードしてハードウェア／ソフトウェア共に波形観測を行った。ハードウェアの波形観測結果を図４に示し、ソフトウェアによる観測結果を図５に示す。

論理合成ツールによってＦＰＧＡチップ(EPF10K10QC208-4)での最高動作周波数とコストが確認できる。また実機による波形観測結果である図４と図５より乱数の発生間隔についても確認することができた。これを表３に示し、論理合成ツールより得られた各回路のコストを表４に示す。

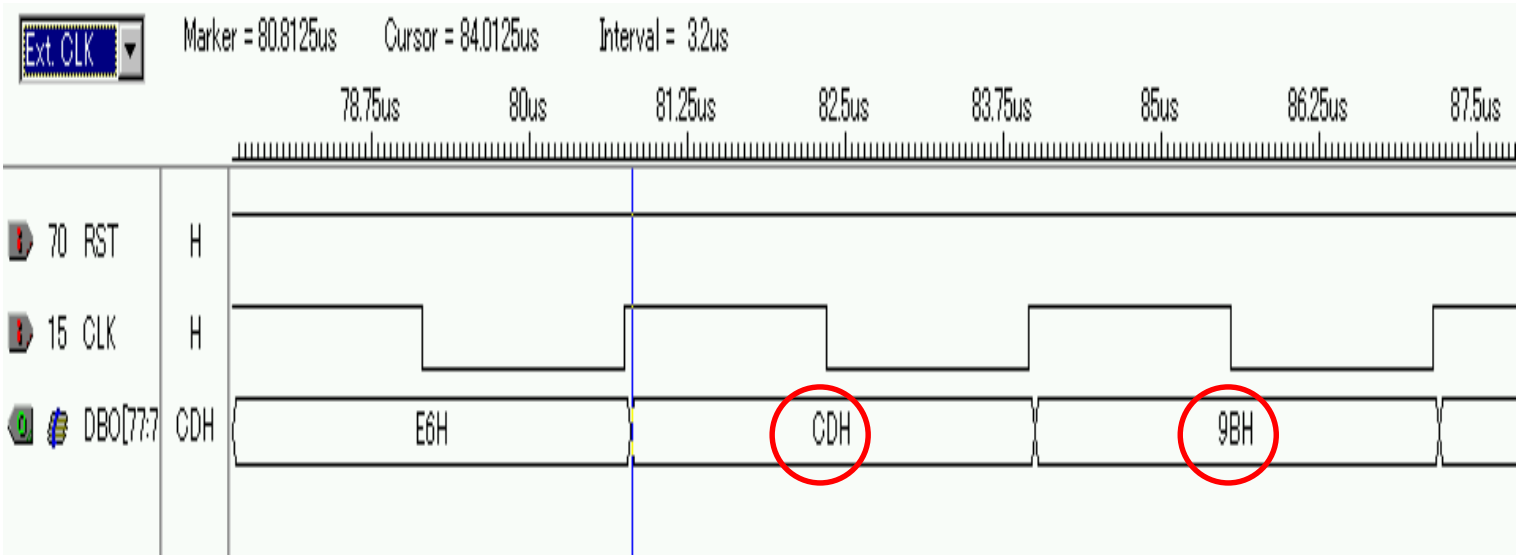

図 4：ハードウェアによる乱数発生の観測結果

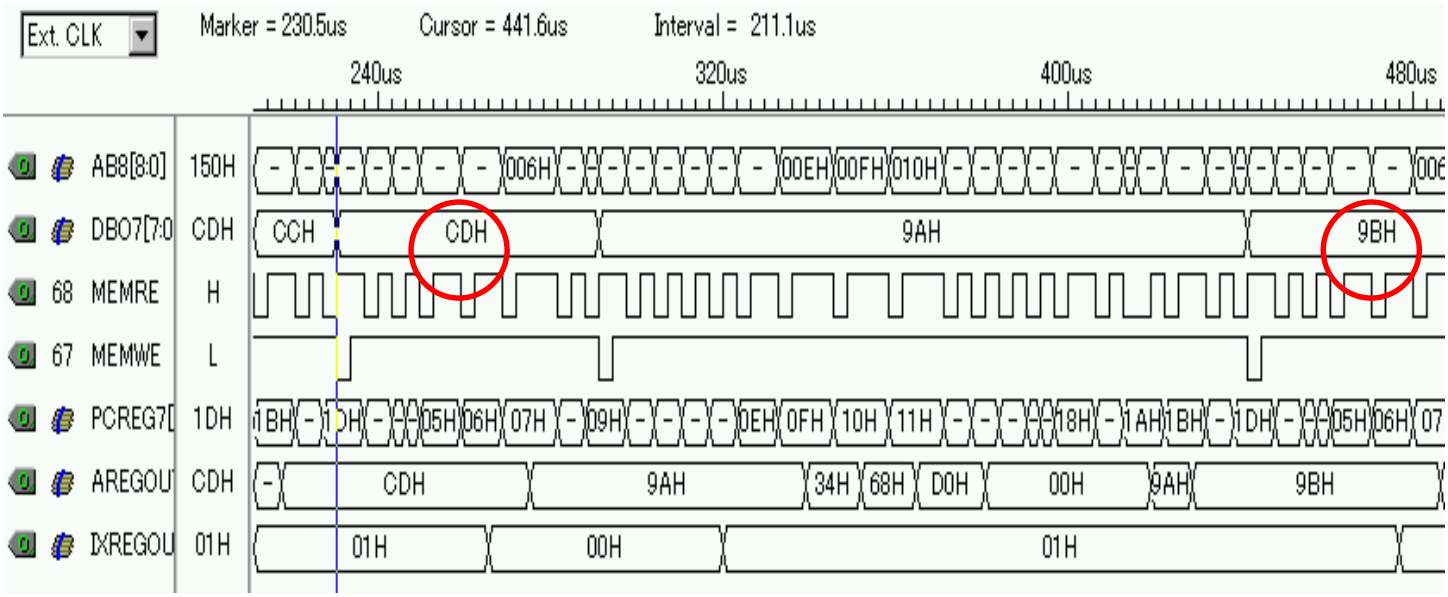

図 5：ソフトウェアによる乱数発生の観測結果

表 3：性能比較

	ハードウェア	ソフトウェア
最高動作周波数[MHz]	102.9	20
乱数発生間隔[us]	3.2	211.1

表 4：コスト比較

		ハードウェア KUE-CHIP2	
コスト	D Flipflops	16	97
	Logic Cells	16	371

3.3 考察

表３より乱数発生に特化したハードウェアでは、まず最高動作周波数が５倍ほど速いことがわかる。また乱数自体の発生間隔では 70 倍もの違いがある。この動作周波数と乱数発生間隔の倍率が圧倒的に違っている原因としては、ハードウェアで発生する場合は１クロックにつき、１つずつ乱数を生成できる。それに対してソフトウェアで発生する場合は、数十クロックで一連の乱数発生プログラムを実現しているためだと考えられる。またハードウェア量に関してもプロセッサである KUE-CHIP2 と比較して、かなり小さいことがわかる。

次にこの回路をプロセッサに追加機能として実装することについて考察する。この回路をプロセッサに搭載することにより、そのプロセッサはクロックごとに乱数を生成することができるため、乱数を生成することに処理を奪われることがなくなる。しかしその分この回路を搭載するということはコストが増加してしまうということになる。近年のコンピュータ用ＣＰＵなどに使用されているＬＳＩやＶＬＳＩでは大規模な回路を使用しているため、本研究で作成した乱数発生回路なら実装してもチップの肥大化による心配はないと思われる。そのため汎用チップであるならば実装する利点はあるものと思われる。

しかし例えば KUE-CHIP2 のような小規模なプロセッサであると無駄なチップの肥大化ということになりかねない。KUE-CHIP2 にこの乱数発生回路を搭載したら、

16÷(371+16)　0.04

つまり約４％占有することとなる。これは高機能よりも理解性・コスト性を重視する KUE-CHIP2 では大きな占有量となってしまう可能性がある。

4 差集合巡回符号エラー訂正回路の設計

本研究は、Design Wave Magazine Nov.2001 に掲載されていた和田知久琉球大学教授による差集合巡回符号エラー訂正回路設計仕様書[8]を参考にして、この差集合巡回符号エラー訂正の受信機を作成したものである。送信機は既に作成され掲載されていたものを使用した。

4.1 差集合巡回符号エラー訂正のアルゴリズム

4.1.1 送信データ（データ＋エラー訂正用）生成アルゴリズム

転送したい 11 ビットの情報ビットを{u0, u1, u2, u3, u4, u5, u6, u7, u8, u9, u10}とし、以下の多項式 U(ｘ)で表す。

$$U(x) = u_0 + u_1 \cdot x^1 + u_2 \cdot x^2 + u_3 \cdot x^3 + u_4 \cdot x^4 + u_5 \cdot x^5 + u_6 \cdot x^6 + u_7 \cdot x^7 + u_8 \cdot x^8 + u_9 \cdot x^9 + u_{10} \cdot x^{10}$$

---- (1)

そうすると、21 ビットの送信ビットは以下のような多項式 SB(x)で示すことができる。

$$SB(x) = u_0 \cdot x^{10} + u_1 \cdot x^{11} + u_2 \cdot x^{12} + u_3 \cdot x^{13} + u_4 \cdot x^{14} + u_5 \cdot x^{15} + u_6 \cdot x^{16} + u_7 \cdot x^{17} + u_8 \cdot x^{18} + u_9 \cdot x^{19} + u_{10} \cdot x^{20} + r_0 + r_1 \cdot x^1 + r_2 \cdot x^2 + r_3 \cdot x^3 + r_4 \cdot x^4 + r_5 \cdot x^5 + r_6 \cdot x^6 + r_7 \cdot x^7 + r_8 \cdot x^8 + r_9 \cdot x^9$$

----(2)

これを表で示すと、以下の表５のようになる。

表 5：情報ビット（１１ビット）と送信ビット（２１ビット）

	エラー訂正用（10 ビット）										情報ビット（11 ビット）										
情報ビット（１１ビット）											u0	u1	u2	u3	u4	u5	u6	u7	u8	u9	u10
送信ビット（２１ビット）	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	u0	u1	u2	u3	u4	u5	u6	u7	u8	u9	u10

すなわち、

$$R(x) = r_0 + r_1 \cdot x^1 + r_2 \cdot x^2 + r_3 \cdot x^3 + r_4 \cdot x^4 + r_5 \cdot x^5 + r_6 \cdot x^6 + r_7 \cdot x^7 + r_8 \cdot x^8 + r_9 \cdot x^9$$

----(3)

を求めれば、U(x)が与えられているので、容易にSB(ｘ)を生成することが可能となる。実はR(x)は

$$X(x) = u_0 \cdot x^{10} + u_1 \cdot x^{11} + u_2 \cdot x^{12} + u_3 \cdot x^{13} + u_4 \cdot x^{14} + u_5 \cdot x^{15} + u_6 \cdot x^{16} + u_7 \cdot x^{17} + u_8 \cdot x^{18} + u_9 \cdot x^{19} + u_{10} \cdot x^{20}$$

----(4)

を

$$G(x) = 1 + x^2 + x^4 + x^6 + x^7 + x^{10} \quad \text{----(5)}$$

で割り算を行った余りとなる。

（５）式の次数は１０次なので、（４）式を（５）式で割った余りは９次式以下の次数となる。この多項式の余りを求める方法は、

$$x^{10} = 1 + x^2 + x^4 + x^6 + x^7 \quad \text{----(6)}$$

を用いて、１０次の項を７次以下の式への変換を繰り返し用いることで、（４）式の次数を９次以下に下げるように変換することで求めることができる。但しこのとき、係数同士の加算や減算はEXORで代用する。よって、

$$x^2 + x^2 = (1 \oplus 1) \cdot x^2 = 0 \cdot x^2 = 0$$
$$x^2 + x^2 + x^2 = (1 \oplus 1 \oplus 1) \cdot x^2 = 1 \cdot x^2 = x^2 \quad \text{----(7)}$$

のように計算します。

4.1.2 エラー訂正アルゴリズム

送信したいデータ１１ビットを｛1111 1111 111｝とすると、実際に送信されるデータは｛0011 0010 0011 1111 1111 1｝となる。このデータから

表６に示すような５種類のＡ１～Ａ５のパリティチェックを考える。

表 6：送信ビット（２１ビット）の例

ビットの位置番号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
送信ビット（SB）	0	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1
パリティチェック A1										1			1	1					1		1
パリティチェック A2		1										1			1	1					1
パリティチェック A3					1		1										1			1	1
パリティチェック A4	1					1		1										1			1
パリティチェック A5			1	1					1		1										1

パリティチェック A1 ではビット番号で 9，12，13，18，20 のところに１がある。この５つの位置に対応する送信ビット SB(9)，SB(12)，SB(13)，SB(18)，SB(20)の XOR を取ると、’０’となる。同様に A2～A5 でもすべて’０’となる。つまり表７のようなかたちでチェックしている。

表 7：５種類のパリティチェック

パリティチェック A1	A1 = SB (9) xor SB(12) xor SB(13) xor SB(18) xor SB(20) = 0
パリティチェック A2	A2 = SB (1) xor SB(11) xor SB(14) xor SB(15) xor SB(20) = 0
パリティチェック A3	A3 = SB (4) xor SB(6) xor SB(16) xor SB(19) xor SB(20) = 0
パリティチェック A4	A4 = SB (0) xor SB(5) xor SB(7) xor SB(17) xor SB(20) = 0
パリティチェック A5	A5 = SB (2) xor SB(3) xor SB(8) xor SB(10) xor SB(20) = 0

ここで送信ビットの 20 ビット目 SB(20)がエラーで反転した場合について考える。SB(20)は 5 種類のパリティチェック式にすべて含まれている。したがって、５つのパリティチェック式 A1 から A5 はすべて’１’となるはずである。

また、送信ビットの 20 ビット目以外で 1 箇所のエラーが発生した場合について考える。表６を見てわかるように、送信ビットの 20 ビット目以外のビットは５種類のパリティチェック式のどれかに 1 回のみ使用されている。そのため、A1 から A5 のうちどれか１つが’1’となるはずである。したがって、上記パリティチェック式を計算し、その多数が’1’となると SB(20)にエラーが発生していることが検知され、SB(20)のビットを反転することでエラーの訂正を行うことができるようになる。

では SB(20)以外のビットのエラー訂正について、表８に SB(19)を訂正するためのパリティチェックの方法を示す。これは単に表６の SB(20)を訂正するパリティチェックを示す’1’を全体に左に 1 ビットシフトしたものとなっている。そして、一番左端であふれた’1’を右端に巡回させています。このようにシフトしたパリティチェックを用いれば SB(19)ビットを同様に訂正することができる。

同様にパリティチェック式をシフトすることで、他のビット位置のエラーを訂正することが可能となる。

表 8：SB(19)を訂正するパリティチェック

ビットの位置番号	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
送信ビット（SB）	0	0	1	1	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1
パリティチェック A1									1			1	1					1		1	
パリティチェック A2	1										1			1	1					1	
パリティチェック A3				1		1										1			1	1	
パリティチェック A4					1		1										1			1	1
パリティチェック A5		1	1					1		1										1	

4.2 送信回路

この差集合巡回符号エラー訂正回路は、シリアルで１１ビットの情報ビットを入力（ＩＮ）とし、それに１０ビットのエラー訂正用の信号を付加して、２１ビットで１セットとした送信データをシリアルで出力する回路である。この回路図を図６に示す。

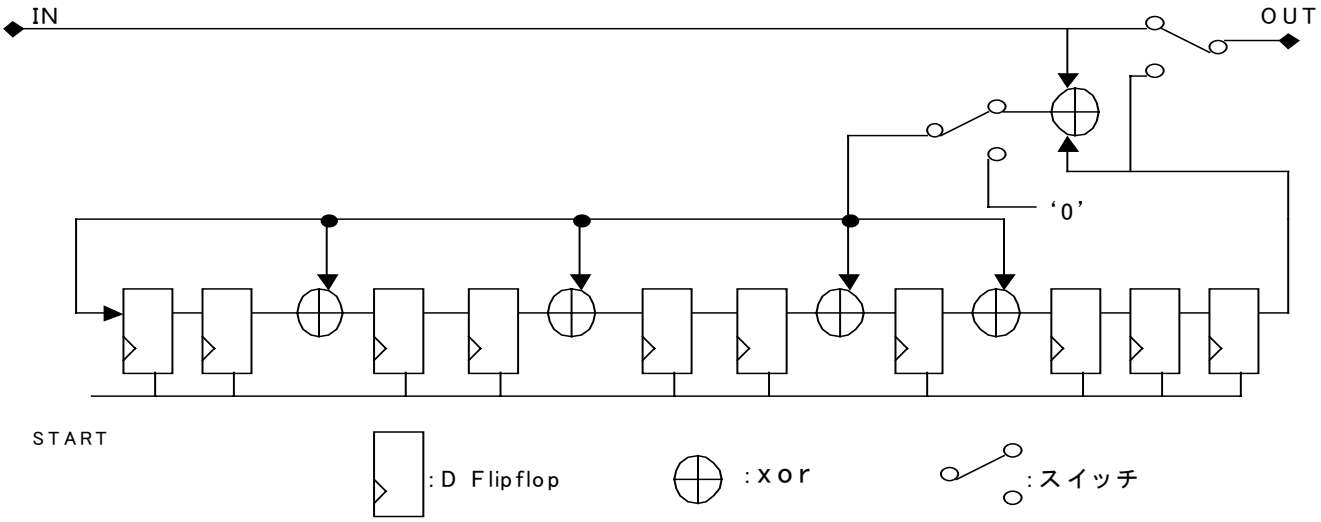

図 6：エラー訂正送信機

4.3 受信回路

送信機より転送されてきた２１ビットのシリアルデータを入力とし、これを入力と同時にレジスタにシフトすることにより格納する。そして出力時にエラー訂正を行った上で１１ビットの元データを出力するという回路である。この回路図を図７に示す。

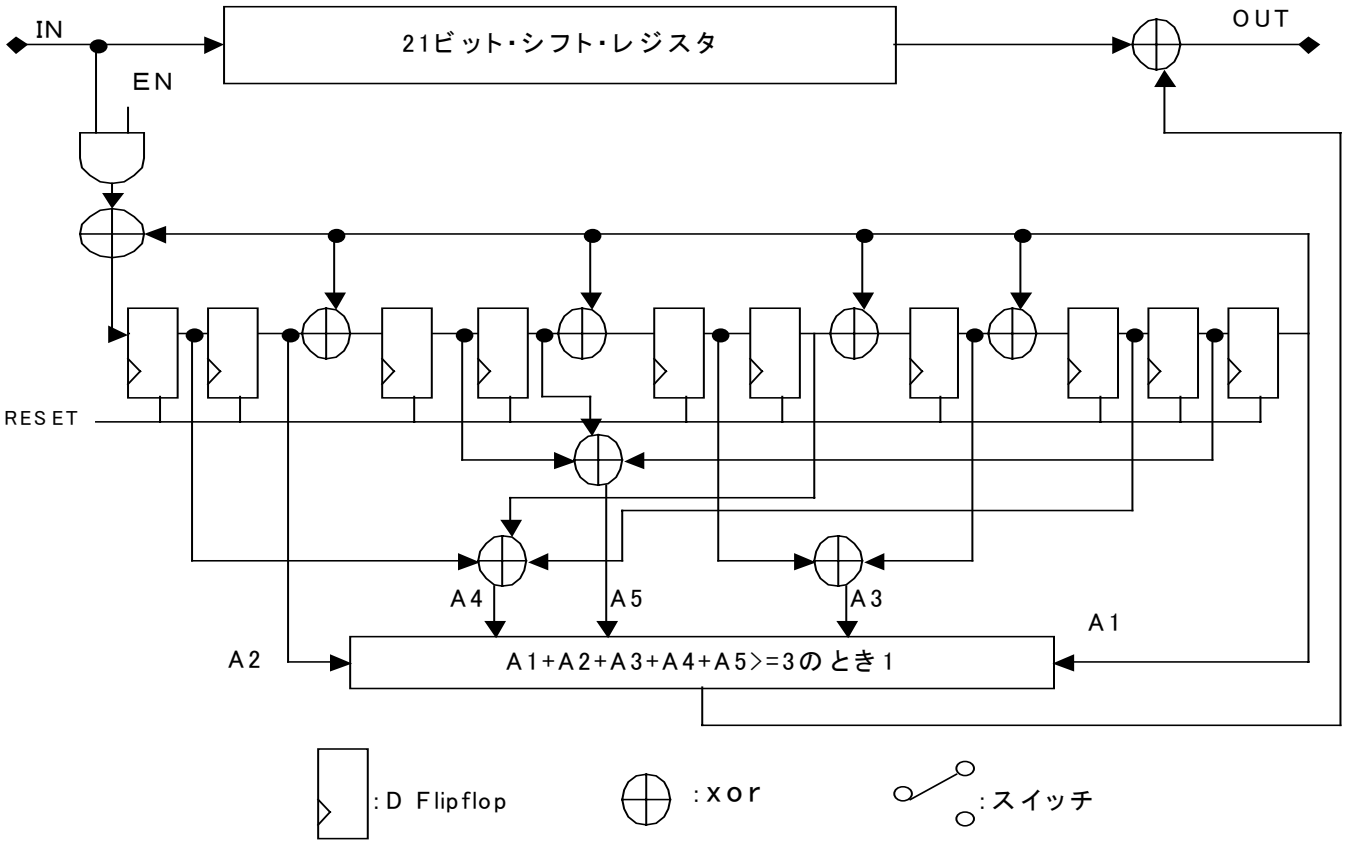

図 7：エラー訂正受信機

4.4 結果と考察

作成した受信回路を配置配線ツールにてシミュレーションした結果を図８に示す。0.8us の縦ラインが START の ’1’、つまり送信機から送られてきたデータ列の最初を表すラインである。そして 7.4us のラインがエラー訂正を完了したデータ列の出力し終わったラインである。つまり 21 ビット入力データ中の最初のビット入力から、11 ビットの出力データの最後のビットを出力し終わるまでの時間は、6.6us ということがわかる。図８は正常時では{1111 1111 111}を入力としている。しかしエラーを含めたためこの図８中では赤枠内の{1111 0111 111}を入力としている。しかしそれ以降のエラー訂正用ビットによって、青枠内の出力では{1111 1111 111}が出力されていることが確認できる。

また差集合巡回符号エラー訂正受信機の回路のコスト、および最高動作周波数を表９に示す。

ソフトウェアに関しては完成には至らなかったが、この通信はハードウェアで実現

する方がよいと思われる。その理由としては、ソフトウェアによってエラー訂正を実現すると受信後、転送されてきたデータからエラー訂正を完了させ、メモリなどに格納する工程が必要となる。そのため受信後、必要なデータが内部で利用可能になるまでに時間がかかってしまう。またそのエラー訂正のプログラムを実行するプロセッサがＣＰＵのようなものであった場合、そのＣＰＵが処理すべきほかのタスクが遅れてしまう可能性があるということである。そのためこれをハードウェアで実現して実装していれば、データ受信中であってもＣＰＵは他の処理に向けられる。また逆にＣＰＵが処理中でもデータ受信からエラー訂正まで行うことができる。しかもそれをハードウェアでは高速に行うことができるため、通信を常にする必要がある、または通信の頻度も高くそれ以外の処理量も多い用途の仕様であれば、エラー訂正をハードウェアで実現するメリットは十分にあると考えられる。

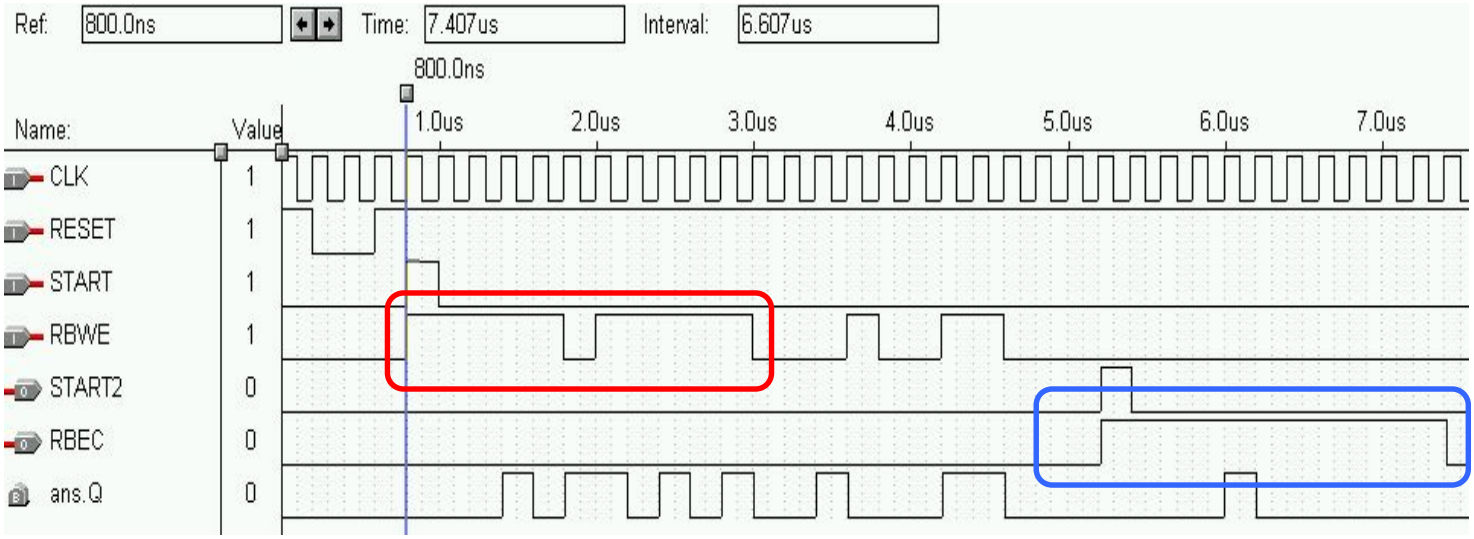

図 8：差集合巡回符号エラー訂正波形観測結果

表 9：差集合巡回符号エラー訂正ハードウェア仕様

Dflipflop	35
Logic Cells	55
最高動作周波数[MHz]	38.3

5 おわりに

本研究では、ＦＰＧＡを使用したハードウェア設計の手順を理解するため、TopDown設計について学んだ。そのためにまず「TopDown 設計 Training Kit」によってＨＤＬ記述からＦＰＧＡへのダウンロードまでの一連の工程を理解した[1]。そして付属されていたKUE-CHIP2のＶＨＤＬファイルを論理合成し、配置配線ツールによってＦＰＧＡチップにダウンロードできることを確認し、プログラムの動作も確認することができた。自作のハードウェアとしてはインクリメントの回路を作成し、チップ上で動作することを確認した。その上でM系列乱数発生を実現するハードウェアとソフトウェアの製作をした。

またエラー訂正回路については受信機部分の配置配線後のシミュレーションのみによる動作確認となった。シミュレーション上で入力データに１ビットエラーを含めて入れることで、そのエラーが修正されていることが確認できた。これによりこの差集合巡回符号エラー訂正によるエラー訂正が可能であることがわかった。

今後の課題としては、差集合巡回符号エラー訂正のソフトウェアによるプログラムの作成があげられる。しかし、送受信の動作手順がハードウェアと異なるため送信と受信を１つのプログラムで行っては、評価対象として違う存在となってしまうと思われる。そのため送信のみ、または受信のみのプログラムによっての検証となると思われるが、よりアルゴリズムを理解した上で、プログラムの作成と検証を行っていきたい。また今回作成した乱数発生，エラー訂正回路、共にプッシュボタンによるインターフェースを使用しなかった。そのため作成できる回路の範囲がかなり狭くとなってしまった。しかしこれを制御することができるようになれば、もっと様々な回路を作成することができるようになる。よってボタンによるインターフェースや液晶表示による動作結果の確認などを行うことも今後の課題としてあげられる。

謝辞

本研究に関して貴重なご助言，ご指導をいただきました山崎　勝弘教授，西村　俊和助教授に感謝いたします。またＦＰＧＡボードの使用方法などについての様々な質問にお答えしていただいた株式会社トッパン・テクニカル・デザインセンターの茂木浩介氏に感謝いたします。

またハードウェア設計に際し、いろいろな相談に応じてくださったTran Cong So氏，松井　誠二氏，そして同じハードウェア班として相談にのってくれた池田　修久氏，大八木　睦氏，および本研究室の皆さんに感謝いたします。

参考文献

[1] 茂木浩介・山田明宏：TopDown 設計 Training Kit，凸版印刷株式会社エレクトロニクス事業本部(2001)

[2] 田丸啓吉：論理回路の基礎（改訂版），工学図書株式会社(1989)

[3] 長谷川裕恭：ＶＨＤＬによるハードウェア設計入門，ＣＱ出版社(1995)

[4] メモリーＩＣ規格表，ＣＱ出版社(1988)

[5] ７４シリーズＩＣ規格表，ＣＱ出版社(1990)

[6] Exemplar. LeonardoSpectrum, http://www.exemplar.com

[7] 三菱電機マイコン機器ソフトウェア株式会社京都事業所：Rapid Prototyping Kit MU200-VIKT ユーザーズマニュアル(2000)

[8]和田知久：差集合巡回符号エラー訂正回路設計仕様書（ＣＱ出版社：Design Wave Magazine vol.48 Nov.2001 pp.155～168）

[9] 琉球大学　和田知久　教授, http://bw-www.ie.u-ryukyu.ac.jp/~wada/

[10]ＩＣ／ＬＳＩ基礎用語辞典（トランジスタ技術付録）,ＣＱ出版社

[11] KUE-CHIP2 用アセンブラ：筑波大学電子・情報工学系技官室技術主任　小野 雅晃氏,
http://orchid3.is.tsukuba.ac.jp/~ono/homepage/kueasm.html

[12] KUE-CHIP2 Web アセンブラ：神戸大学大学院自然科学研究科　上嶋　明氏,
http://www25.cs.kobe-u.ac.jp/~uejima/kue-chip2.html

[13] 鈴木直美の「PC Watch 先週のキーワード」：ＦＭ文字多重放送の説明
http://www.watch.impress.co.jp/pc/docs/article/991119/key98.htm

[14] 情報学実験III　教育用ボードコンピュータ（KUE-CHIP2 の仕様）

[15] 情報科学辞典，岩波書店(1990)

付録１　M系列乱数発生 KUE-CHIP2 用アセンブリプログラム

mrand.asm

初期値を 50h 番地に代入
一工程ごとに 50h 番地に乱数を出力

```
ldacc,55H
stacc,(50H)* 初期値代入
loop:rcf
eorix,ix
sllacc
stacc,(8FH)
bncj1
ldix,01H
rcf
j1:sllacc
sllacc
sllacc
ldacc,00H
bncj2
ldacc,01H
j2:eorix,acc
ldacc,(8FH)
oracc,ix
stacc,(50H)* 算出結果を出力
baloop

hlt
end
```

付録２　M系列乱数発生ＶＨＤＬ

mrand.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mrand is
port( CLK, Reset: in std_logic;
RAND: out std_logic_vector(7 downto 0) );
end mrand;

architecture STRUCTURE of mrand is
begin
process (CLK, Reset)
variable ShiftReg : std_logic_vector(7 downto 0);
begin
if (Reset = '0') then
ShiftReg := "01010101";
elsif (CLK'event and CLK = '1') then
for i in 7 downto 1 loop
ShiftReg(i) := ShiftReg(i-1);
end loop;
ShiftReg(0) := ShiftReg(7) xor ShiftReg(4);
RAND <= ShiftReg;
end if;
end process;
end STRUCTURE;
```

付録 3　差集合巡回符号エラー訂正送信機

transmitter.vhd

```
-- Synopsys Design Contest 2002
-- transmitter.vhd
-- 2001/September/10th
-- TASK: Differencial Cyclic Code FEC
-- Copyright by  Tom Wada@Univ. of the Ryukyus

library IEEE;
use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;

entity TRANSMITTER is
    port (SBWE     : out std_logic;
  SB       : out std_logic;
  START    : out std_logic;
  RESET    : in  std_logic;
  CLK      : in  std_logic );
end entity TRANSMITTER;

architecture RTL of TRANSMITTER is
    -- phase counts from 0 to 41
    -- phase  0 to 20 : sync bits are transmitted
    -- phase 21 to 31 : information bits transmitted
    -- phase 32 to 41 : parity bits transmitted
    signal  phase    : unsigned (5 downto 0);
    -- 11bits information bits
    signal  infbit   : unsigned (10 downto 0);
    -- internal start signal
    signal  intstart : std_logic;
    -- internal sb signal
    signal  intsb,intsb2    : std_logic;
    -- 10 parity bits
    signal  parity   : unsigned (9 downto 0);
    -- error signal
```

```
    signal  errsig   : std_logic;
    -- error counter
    signal  errcnt   : unsigned (9 downto 0);

-- RESET = '0'
constant RST_ACT: std_logic := '0';


begin
-----------------------------------------------
-- 42 phase (0 to 41 counter) generation unit
-----------------------------------------------
    PHASE_CNT: process(CLK,RESET)
    begin
        if (RESET= RST_ACT) then
    phase <= "000000";
elsif rising_edge(CLK) then
            if (phase="101001") then -- if phase=41
phase <= "000000";
            else
                phase <= phase + 1;
    end if;
end if;
    end process PHASE_CNT;
-----------------------------------------------
-- 11 bits information generation unit
-- count down from 1111111111 by 1
-----------------------------------------------
    INF_GEN: process(CLK, RESET)
    begin
if (RESET= RST_ACT) then
    infbit <= "00000000010";
        elsif rising_edge(CLK) then
            if (phase="010100") then -- if phase=20
infbit <= infbit - 1;
            end if;
```

```
        end if;
    end process INF_GEN;
-----------------------------------------------
-- internal start (intstart)  generation
-----------------------------------------------
    START_GEN: process(CLK, RESET)
    begin
if (RESET= RST_ACT) then
    intstart <= '0';
        elsif rising_edge(CLK) then
    if (phase="010101") then -- if phase=21
       intstart <= '1';
            else intstart <= '0';
    end if;
        end if;
    end process START_GEN;
-----------------------------------------------
-- internal sb signal (intsb) generation
-----------------------------------------------
    SB_GEN: process(CLK, RESET)
    begin
if (RESET= RST_ACT) then
            intsb <= '0';
        elsif rising_edge(CLK) then
            case phase is
      when "000000"=> intsb <= '0'; -- 0
      when "000001"=> intsb <= '0'; -- 1
      when "000010"=> intsb <= '1'; -- 2
      when "000011"=> intsb <= '1'; -- 3
      when "000100"=> intsb <= '0'; -- 4
      when "000101"=> intsb <= '1'; -- 5
      when "000110"=> intsb <= '0'; -- 6
      when "000111"=> intsb <= '1'; -- 7
      when "001000"=> intsb <= '1'; -- 8
      when "001001"=> intsb <= '1'; -- 9
      when "001010"=> intsb <= '1'; --10
```

```
      when "001011"=> intsb <= '0'; --11
      when "001100"=> intsb <= '1'; --12
      when "001101"=> intsb <= '1'; --13
      when "001110"=> intsb <= '1'; --14
      when "001111"=> intsb <= '0'; --15
      when "010000"=> intsb <= '0'; --16
      when "010001"=> intsb <= '0'; --17
      when "010010"=> intsb <= '0'; --18
      when "010011"=> intsb <= '0'; --19
      when "010100"=> intsb <= '0'; --20
      when "010101"=> intsb <= infbit(10); --21
      when "010110"=> intsb <= infbit(9);  --22
      when "010111"=> intsb <= infbit(8);  --23
      when "011000"=> intsb <= infbit(7);  --24
      when "011001"=> intsb <= infbit(6);  --25
      when "011010"=> intsb <= infbit(5);  --26
      when "011011"=> intsb <= infbit(4);  --27
      when "011100"=> intsb <= infbit(3);  --28
      when "011101"=> intsb <= infbit(2);  --29
      when "011110"=> intsb <= infbit(1);  --30
      when "011111"=> intsb <= infbit(0);  --31
      when  others => intsb <= 'X';  --others
           end case;
end if;
    end process SB_GEN;
-----------------------------------------------
-- parity calculation
-----------------------------------------------
    PARITY_CAL: process(CLK, RESET)
    begin
if (RESET= RST_ACT) then
           parity <= "0000000000";
       elsif rising_edge(CLK) then
           if (phase="010101") then
parity <= "0000000000";
           elsif (phase>="010110" and phase<="100000") then
```

```
parity(9) <= parity(8);
parity(8) <= parity(7);
parity(7) <= parity(6) xor intsb xor parity(9);
parity(6) <= parity(5) xor intsb xor parity(9);
parity(5) <= parity(4);
parity(4) <= parity(3) xor intsb xor parity(9);
parity(3) <= parity(2);
parity(2) <= parity(1) xor intsb xor parity(9);
parity(1) <= parity(0);
parity(0) <= intsb xor parity(9);
            else
                parity <= parity(8 downto 0) & parity(9);
            end if;
        end if;
    end process PARITY_CAL;
-----------------------------------------------
-- error interval counter
-----------------------------------------------
    ERR_CNT: process (CLK, RESET) begin
        if (RESET= RST_ACT) then errcnt <= "0000000000";
elsif rising_edge(CLK) then
            if (errcnt = "0000001001") then -- max=9 (0-9)
 errcnt <= "0000000000";
            else errcnt <= errcnt +1;
            end if;
end if;
    end process ERR_CNT;
-----------------------------------------------
-- error signal generation
-----------------------------------------------
    ERROR_GEN: process (CLK, RESET)
    begin
        if (RESET= RST_ACT) then
   errsig <= '0';
elsif rising_edge(CLK) then
           if (errcnt="0000000000") then
```

```
       errsig <= '1';  -- error happens every 10 cycle
           else errsig <= '0';
   end if;
        end if;
    end process ERROR_GEN;
-----------------------------------------------
-- output signals
-----------------------------------------------
    SBOUT: process (CLK, RESET)
    begin
if (RESET= RST_ACT) then
    intsb2 <= '0';
        elsif rising_edge(CLK) then
            if(phase >="100001" or phase="000000") then
                intsb2 <= parity(9);
            else
intsb2 <= intsb;
            end if;
end if;
    end process SBOUT;

    STARTOUT: process (CLK, RESET)
    begin
if (RESET= RST_ACT) then
    START <= '0';
        elsif rising_edge(CLK) then
            START <= intstart;
        end if;
    end process STARTOUT;

    SB   <= intsb2;
    SBWE <= intsb2 xor errsig;

end architecture RTL;
```

付録４　差集合巡回符号エラー訂正受信機

receive_mak.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity RECEIVER is
    port (START: in std_logic;
  RBWE: in std_logic;
  RESET: in std_logic;
  CLK: in std_logic;
  START2: out std_logic;
  RBEC: out std_logic );
end RECEIVER;

architecture RTL of RECEIVER is
-- phase
signalphase: unsigned( 5 downto 0 );
-- internal register & answer
signalans: std_logic;
-- ShiftReg
signalShiftReg: std_logic_vector( 20 downto 0 );
-- RESET = '0'
constant RST_ACT: std_logic := '0';
-- EN = '0'
signalEN: std_logic;

begin

--------------------------------------------------------------------------
-- start bit recieving
--------------------------------------------------------------------------
rcv_flag:process( RESET, START, CLK, phase )
begin
if( RESET = RST_ACT )then
EN<= '1';
```

```
elsif( rising_edge(CLK) )then
if( START = '1' or phase = "1000001" )then
EN<= '1';
elsif( phase = "010011" )then  -- phase = 15-2h
EN<= '0';
end if;

end if;
end process rcv_flag;


-------------------------------------------------------------------------
-- phase counter
-------------------------------------------------------------------------
PHASE_CNT:process( CLK, RESET, START )
begin
if( RESET = RST_ACT )then
phase <= "000000";
elsif( rising_edge(CLK) )then
if( START = '1' )then
phase <= "000000";
else
phase <= phase + 1;
end if;
end if;

end process PHASE_CNT;


-------------------------------------------------------------------------
-- Start2 generator
-------------------------------------------------------------------------
stt_gen2:process( RESET, phase )
begin
if( RESET = RST_ACT )then
START2<= '0';
```

```
elsif( rising_edge(CLK) )then
if(phase = "010100")then-- phase = 14h
START2<= '1';
else
START2<= '0';
end if;
end if;
end process stt_gen2;


--------------------------------------------------------------------------
-- ShiftRegister worker
--------------------------------------------------------------------------
Shftrg:process( CLK, RESET )
begin
if( RESET = RST_ACT )then
ShiftReg<= ( others => '0');
else
if( rising_edge(CLK) )then
ShiftReg <= ShiftReg(19 downto 0) & RBWE;
end if;
end if;
end process Shftrg;


--------------------------------------------------------------------------
-- error cancel worker
--------------------------------------------------------------------------
DECODE:process( CLK, RESET, START )
variable reg: std_logic_vector( 9 downto 0 );
variable ov_reg: std_logic;
variable A1, A2, A3, A4, A5: std_logic;
variable addans: unsigned( 2 downto 0 );

begin
```

```
A1:= '0';
A2:= '0';
A3:= '0';
A4:= '0';
A5:= '0';
addans:= "000";

if( RESET = RST_ACT )then
reg:= (others => '0');

elsif( rising_edge(CLK) )then
ov_reg:= reg(9);
reg(9):= reg(8);
reg(8):= reg(7);
reg(7):= reg(6) xor ov_reg;
reg(6):= reg(5) xor ov_reg;
reg(5):= reg(4);
reg(4):= reg(3) xor ov_reg;
reg(3):= reg(2);
reg(2):= reg(1) xor ov_reg;
reg(1):= reg(0);
reg(0):= ( RBWE and EN ) xor ov_reg;

A1:= ov_reg;
A2:= reg(1);
A3:= reg(4) xor reg(6);
A4:= reg(0) xor reg(5) xor reg(7);
A5:= reg(2) xor reg(3) xor reg(8);

if(A1 ='1')then
addans := addans + 1;
end if;
if(A2 ='1')then
addans := addans + 1;
end if;
if(A3 ='1')then
```

```
addans := addans + 1;
end if;
if(A4 ='1')then
addans := addans + 1;
end if;
if(A5 ='1')then
addans := addans + 1;
end if;

if(addans >= "011" )then-- A1+A2+A3+A4+A5 >= 3
ans <= '1';
else
ans <= '0';
end if;

if( phase >= "010100" and phase <= "100000" )then
RBEC <= ShiftReg(20) xor ans;
elsif( START='1' )then
RBEC <= ShiftReg(20) xor ans;
else
RBEC<= '0';
end if;
end if;

end process DECODE;

end RTL;
```