発表プログラム番号　TB-03

令和6年度 卒業論文

FPGAを用いたOFDM復調器の製作

Development of OFDM Demodulator using FPGA

山口雄大

指導教員　髙﨑和之

東京都立産業技術高等専門学校

目次

1. はじめに

流星バースト通信（Meteor Burst Communication）は地球に降り注ぐ宇宙の塵と大気の摩擦により生じる電離気体柱を反射体として利用する見通し外通信である。MBC の装置は極めて簡単な構成で作ることがで、簡易性・経済性に優れているが、通信路が開く時刻と継続時間が確率的であり、通信路が開いてからの受信信号強度は指数減衰するという特徴がある[1]。

MBC では従来手法として、位相シフトキーイング（Phase Shift Keying）によるシングルキャリア伝送が用いられてきた。しかし、この変調方式は MBC の受信信号強度が指数減衰する特性の影響を受けて、パケットの後半部分に連続した誤りが発生し、結果としてパケット全体が破棄されてしまう問題があるが、直交周波数分割多重方式（Orthogonal Frequency Division Multiplexing）を用いた並列伝送を行えばビット誤り率を大幅に低減できることが分かっている[2]。しかし、先行研究では PC を用いた OFDM の変復調は行われていたが、消費電力や、一般的な PC の OS はリアルタイム処理に不向きであるなどの問題がある。そこで、本研究ではデバイス内の論理ブロックを複数組み合わせて所望の論理回路を実現する Field Programmable Gate Array(FPGA)を用いて、MBC 環境での利用を目的とした OFDM のリアルタイム復調器を製作し、その性能を評価した。

FPGA に Gowin 社の GW1NR-9、評価基板に Tang Nano 9K、アナログディジタル変換機（Analog to Digital Converter）に Microchip 社の MCP3002 を採用した。復調器は部品点数が少なく、シンプルな構造であるため、ブレッドボードで製作することができた。製作費用も約 3500 円と安価にすることができた。OFDM の一次変調方式にはノイズに強く、回路、プログラムの製作が容易である 2 値位相シフトキーイング（Binary Phase Shift Keying）を採用した。信号仕様は先行研究に準じて決定した。一度の OFDM シンボルで送信するデータは 12 バイト、OFDM シンボルを 9 回送信、1 回休止を 1 セットとし、10 セット繰り返したものを OFDM 信号として用いた。Python で書かれた自作プログラムを実行することで、生成された OFDM 信号を PC からオーディオインターフェイスを介して復調器に入力し、評価を行った。まず、送信したデータと復調器から出力されたデータは全て一致しており、サブキャリアとパイロット信号の比が信号仕様通りであることから、FPGA が正しく動作していることを確認できた。また、1 シンボルの復調にかかる時間は 0.634ms~3.170ms であり、OFDM のシンボル時間の 21.3ms の約 3~14%の時間で復調できることを確認した。FPGA の使用していないリソースは十分にあり、今後の機能追加などで他の処理を組み込めることも確認した。しかし、製作した復調器は相関を用いた同期処理を行っていないため、途中から受信された OFDM シンボルを復調することができない。MBC 伝送路が信号を送信している途中に発生した場合など、信号が途中から受信されることは十分起こり得るため、この機能の実装は必須である。また、パイロット信号を用いた伝送路特性の補正を行う予定だったが、補正用の除算器の設計が間に合わず、補正を行うことができなかった。実際の MBC 環境で復調を行うためにはこれらの機能の実装が今後の課題である。

2. 理論

2.1 ディジタル変調

搬送波に情報を含ませることを変調といい、搬送波を余弦波としたとき、変調信号$v(t)$は式(2.1)で表せる。

$$v(t) = A_c \cos(2\pi f_c t + \emptyset_c) \tag{2.1}$$

振幅A_c、周波数f_c、位相ϕ_cのどれか一つを変化させる変調方式をそれぞれ振幅変調（Amplitude Modulation）、周波数変調（Frequency Modulation）、位相変調（Phase Modulation）という。送信する情報にアナログ信号などの連続波形を用いるものをアナログ変調、2 値パルスなどのディジタル信号を用いるものをディジタル変調という。前述した 3 つの変調方式はアナログ変調のときの名称であり、ディジタル変調では、それぞれ振幅シフトキーイング（Amplitude Shift Keying）、周波数シフトキーイング（Frequency Shift Keying）、位相シフトキーイング（Phase Shift Keying）という。アナログ変調とディジタル変調では変調方式の名称が異なるが、搬送波の操作内容はどちらも同じである。ディジタル変調方式にはこれら 3 つの他に、ASK と PSK を組み合わせた直交振幅変調などがある。本研究では、PSK の一種である、2 値のディジタル情報によって位相を変化させる 2 値位相シフトキーイング（Binary Phase Shift Keying）を用いた。一度の変調で送るデータのまとまりをシンボルといい、BPSK は 1 シンボルに割り当て可能なデータが 1 ビットと少ないが、実部のみを扱うため、回路やプログラムの作成が容易である。また、振幅の比較のみを行う単純なアルゴリズムで復調できるため、雑音にも強い。図 2.1 に 4 つのシンボルを伝送したときの BPSK の信号波形と位相と符号の割り当てを示す。このとき、位相と符号の割り当ては余弦波（搬送波）の位相$\phi_c = 0$のとき符号”1”、 余弦波（搬送波）の位相$\phi_c = \pi$のとき符号”0”とした。

BPSK を含むこれまで紹介した方式は搬送波を直接操作することから一次変調方式と呼ばれている。対して、搬送波を直接操作せず、一次変調された信号に追加の操作を加える方式は二次変調方式と呼ばれている。二次変調方式には大きく分けて、拡散方式と多重化方式の 2 つがあり、本研究では多重化方式を用いた。

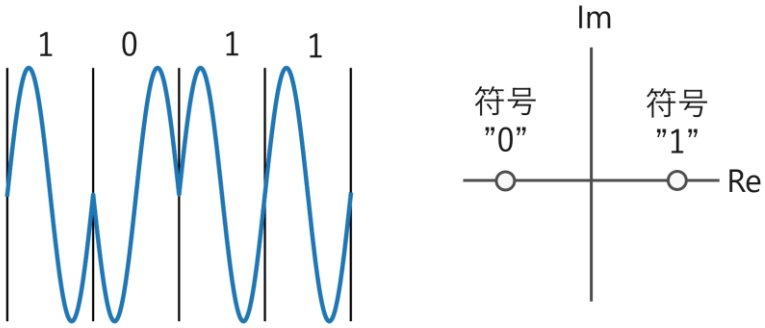


図 2.1　BPSK の信号と位相と符号の割り当て

2.2 周波数分割多重方式（FDM）

周波数分割多重方式（Frequency Division Multiplexing）とは、周波数帯域が重ならないように複数の異なる周波数の搬送波を用いて、それぞれ変調する多重化方式である。AM と FM で同じチャネル数を用いる場合、AM の方が FM より多重化した後の周波数帯域幅は小さくなる。FM は周波数帯域幅を広くとる必要があるため、帯域幅を広くとることができるマイクロ波などで多重度が大きい場合に使われる。ディジタル変調では、ディジタル信号との親和性の高さから後述する OFDM が用いられることが多い。多重化通信では周波数領域で搬送波同士が干渉してしまうと、正しく通信できなくなってしまうため、干渉を防ぐ必要がある。FDM では搬送波間にガードバンドと呼ばれる周波数軸上で電力が 0 の部分を設けることで、搬送波同士の干渉を防いでいる。FDM はガードバンドの帯域幅分だけ周波数帯域幅が増えるため、周波数利用効率が悪い。

図 2.2 に一次変調に AM を用いたときの FDM のブロック図を示す。送信は複数の異なる搬送波をチャネルごとに変調し、変調した信号を足し合わせて一つの信号にし、伝送する。特定のチャネルを取り出すには、帯域通過フィルタ（Band Pass Filter）を用いて、必要なチャネル以外の周波数を減衰させ、その信号を復調器に入力することで、元の信号を取り出すことができる。FDM は搬送波の数だけ変調器・復調器が必要となるため、搬送波数が増えると回路規模やコストの面で問題となる。FDM の利用例として、第一世代のアナログ方式の携帯電話があげられる。

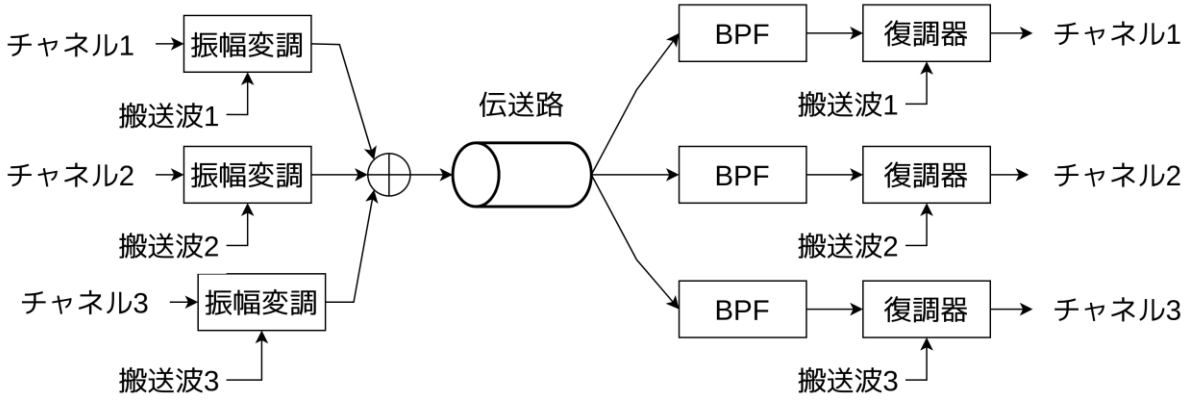


図 2.2　一次変調に AM を用いたときの FDM のブロック図

2.3 直交周波数分割多重方式（OFDM）

直交周波数分割多重方式（Orthogonal Frequency Division Multiplexing）はサブキャリアと呼ばれる多数の直交した周波数の搬送波を用いて変調し、並列伝送を行うマルチキャリア方式であり、二次変調方式の一種でもある。OFDM では送信データを周波数領域の振幅や位相にマッピングして扱う。変調は周波数領域の送信データを逆離散フーリエ変換（Inverse Discrete Fourier Transform）し、復調は受信した時間領域の信号を離散フーリエ変換（Discrete Fourier Transform）する。時間領域から周波数領域に変換することで、周波数領域の送信データを取り出すことができる。IDFT は送信データとサブキャリアを掛け合わるため変調に相当し、DFT は受信信号からサブキャリアを取り除いてデータを取り出すため復調に相当する。OFDM には次の特徴がある。

- サブキャリアが互いに直交していて、サブキャリア同士の干渉は発生しないため、ガードバンドが不要である。また、サブキャリアの直交性を利用しているため、一般的に FDM より OFDM の方がサブキャリア間隔は狭い。
- FDM はサブキャリア数と同じ個数だけの変調器・復調器が必要であるが、OFDM は変調に IDFT、復調に DFT を用いるため、一つの変調器・復調器で実現可能であり、回路規模やコストの面で優れている。
- マルチキャリア方式であるため、特定の周波数に強い影響を与える周波数選択性フェージングに強い。誤り訂正を用いると、フェージングの影響を受けた場合に訂正することが可能となるため、さらにフェージングの影響を低減できる。

FDM と OFDM の周波数利用効率の比較を図 2.3 に示す。図 2.3 より、ガードバンドの有無やサブキャリア間隔の違いで OFDM の方が周波数利用効率に優れていることが視覚的に分かる。

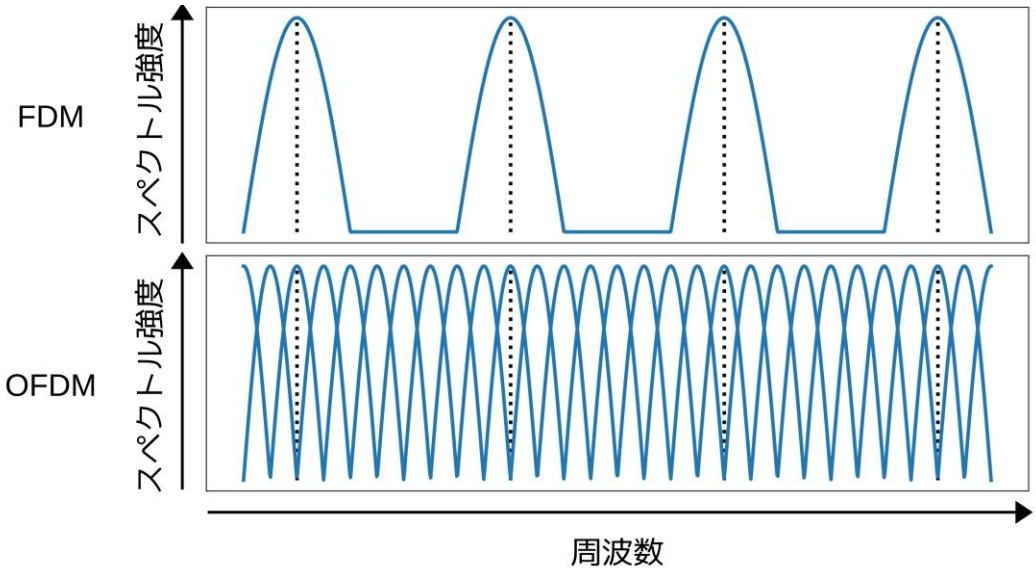


図 2.3　FDM と OFDM の周波数利用効率の比較

2.3.1 送受信

サブキャリア数をN、サブキャリア間隔をf_0、n番目のサブキャリアで伝送される複素データシンボルをD_nとしたとき、OFDM 信号x_kは式(2.2)で与えられる。

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} D_n e^{j\frac{2\pi}{N}nkf_0} \tag{2.2}$$

このとき、シンボル長Tは$1/f_0$となる。式(2.2)よりデータシンボルとサブキャリアを掛け合わせ、結果を足し合わせる動作は IDFT と同じであるため、OFDM の変調には IDFT が用いられる。復調は変調の逆の動作を行えば元のデータシンボルを取り出すことができるため、DFT が用いられる。x_nを時間領域の OFDM 信号、X_kを受信した複素データシンボルとすると、OFDM の復調は式(2.3)で与えられる。

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}nkf_0} \tag{2.3}$$

OFDM 送信機の構成例を図 2.4、受信機の構成例を図 2.5 に示す。送信では、まずビット列をシンボルに割り当てて、IDFT をする。IDFT の結果は離散信号であるため、ディジタルアナログ変換器（Digital to Analog Converter)を通して出力することで、離散信号を連続信号に変換する。DAC の出力信号は実部と虚部を含む複素信号であるため、実部に余弦波、虚部に正弦波を掛け合わせることで、実部のみの実信号に変換する。受信は送信の逆の処理を行えばよいので、受信した実信号に余弦波を掛け合わせ、低域遮断フィルタ（Low Pass Filter）を通して複素信号の実部を作る。同様に、受信した実信号に正弦波を掛け合わせ、LPF を通して複素信号の虚部を作り、実信号から複素信号への変換を行う。アナログディジタル変換機（Analog to Digital Converter）を用いてサンプリングし、連続信号を離散信号に変換する。DFT をしてシンボルを取り出すことで、元のビット列を得ることができる。

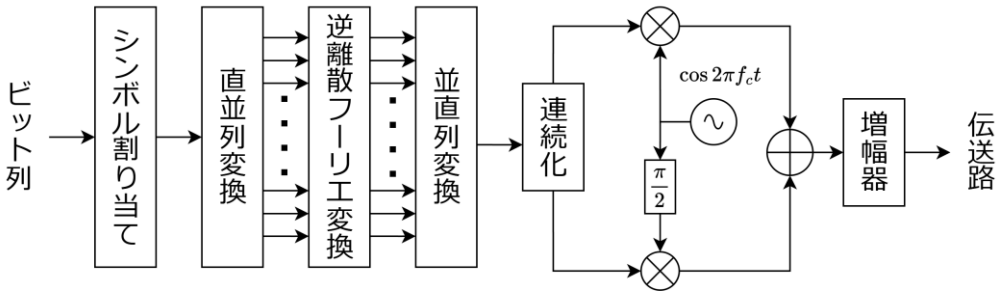


図 2.4　OFDM 送信機の構成例

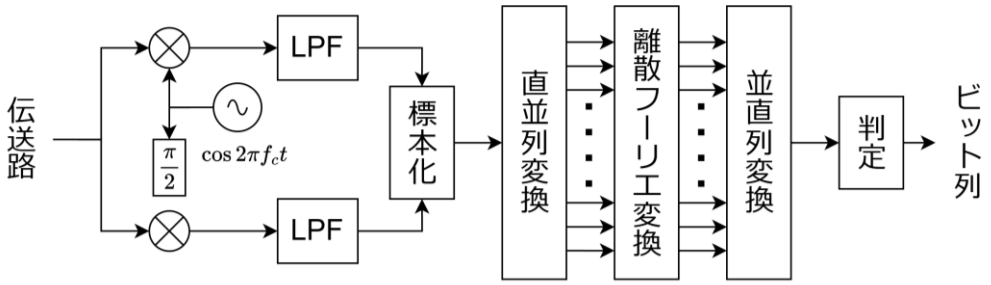


図 2.5 OFDM 受信機の構成例

2.3.2 時間領域の直交性

mとnを任意の整数とすると、式(2.4)~(2.6)の関係が成り立つ。

$$\int_{-\pi}^{\pi} \sin(mx)\cdot\sin(nx)\,dx = \begin{cases} 0 & m \neq n \\ \pi & m = n \end{cases} \tag{2.4}$$

$$\int_{-\pi}^{\pi} \cos(mx)\cdot\cos(nx)\,dx = \begin{cases} 0 & m \neq n \\ \pi & m = n, m \neq 0 \\ 2\pi & m = n = 0 \end{cases} \tag{2.5}$$

$$\int_{-\pi}^{\pi} \sin(mx) \cdot \cos(nx)\, dx = 0 \tag{2.6}$$

この関係を三角関数の直交性という。式(2.4)~式(2.6)より、周波数が異なり、その差が基本周波数の整数倍のとき、振幅の値に関係なく積分した値は 0 となる。このとき、それぞれの波の振幅は保たれ、互いに干渉しないということが分かる。

図 2.6 にサブキャリア数が 8 のときのサブキャリアと OFDM の信号波形を示す。上から 1~8 番目の波形はそれぞれ整数倍のサブキャリア、一番下の波形は式 8 つのサブキャリアを足し合わせて作られた OFDM の信号である。サブキャリアは BPSK で変調されており、サブキャリアの位相は図 2.6 中の右側に書かれた値が位相に対応している。OFDM のサブキャリア周波数はf_0の整数倍となるように配置されていることから、すべてのサブキャリアは直交している。図 2.6 より、周波数nf_0のサブキャリアにはシンボル長Tの区間にn周期分含まれていることが読み取れることから、図からもサブキャリアが直交していることが確認できる。また、このことから、BPSK で変調されていても、直交性が保たれていることも確認できる。

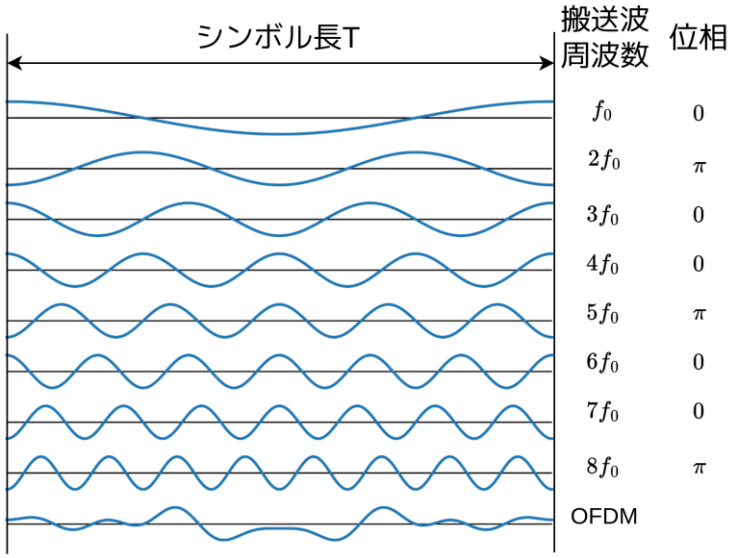


図 2.6 サブキャリア数が 8 のときの各サブキャリアの信号波形と OFDM の信号波形

2.3.3 スペクトル

OFDM はサブキャリア数が 1 のとき、シンボル長Tの余弦波となる。$f_{rect}(t)$を長さTの矩形窓とすると、シンボル長Tの余弦波のフーリエ変換は式(2.7)で与えられる。導出過程は付録 A に示す。

$$F\{f_{rect}(t) \cdot \cos(2\pi f_0 t)\} = \frac{\sin\{\pi(f+f_0)T\}}{2\pi(f+f_0)} + \frac{\sin\{\pi(f-f_0)T\}}{2\pi(f-f_0)} \tag{2.7}$$

X_kをk番目のサブキャリアのスペクトルとし、負の周波数は考えないものとしたとき、OFDM のサブキャリアのスペクトルは式(2.8)で与えられる。

$$X_k = \left| \frac{\sin\{\pi(f - kf_0)T\}}{2\pi(f - kf_0)} \right| \tag{2.8}$$

図 2.7 にサブキャリア数が 8 のときの OFDM のスペクトルを示す。それぞれの色の線はサブキャリアのスペクトルに対応している。図 2.7 より、あるサブキャリアのスペクトルが最大値のとき、他のサブキャリアのスペクトルは 0 であることから、周波数領域でも直交性が成り立っていることが分かる。f_0~f_7のスペクトルの最大値は一定であり、サブキャリア数が十分に多く、かつサブキャリア間隔が十分に狭いとき、OFDM のスペクトルは黒色の線で示した方形のような形となることから、スペクトルが集中していることがわかる。

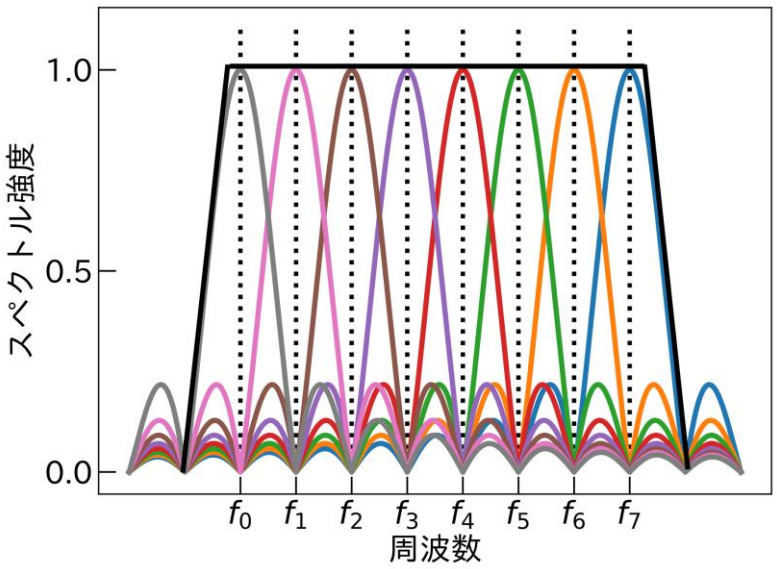


図 2.7 サブキャリア数 8 のときの OFDM のスペクトル

2.3.4 ガードインターバル

信号の先頭に先行波と遅延波の干渉を防ぐために挿入する信号のことをガードインターバルという。ガードインターバルがないと OFDM 信号が先行波とは別の遅延波がやってきて干渉したときに、干渉した部分の信号が使えなくなってしまい、復調できない。干渉を防ぐために信号の先頭に無信号区間を設ける方法があるが、無信号区間を設けると直交性が崩れてしまう。干渉を防ぐ対策として、OFDM シンボルの後半を先頭にコピーすることで直交性を崩すことなくガードインターバルを挿入するサイクリックプレフィックスと呼ばれる方法がある。図 2.8 にサイクリックプレフィックスを用いたガードインターバルの図を示す。送信する信号を A~Z までのアルファベットとし、ガードインターバルはアルファベットの後半部分にあたる WXYZ とする。このとき、先行波と遅延波どちらも、アルファベットの周期信号となっていることが分かる。図 2.9 にサイクリックプレフィックスの直感的な説明を示す。信号を車としたとき、異なる遅延時間の信号でも、サイクリックプレフィックスを用いているため、同じ信号を無限に繰り返すと、元の周期信号であることが分かる。

OFDM シンボルを途中から受信した場合、シンボルの開始位置がずれ、結果として DFT の開始位置がずれてしまい、正しく復調できないため、OFDM シンボルに付与されているガードインターバルを用いて同期処理を行う。ガードインターバルは後ろの信号を先頭にコピーしているため、シンボルの最初

と最後に$T = 1/f_0$だけ離れたところに同じ信号が存在するため、受信信号の自己相関関数（後述）を計算し、ピーク値を検出することで、シンボルの開始位置を検出することができる。

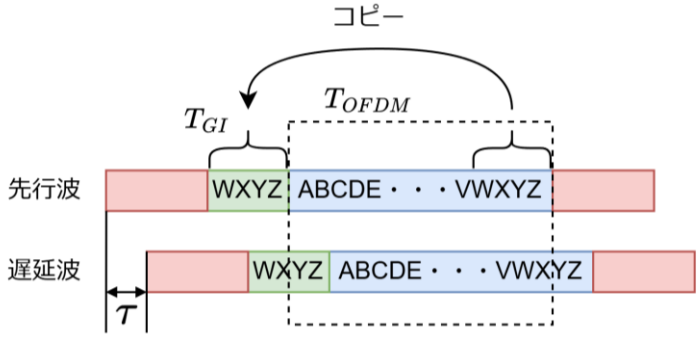
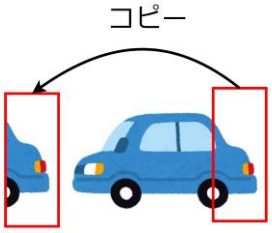
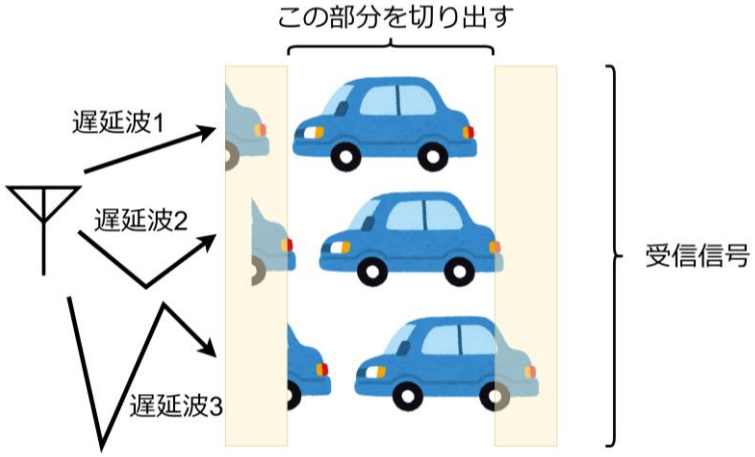


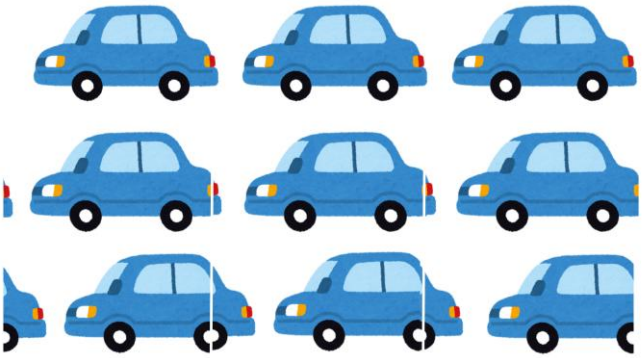
図 2.8　サイクリックプレフィックスを用いたガードインターバル



(a) 車に見立てたサイクリックプレフィックス付き OFDM シンボル



(b) 遅れの異なる 3 台の車と先頭車に合わせた車一台分での切り出し



(c) 無限に並ぶ同じ車の列

図 2.9　サイクリックプレフィックスの直感的な図説

2.4 高速フーリエ変換（FFT）

フーリエ変換は式(2.9)、逆フーリエ変換は式(2.10)を用いて行われる。

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot e^{-j2\pi ft} dt \tag{2.9}$$

$$x(t) = \int_{-\infty}^{\infty} X(f) \cdot e^{j2\pi ft} df \tag{2.10}$$

式(2.9)より、入力信号をx_n、サンプル数をN、離散フーリエ係数をX_kとすると、DFT は式(2.11)で与えられる。

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}nk} = \sum_{n=0}^{N-1} x_n W^{nk} \tag{2.11}$$

Wは回転因子と呼ばれ、$W \equiv e^{-j\frac{2\pi}{N}}$である。このとき、式(2.11)に忠実に計算すると DFT の計算量は$O(N^2)$となる。このときのプログラム例を付録 B に示す。式(2.11)の入力信号x_nの添え字nを偶数と奇数に分割すると、$k < N/2$のとき、式(2.12)と式(2.13)が与えられる。

$$X_k = \sum_{n=0}^{N/2-1} x_{2n-1} W^{2nk} + W^k \sum_{n=0}^{N/2-1} x_{2n} W^{2nk} \tag{2.12}$$

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n-1} W^{2nk} - W^k \sum_{n=0}^{N/2-1} x_{2n} W^{2nk} \tag{2.13}$$

X_kおよび$X_{k+N/2}$の第一項はxの添え字が奇数のものを DFT した結果であり、第二項はxの添え字が偶数のものを DFT し、その結果に回転因子W^kを掛け合わせたものである。回転因子W^kは$k < N/2$のとき符号はプラス、それ以外は符号がマイナスとなる。サンプル数Nの DFT はサンプル数$N/2$の DFT を用いて計算し、サンプル数$N/2$の DFT はサンプル数$N/4$の DFT を用いて計算する。同様に分割を続けていくと、最終的にサンプル数 1 の DFT を用いて計算することになる。これを分割統治法とよび、その流れを図 2.10 に示す。図の DFT の後ろの数字はサンプル数を示している。サンプル数Nが 2 の冪乗のときに分割統治法を用いると、DFT の計算量を$O(N^2)$から$O(N\log N)$に削減でき、このときのアルゴリズムを高速フーリエ変換（Fast Fourier Transform）という[3]。

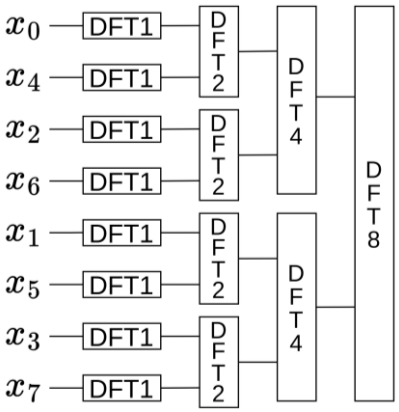


図 2.10　FFT の分割統治法の流れ

IDFT を高速で計算する逆高速フーリエ変換(Inverse Fast Fourier Transform)は FFT を用いて計算することで計算量$O(NlogN)$となる。式(2.14)のように、入力信号の複素共役を FFT し、その結果を複素共役し、最後にNで割ればよい。FFT と IFFT のプログラム例を付録 B に示す。

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k\, W^{-nk} = \frac{1}{N}\overline{\sum_{k=0}^{N-1} \overline{X_k}\; W^{nk}} \tag{2.14}$$

分割・計算手順を図にすると、図 2.11 に示すように蝶のような形になることから、バタフライ演算とも呼ばれる。図 2.11 中の⊕は加算を意味し、矢印は矢印の下の数との乗算を意味している。バタフライ演算は、演算の過程でデータの順序が変化するため、入力か出力のどちらかのインデックスをビット逆順にする必要がある。ビット逆順とは、十進数の数字を二進数にし、その二進数を逆順に並べ替えたものを、十進数に変換する動作のことである。本研究では、入力側のインデックスnをビット逆順にする時間間引き法で考える。

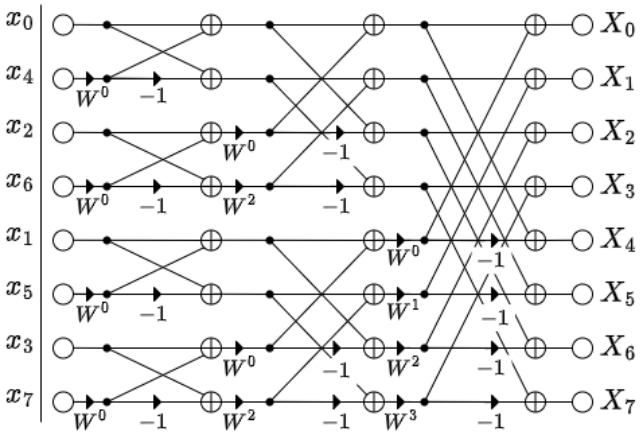


図 2.11　サンプル数$N = 8$のときのバタフライ演算

バタフライ演算は式(2.15)に示す演算を行うハードウェアのバタフライ演算機を用いることで、効率よく計算できる。x_a、x_bはバタフライ演算機の入力、X_a、X_bはバタフライ演算機の出力である。バタフ

ライ演算機では、入力と出力は同じアドレスの変数を用いても演算に影響しないため、入出力で同じアドレスの変数を共有することで、メモリを削減できる。

$$\begin{cases} X_a = x_a + W^k x_b \\ X_b = x_a - W^k x_b \end{cases} \tag{2.15}$$

2.5 相関

相関関数とは一方の波形を時間軸上で少しずらしながら、両波形の類似度をずらした時間（遅れ時間）の関数として表したものである。一つの信号に対して相関処理を行い、一つの信号の中に周期性があるかを調べることを自己相関、異なる二つの信号波形に対して相関処理を行い、二つの波形の類似度を調べることを相互相関という。プログラムでは入力信号が二つである相互相関が計算できれば、入力信号が一つである自己相関も計算できるため、これ以降相互相関についてのみ考える。無限長の連続信号のとき、入力信号を$x(t)$と$y(t)$、遅れ時間をτとすると、相互相関関数$R(t)$は式(2.16)で与えられる。

$$R(\tau) = \int_{-\infty}^{\infty} x(\tau) \cdot y(t-\tau)\ d\tau \tag{2.16}$$

式(2.16)より、有限長の離散信号の相互相関関数$R[m]$は入力信号を離散信号のxとy、 Nをxの大きさ、$0 \le m \le N$とすると式(2.17)で与えられる。

$$R[m] = \sum_{n=0}^{N-m-1} x[n] \cdot y[n-m] \tag{2.17}$$

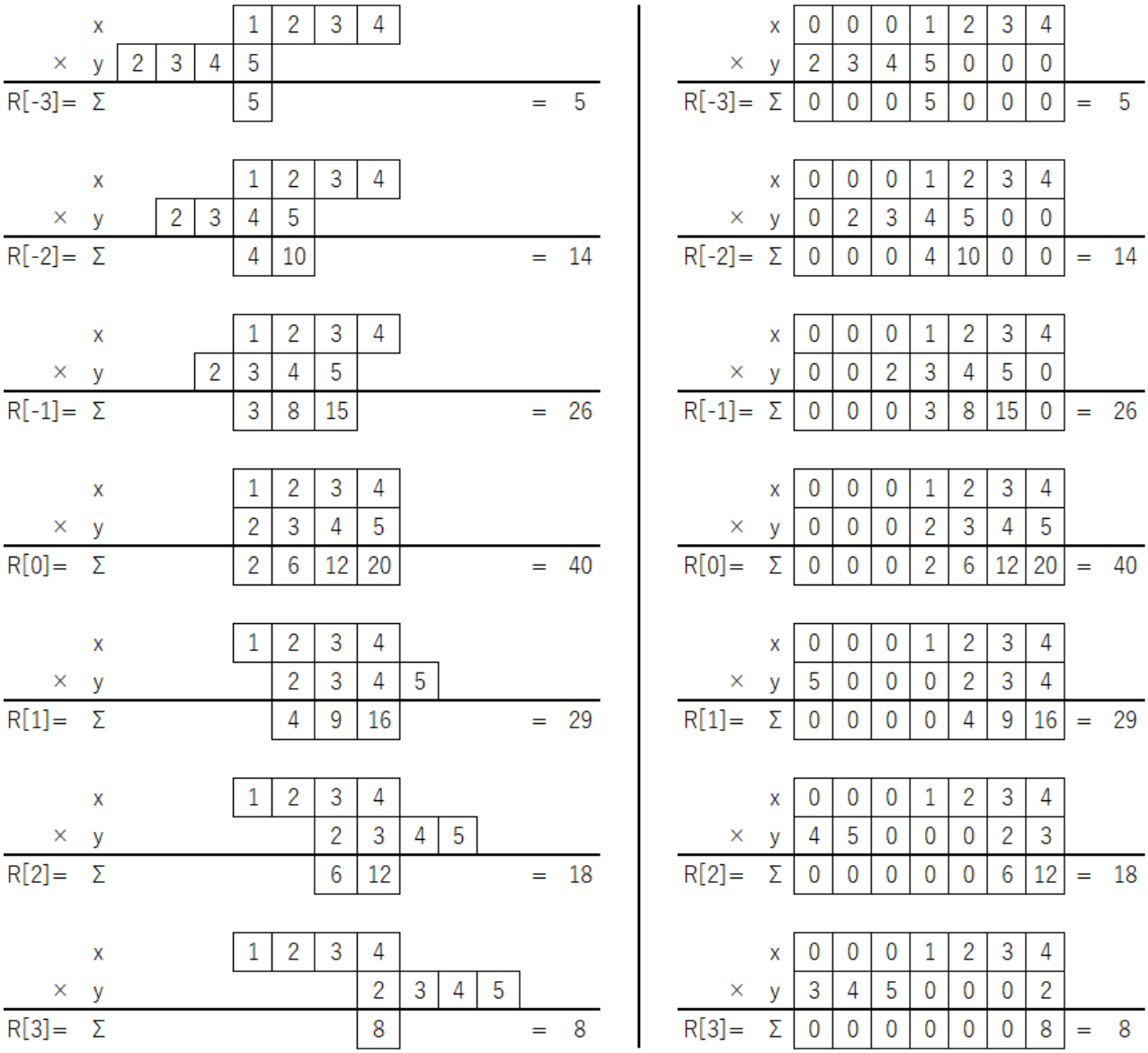
相関の計算量は式(2.17)に忠実に計算すると$\mathrm{O}(N^2)$となる。このときのプログラム例を付録 C に示す。計算量を少なくする方法として FFT を用いる方法がある。FFT を用いた相関では入力信号を周期関数として考える。xや y の長さを求める関数を$len(x)$、$len(y)$とし、xと y の長さをそれぞれ$\mathrm{len}(x) + \mathrm{len}(y) - 1$となるように 0 を挿入する。通常の相関では入力信号をシフトするが、この方法では入力信号が周期関数であることを利用し、巡回させて考える。このとき、相関はxの FFT とyの FFT の複素共役を乗算し、その結果を IFFT すること求めることができ、計算量は$\mathrm{O}(N\log N)$に削減できる。入力信号$x = [1,2,3,4]$、$y = [2,3,4,5]$としたときの相関の計算例を図 2.12 に示す。FFT の窓サイズが 2 の冪乗であることを考慮すると、次の手順で実装できる。

1. FFT の窓サイズを$2^{\lceil \log(\mathrm{len}(x)+\mathrm{len}(y)-1) \rceil}$とする。
2. xとyに 0 を挿入したものをそれぞれ$x1$、$y1$とする。$x1$は$len(y)-1$個の 0 を挿入し、その後xを挿入。残りは FFT の窓サイズに合うように 0 を挿入。$y1$はyを挿入し、その後に$len(x)-1$個の 0 を挿入。残りは FFT の窓サイズに合うように 0 を挿入。
3. $x1$を FFT し、結果を$X1$とする。同様に y1 を FFT し、結果を$Y1$とする。$X1$と$Y1$の複素共役を乗算し、乗算した結果を IFFT し、結果を$R1$とする。
4. $R1$は大きさを FFT の窓サイズに合わせた都合上、不要なデータが含まれているため、$0 \sim \mathrm{len}(x) +$

$\mathrm{len}(y)-2$番目までのデータを切り出し、それを相関の結果をRとする。

この手順の Python のプログラム例を付録 C に示す。

Matlab の xcorr では入力信号xと y の大きさが異なる場合、$N=\max(\mathrm{len}(x),\mathrm{len}(y))$とし、相関$R$の大きさは$2N-1$となり、$R[0]$が配列の中央の要素に対応するため、扱いやすい。配列の中央の要素とは、配列の前から$\lfloor \mathrm{len}(R)/2 \rfloor$番目の要素のことを示す。前述した実装方法や scipy.signal.correlate はxとyの大きさが異なっていてもRの大きさは$\mathrm{len}(x)+\mathrm{len}(y)-1$となり、このとき R[0]は配列の中央の要素に対応しないため、扱いにくい。本研究では Python や Matlab 上ではなく FPGA 上で実行する都合上、事前にxとyの大きさが決まっているため$R[0]$が配列の中央に来なくても問題ないこと、後者の方が実装が簡単であることから、後者の方法を採用することにしたが、相関のアルゴリズムを FPGA に組み込む開発時間を確保することができなかったため、相関を用いた信号の同期は断念した。



(a) シフトさせて計算する相関　　(b) 巡回させて計算する相関

図 2.12　相関の計算例

3 復調器

3.1 信号仕様

本研究で用いた OFDM の信号仕様を表 3.1、周波数領域のサブキャリアの配置を図 3.1 に示す。後述する復調器の仕様より FFT のサンプル数が 1024、ADC のサンプリング周波数が 48kHz なので、サブキャリア間隔は 48000/1024=46.875Hz、OFDM のシンボル時間は$1024/48000 \approx 21.3$msとなる。この信号仕様は FFT のサンプル数や ADC のサンプリング周波数の都合上、一部条件は異なるが、なるべく先行研究[2]と同じ条件となるようにした。パイロット信号と呼ばれる基準信号を用いて、伝送路特性の補正を行うことができるよう、サブキャリアを配置した。時間領域の OFDM 信号を図 3.2 に示す。OFDM 信号は同一の OFDM シンボルを 9 回送信、1 回休止を 1 セットとし、10 セット繰り返したものを用いた。最初と最後のデータは通信に成功したか判定に使用するため、0x55 の固定データとした。固定データは復調器で通信に成功したかの判定データとして利用する。

表 3.1　OFDM の信号仕様

一次変調方式	BPSK（位相 0° で符号”1”、位相 180° で符号”0”）
二次変調方式	OFDM
帯域幅[Hz]	984.375~5671.875
サブキャリア数	101（パイロット信号含む）
サブキャリアの振幅	1
サブキャリア間隔[Hz]	46.875
パイロット信号数	5（一定間隔で挿入）
パイロット信号の振幅	2
パイロット信号の位相[°]	0
1 シンボルの時間[ms]	21.3
データ量[Byte]	12（最初と最後は 0x55 で固定）

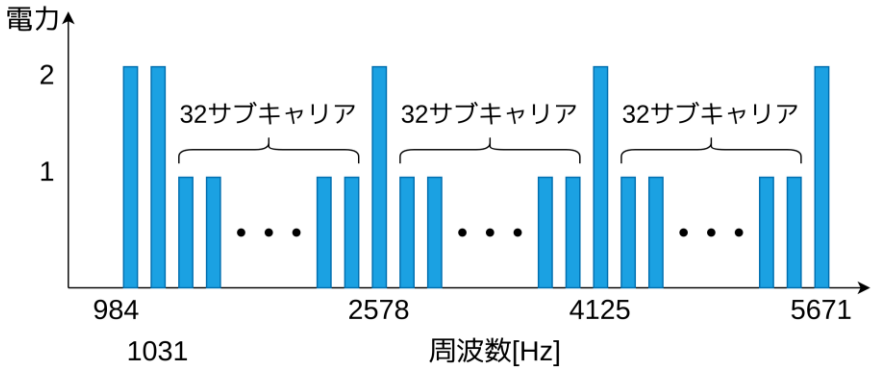


図 3.1　周波数領域のサブキャリアの配置

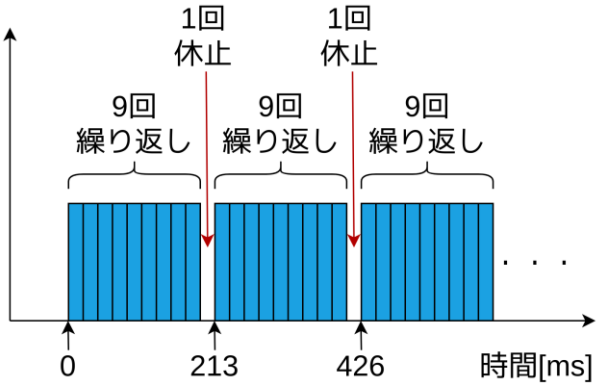


図 3.2　時間領域の OFDM 信号

3.2 ハードウェアの選定

先行研究では PC を用いた OFDM の変復調が行われていた。ここでは、PC は Raspberry Pi などのシングルボードコンピューター(SBC)などを含むものとする。MBC はインフラの整っていない地域で利用されることが多く、そのときの電源は太陽光発電で充電したバッテリーであるため、消費電力を可能な限り小さくする必要がある。低消費電力の PC でも、数 W 程度の電力を消費してしまうため、PC は消費電力の面では不向きである。また、Linux などの一般的な OS は多数のプロセスが常に動いており、割り込みも高頻度で発生するため、決められた短い時間内に処理を終わらせることを高頻度で要求されるシステムを実現するのは難しい。そのため、一般的に PC はリアルタイムの処理は苦手といわれている。これらの問題を解決するハードウェアとして、マイコンと Field Programmable Gate Array(FPGA)があげられる。マイコンとはシンプルな小型のコンピューターのことであり、FPGA とはデバイス内の論理ブロックを複数組み合わせて所望の論理回路を実現するデバイス[4]のことである。各ハードウェアの特徴を表 3.2 に示す。

表 3.2　各ハードウェアの特徴

	PC(SBC)	マイコン	FPGA
消費電力	高い	低い	低い
動作周波数	高い	低い	低い
リアルタイム処理	苦手	普通	得意
開発難易度	低い	普通	高い
得意な処理方式	逐次処理	逐次処理	並列処理
価格	高い	安い	普通

PC の消費電力は低消費電力のものでも数 W 程度であるのに対して、マイコンと FPGA の消費電力は mW オーダーか、それよりも小さいことから、PC よりマイコンと FPGA の方が消費電力は小さいといえる。マイコンと FPGA の消費電力は製品によって異なり、どちらの方が低消費電力かというのは一概には言えない。PC の動作周波数は GHz オーダーのものがほとんどであるが、マイコンや FPGA は MHz オーダーのものがほとんどであるため、PC の方が動作周波数は高いと評価した。一方で、PC でのリア

ルタイム処理は実現不可能ではないが、難易度の高さから一般的に PC はリアルタイム処理が苦手とされている。マイコンは OS を搭載しない、もしくはリアルタイム OS と呼ばれる軽量な OS が用いられるため、リアルタイム処理は得意であるといえる。FPGA はユーザーが論理回路を自由に組み、処理に最適な回路を作ることができるため、リアルタイム処理に特化しているので、リアルタイム処理性能はマイコンより FPGA の方が高い。PC はそのスペックの高さから、ハードウェアの性能、プログラミング言語などの制約がほぼないため開発難易度は低い。マイコンはハードウェアの性能の低さや扱えるプログラミング言語が少ないこと、またハードウェアのことを意識してプログラミングを行わなければいけないので、開発難易度が高い。FPGA はハードウェア記述言語を用いた開発を行わなければいけず、一般的にマイコン開発に用いられる C 言語などの高級言語での開発よりも時間がかかるとされている。また、マイコンのプログラミングでは Central Processing Unit(CPU)の 1 クロックごとの動作を常に考えながらプログラミングする必要はない、あったとしてもそのような機会は少ないないが、FPGA 開発ではクロックの立ち上がり・立ち下がりを意識しながら開発する必要がある。これらのことから、FPGA の方がマイコンより開発難易度は高いとした。PC とマイコンは CPU で演算を行っている。CPU は一度に一つのことを行う逐次処理が得意で、様々なことに対応できることから汎用性に優れているといえる。FPGA は任意の論理回路を組んで一度に複数の処理を行うことができるため、並列処理に優れているといえる。価格は一般的にマイコン、FPGA、PC の順に高価となる傾向がある。

本研究の目的は MBC 用の OFDM のリアルタイム復調器の製作である。低消費電力であり、リアルタイム性や並列処理性能に優れていることから、FPGA を採用した。

3.3 復調器の仕様

FPGA を用いて製作した復調器の仕様を表 3.3 に示す。

表 3.3　FPGA を用いて製作した復調器の仕様

FPGA	GW1NR-9(Gowin)
評価ボード	Tang Nano 9K(Sipeed)
10 ビット ADC	MCP3002(Microchip)
FPGA の動作周波数[MHz]	24
サンプリング周波数[kHz]	48
ADC のクロック周波数[kHz]	800
FFT のサンプル数	1024

FPGA と評価ボードは次の理由から表 3.3 のものを使用することにした。まず、多くの FPGA は海外から購入しないと手に入らないが、Tang Nano 9K は国内で電子部品などを取り扱っている秋月電子通商で取り扱っていることから、入手性に優れているといえる。そして、FPGA の評価ボードは 10000 円を超える高価なものが多いが、Tang Nano 9K は 2025 年現在秋月電子通商で 2980 円と安価である[5]。また、CQ 出版社から発売されている Interface 2022 年 12 月号 別冊付録 1 に 70 ページを超える Tang Nano 9K に関する特集もある。FPGA の開発経験がなかったため、Tang Nano 9K のスペックで何ができるかわからなかったが、RISC-V というオープンソースで提供されている命令セットアーキテクチャを FPGA 上

で実装できること、DVI 信号を生成してディスプレイへの映像出力ができることなどが雑誌に書かれていて[6]、これらのことを実現できるスペックであれば、復調器の製作も行えるだろうと考えた。

サンプリング周波数と FFT のサンプル数は表 3.3 に示す信号仕様を満たすように決定した。WAV ファイルと ADC のサンプリング周波数のずれが原因で予期せぬ問題が発生することを防ぐため、WAV ファイルと ADC のサンプリング周波数はどちらも同じ 48kHz とした。先行研究では 8 ビットの分解能の ADC で復調を行っていたため、分解能は 8 ビット以上あれば十分だと判断した。これらの要求を満たす 10 ビットの ADC である MCP3002 を選定した。MCP3002 は 2025 年現在秋月電子通商にて 240 円で販売されている[7]。データシートより、ADC でサンプリングを行い、データの出力を行うのは合計で 16 サイクルかかることが分かる[8]。そのため、ADC のクロック周波数は 48k×16=768k より大きい値とする必要があり、今回は 800kHz とした。

Tang Nano 9K には 27MHz の水晶発振器が搭載されいるが、27MHz を分周して、サンプリング周波数 48kHz 及び、ADC のクロック周波数 800kHz を作り出すことはできない。そのため、27MHz の周波数を GW1NR-9 に搭載されている PLL を用いることで 27MHz から 24MHz に逓倍することで対応した。

3.4 開発環境

OFDM 信号の生成は PC の OS には Debian GNU/Linux 12 Bookworm（以降 Debian 12）を用いた。プログラミング言語には開発経験があったため Python を用いた。バージョンは Debian 12 にデフォルトでインストールされている 3.11 を用いた。使用した Python の主要なライブラリを表 3.4 に示す。また、FPGA の開発に使用したツールを表 3.5 に示す。

表 3.4　使用した Python の主要なライブラリ

ライブラリ名	機能・用途
Matplotlib	グラフの描画に使用
matplotlib-fontja	Matplotlib の日本語フォント用
NumPy	数値計算ライブラリ
SciPy	科学技術計算ライブラリ
pySerial	FPGA とのシリアル通信に使用

表 3.5　FPGA の開発に使用したツール

用途	ツール名
IDE	Gowin EDA V1.99.03 Education Version
FPGA の書き込み	openFPGALoader
Verilog HDL のシミュレータ	Icarus Verilog
波形ビューア	GTK Wave

FPGA には Gowin 社の GW1NR-9 を使用しているため、IDE には Gowin 社のものを使用した。本来は有料の IDE だが、無料で公開されている教育用のものを使用した。一部機能が無料版のため制限されて

いるが、問題なく開発することができた。Linux 環境では IDE からの書き込みが安定しなかったため、openFPGALoader という無料のツールを利用した。デバッグや FPGA の性能評価には Icarus Verilog 上で Verilog HDL のシミュレーションを行い、波形ビューア GTK Wave で、シミュレーション結果の確認を行った。

3.5 復調の流れ

図 3.3 に復調器のブロック図を示す。復調の流れは次のとおりである。

1. PC 上で表 3.1 の仕様の OFDM 信号を生成し、サンプリング周波数 48kHz の WAV ファイルとして保存したものを、オーディオインターフェイスを介して再生し、復調器に入力した。
2. 入力信号は正負の電圧を含む信号であるが、ADC は正の電圧しか読むことができないため、加算回路を用いてオフセット電圧を加えることで、入力信号を正の電圧のみに変換する。
3. ADC は 48kHz でサンプリングを行い、結果を FPGA に出力する。
4. FPGA では OFDM の復調を行い、復調に成功した場合、結果をシリアル通信でパソコンに送信。

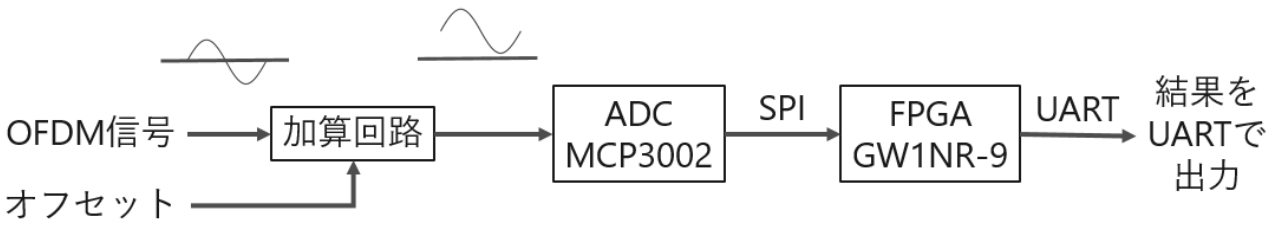


図 3.3　復調器の受信時のブロック図

本研究で製作した復調器の外観を図 3.4、回路図を図 3.5 に示す。回路図の 3.3V は USB 経由で評価ボードから出力される電圧を使用し、それ以外の電圧は安定化電源から供給した。

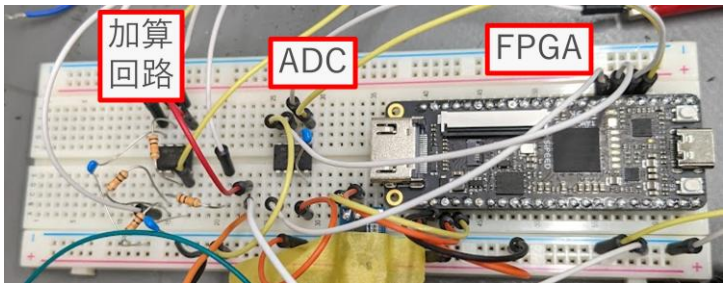


図 3.4　製作した復調器の外観

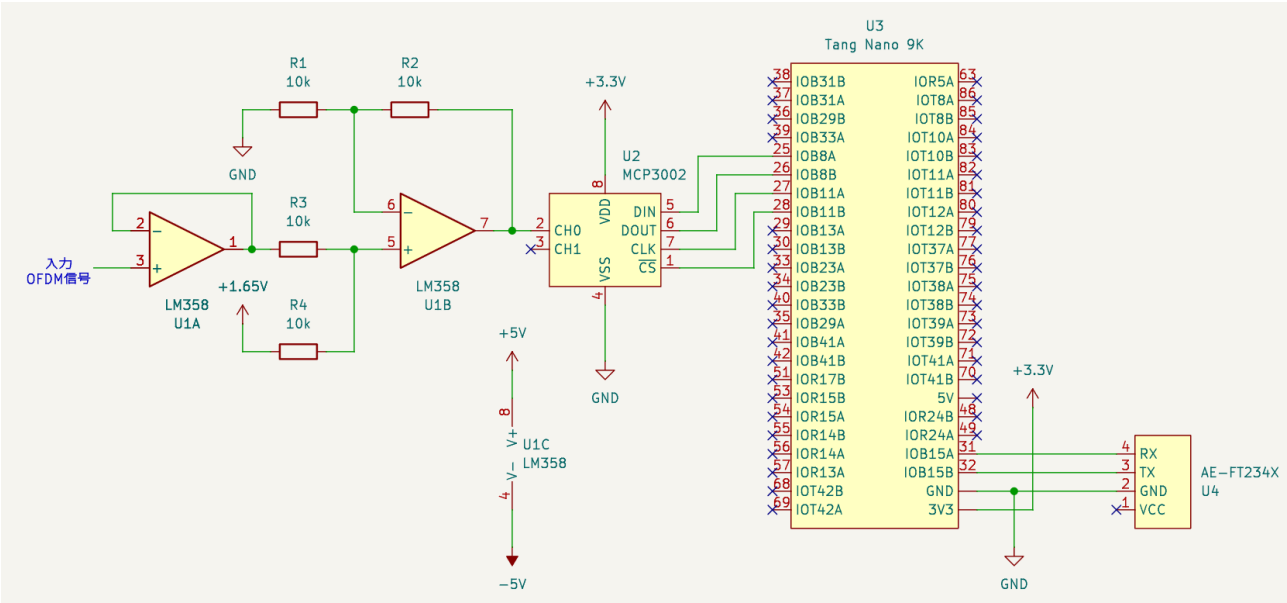


図 3.5　復調器の回路図

3.6 FPGA の動作

図 2.11 より、FFT はバタフライ演算の最終段以外はインデックス N/2 を境に上下で独立して演算可能である。本研究では、バタフライ演算機を二つ搭載することで、FPGA での演算速度の向上を図った。FFT の演算用の RAM が一つの場合、二つのバタフライ演算機が同時に同じ RAM にアクセスすると待ち時間が発生してしまうため、演算用の RAM もインデックス N/2 を境に二つ搭載することで待ち時間が発生しないようにした。回転因子は ROM に保存した正弦波テーブルを用いた。正弦波テーブルは正弦波 1 周期分の値を保存するのではなく、第一象限のみを保存し、他の象限の値を使用するときは、ROM にアクセスするときのインデックスを工夫することで、ROM の使用率を少なくした。N=1024 のとき、バタフライ演算器が 1 つの場合の演算回数は$1024 \times 10 = 10240$回、バタフライ演算器が 2 つの場合の演算回数は$\frac{1024}{2} \times 9 + 1024 = 5632$となり、バタフライ演算器を一つから二つに増やした結果、約 1.81 倍の演算速度の向上に成功した。

BSRAM(Block SRAM)とは FPGA 内部に搭載されているブロック単位の SRAM のことである。FPGA 内部にあるため高速にアクセスすることができ、異なるブロックの BSRAM は並列してアクセスすることができるのが特徴である。表 3.6 に BSRAM の割り当てを示す。それぞれの RAM には用途に応じた名前をつけた。RAM_FFT0 と RAM_FFT1 は図中で説明する際などはスペースの都合上、RAM0、RAM1 と表記する。

表 3.6　BSRAM の割り当て

名前	用途	BSRAM の個数	構成
RAM_FFT0(RAM0)	FFT のインデックス 0~(N/2-1)用	1	32bit×512
RAM_FFT1(RAM1)	FFT のインデックス N/2~(N-1)用	1	32bit×512
RAM_W	回転因子用	1	16bit×1024
RAM_ADC	ADC のサンプリング結果保存用	5	2bit×5×8192

各データのビットの使い方は ADC のサンプリングデータは ADC の分解能に合わせて 10bit、FFT のデータは実部と虚部それぞれ 16bit の符号付固定小数点数、計 32 ビットとした。FFT のデータは MSB 側から順に実部、虚部となるようにした。このように割り当てると、Verilog HDL で回路を記述するとき、x = {x_re, x_im};のように、実部、虚部の順番にすることができ、自然な並びとなる。16bit の符号付固定小数点数は符号 1bit、整数部 0bit、小数部 15 ビットとして割り当てた。値の範囲は$-1 \sim 1 - 2^{-15}$である。固定小数点数の加減算は整数と同じ方法でできるが、乗算を行うときは、整数と同様の方法で乗算を行った後、小数部のビット数分だけ右シフトを行う必要がある。RAM_ADC はサンプリングしたデータを蓄えるため、大きめに確保した。それぞれの RAM に 2bit の情報を 8192 個保存し、5 つ組み合わせて使うことで 10bit のデータを扱えるようにしている。回転因子用の RAM は FPGA の電源投入時に BSRAM の初期値を設定し、初期設定完了後は読み出し専用のメモリ(ROM)として使用している。FPGA の復調の流れを 1 から 5 で説明する。

1. FPGA はサンプリング周波数 48kHz で連続して電圧のサンプリングを行い、サンプリング結果は RAM_ADC に保存する。
2. ADC の値を積分し、閾値を超えたら OFDM シンボルのスタートと判断する。ADC の値を積分する理由は、積分をせずに ADC の値が閾値を超えたら信号の立ち上がりと判定した際に、雑音を信号の立ち上がりとして誤検知するのを防ぐためである。スタートの瞬間を含めて 1024 サンプルが完了した後、RAM_ADC の内容を RAM_FFT0 と RAM_FFT1 に転送する。ここで、RAM_ADC のインデックスをビット逆順したものを、RAM_FFT0 または RAM_FFT1 に書き込む。
3. バタフライ演算器を 2 つ用いて FFT を行う。回転因子の実部を保存する変数を W_RE、虚部を保存する変数を W_IM、搭載する 2 つのバタフライ演算器をそれぞれ BUTTERFLY0 と BUTTERFLY1 とする。BUTTERFLY0 の入出力用変数は X0 と X1、BUTTERFLY1 の入出力用変数は X2 と X3 とする。本研究で使う FFT は図 2.10 より最終段以外は 2 つのバタフライ演算器を用いて独立して行う。式(2.12)と式(2.13)より、X0 と X2、X1 と X3 のインデックスはそれぞれ N/2 ずれていると関係が成り立つ。最終段は BUTTERFLY0 のみを用いて行うものとする。X0 と X2 のインデックスを ADDR0、X1 と X3 のインデックスを ADDR1、W_RE のインデックスを ADDR2、W_IM のインデックスを ADDR3 とする。一時保管用変数をそれぞれ TMP0 と TMP1 とする。RAM0、RAM1、RAM_W の出力をそれぞれ、DOUT0、DOUT1、DOUT_W とする。本研究の構成では BSRAM に読み書き開始命令をしてから 2 サイクル後に実際に操作される。FFT のループ一回分の処理は 5 クロックで行われ、各クロックの大まかな動作を図 3.6 に示す。各クロックの詳細な動作は次のとおりである。各クロックの動作を(1)から(5)に示す。括弧の中の数字は何クロック目かを表していて、最初にコロンがついている場合はコロンの前に書かれている変数に対しての動作を意味している。
   (1) X0:RAM0 からアドレス ADDR0 の読み出しを開始する。
   X2:RAM1 からアドレス ADDR0 の読み出しを開始する。
   W_RE:RAM_W からアドレス ADDR2 の読み出しを開始する。
   インデックスを更新する。また、計算したインデックスから回転因子の符号も求める。各変数は 1 クロック目終了時にインデックスが更新される。1 クロック目で読み出し命令に使用するインデックスは更新前の値であるため、1 クロック目だけはインデックスを更新した値を別途計算し、使用する。

(2) 変数 X1:RAM0 からアドレス ADDR1 の読み出しを開始する。
X3:RAM1 からアドレス ADDR1 の読み出しを開始する。
W_IM:RAM_W からアドレス ADDR3 の読み出しを開始する。

(3) 3 サイクル目では 1 サイクル目に行った読み出し命令の結果が RAM から送られてくるので、その値を各変数に代入する。
X0 に DOUT0 をノンブロッキング代入する。
X2 に DOUT1 をノンブロッキング代入する。
W_RE に DOUT_W をノンブロッキング代入する。

(4) 4 サイクル目では 2 サイクル目に行った読み出し命令の結果が RAM から送られてくるので、その値をバタフライ演算器に入力する。バタフライ演算器で計算した結果は 1 つは RAM0 もしくは RAM1 に代入、もう一つは一時保管用変数に保存する。読み出し命令の結果は X1 が DOUT0、X3 が DOUT1、W_IM が DOUT_W に対応している。
バタフライ演算器 BUTTERFLY0 に DOUT0 と DOUT_W を入力する。
バタフライ演算器 BUTTERFLY1 に DOUT1 と DOUT_W を入力する。
BUTTERFLY0 の結果の X0 は RAM0 のアドレス ADDR0 にノンブロッキング代入する。X1 は一時保管用変数 TMP0 にノンブロッキング代入する。
BUTTERFLY1 の結果 X2 は RAM1 のアドレス ADDR0 にノンブロッキング代入する。X3 は一時保管用変数 TMP1 にノンブロッキング代入する。

(5) RAM0 のアドレス ADDR1 に TMP0 をノンブロッキング代入する。
RAM1 のアドレス ADDR1 に TMP1 をノンブロッキング代入する。

4. FFT 完了後は演算結果の符号を用いて BPSK を復調して、ビット列の最初と最後のデータが 0x55 である場合、復調成功と判断する。成功時は結果の 12 バイトをシリアル通信で送信する。失敗時は、スタート検出の際に、検出位置がわずかにずれてしまった可能性があるため、RAM_ADC のデータを一つシフトして一連の処理を試行する。シフトを 5 回繰り返して成功しない場合は、失敗と判断する。
5. 復調に成功した場合、同じ OFDM シンボルを繰り返し送信しているため、次の OFDM シンボルは前回の検出位置から 1024 サンプル離れた場所から始まることを利用して、復調を続ける。

Verilog HDL で実装した FFT のプログラムは付録 D に示す。

復調器の結果の出力方法には二種類のモードを用意した。一つ目は連続で信号の復調を行い、結果のみをシリアル通信で送信するモードである。二つ目はデバッグや性能評価を目的とした、1 回分の OFDM シンボルのみを復調する代わりに、FFT の結果をシリアル通信で送信するモードである。パイロット信号を用いて振幅の補正を行う予定だったが、補正には除算器が必要であり、除算器の作成が間に合わなかったため、パイロット信号を用いた補正は現時点では行っていない。

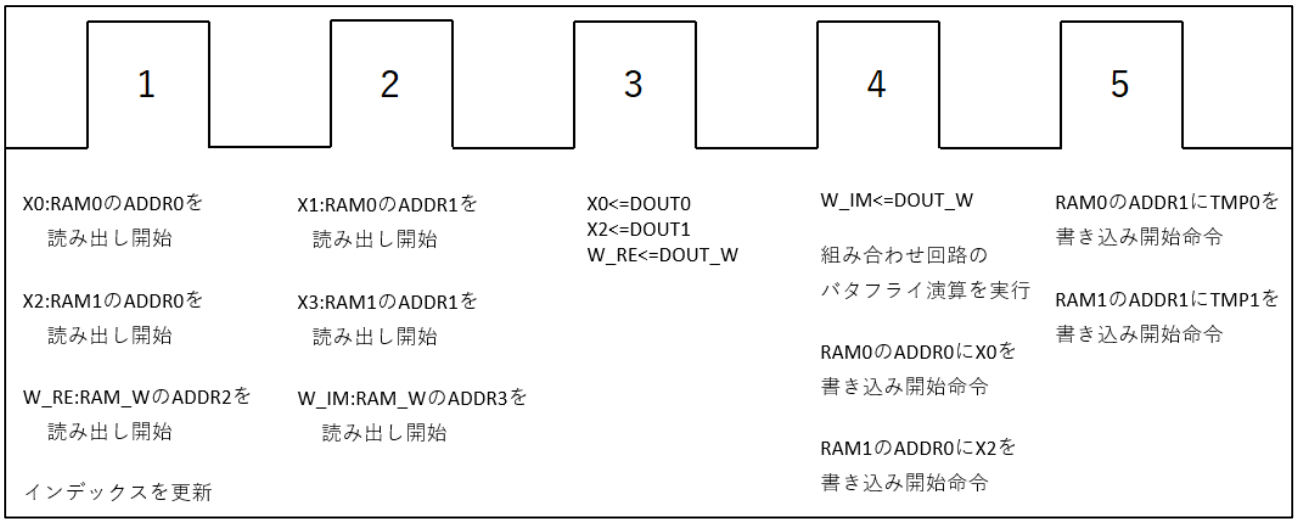


図 3.6　FFT の各クロックの動作

4 実験結果

PC 上で生成した OFDM 信号と OFDM 信号を加算回路に入力した結果を図 4.1 に示す。

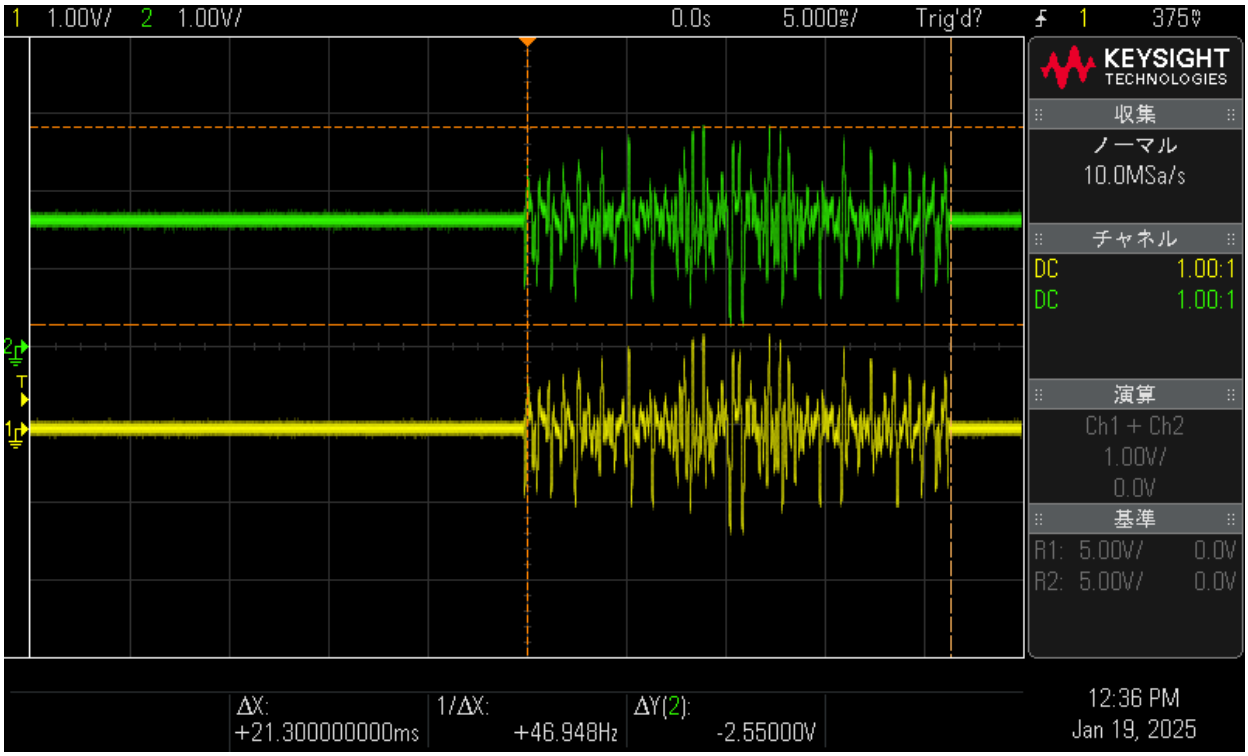


図 4.1　PC から入力した OFDM 信号（CH1、黄）と OFDM 信号をレベルシフトした結果（CH2、緑）

図 4.1 より、PC 上で生成した信号は正負の電圧であるのに対し、レベルシフトを行った信号は正の電圧のみであることから、レベルシフトに成功していることがわかる。

OFDM 信号をオーディオインターフェイスを介して製作した復調器に入力し、復調器で計算したフーリエ係数を図 4.2 に示す。

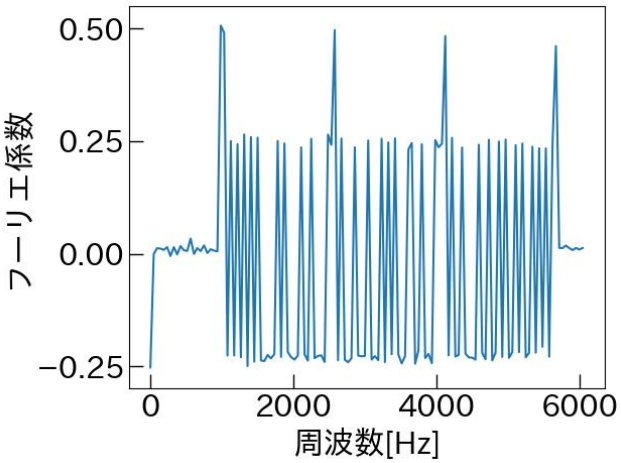


図 4.2　復調器で計算したフーリエ係数

サブキャリアのフーリエ係数が 0 より大きいものを符号”1”、0 より小さいものを符号”0”として割り当てた。図 4.2 より、サブキャリアの正負を確認すると、入力したビット列と一致していることが確認できた。10 セット繰り返して送った信号も、すべて復調することができた。パイロット信号の振幅が 0.5 程度であること、サブキャリアの振幅が 0.25 程度であり、その比が 2:1 であることからも正しく演算できていると考えられる。0Hz のフーリエ係数が 0 でないことから、レベルシフトの影響を受けていることや DC バイアス等の雑音があると考えられる。

FPGA の処理時間の評価は Icarus Verilog 上で行った。サンプリング終了後、RAM_ADC から RAM_FFT へのデータ転送に 42.8us、FFT の計算に 587us、BPSK 及び符号判定処理に 4.46us、合計 634us で OFDM の復調処理を行うことができた。シフト回数は最大 5 回であるため、最大シフト時の処理時間は 3.17ms となり、OFDM のシンボル時間である 21.3ms 以内に処理が終わることを確認した。これらのことから、シンボル時間の約 3~14%で演算可能だということが分かった。

使用した FPGA のリソースは表 4.1 に示す。

表 4.1　使用した FPGA のリソース

Register	834 / 6480 (12.9%)
LUT4	2095 / 8640 (24.2%)
16Kbit BSRAM	8 / 26 (30.8%)
MULT18X18	8 / 20 (40%)

FPGA のリソースは論理合成を行った際に IDE 上に表示されるものを用いた。MULT18X18 とは FPGA に搭載されているハードウェアの乗算器のことである。BSRAM は RAM_ADC が 5 つ、RAM_FFT0 が 1 つ、RAM_FFT1 が 1 つ、回転因子用の RAM が 1 つの計 8 つを使用している。表 4.1 より、FPGA のリソースは全体的に余裕があり、他の処理を組み込む余裕があると確認できた。

安定化電源（TEXIO PW24-1.5AQ）から電源を供給したときの復調器の消費電力を表 4.2 に示す。電圧と電流は、安定化電源に表示された値を使用した。表 4.2 より、消費電力の大半は FPGA が占めていて、合計電力は約 0.196W であることがわかる。オペアンプ用の負電源-5V に流れた電流が 0.000A なのは、オペアンプの消費電力が小さいため、安定化電源に表示されている桁数よりも少ない電流が流れたからと考えられる。

表 4.2　復調器の消費電力

電圧[V]	電流[A]	消費電力[W]
5.004	0.039	0.195156
-5.000	0.000	0.000
1.652	0.001	0.001652

5 考察と今後の課題

本研究の目標は FPGA を用いた OFDM のリアルタイム復調である。まず、入力したデータと出力したデータが完全に一致していることから復調に成功していることが確認できた。目標のリアルタイム復調は、シリアル通信の伝送時間を含めずに考えた場合、OFDM の 1 シンボルが 21.3ms であるのに対して復調時間が 634us~3.17ms であり、これはシンボル長の約 3~14%の時間である。このことから、十分高速に復調を行うことができたため、目標を達成できたといえる。また、FPGA のリソースも表 4.1 に示すよう十分に余裕があり、今後の機能追加が可能であることがわかる。製作した復調器は合計で約 3500 円程度と安価に製作することができた。部品点数が少ないため、本研究ではブレッドボードを用いて復調器を製作することができた。部品点数が少なくシンプルな構成であることも利点である。そして、FPGA を用いて復調を行うことで、わずか約 0.196W の消費電力の復調器を製作できた。PC の消費電力は一般的に数 W 程度であるため、十分消費電力の少ない復調器を作ることができたといえる。

今回製作した復調器は理想的な条件下で復調できるかどうかの性能評価しか行っておらず、SN 比を変えた場合や PC から出力される OFDM 信号を時間減衰させた場合の性能評価など詳細な性能評価を行うことができなかったため、性能評価は今後の課題である。

流星バースト通信は信号を送信している途中に、流星が地球に降り注ぎ、電離気体柱が発生し、通信路が作り出されることがある。このとき、復調器は途中から OFDM シンボルを受信することになるが、今回製作した復調器は信号の立ち上がりで OFDM シンボルの開始位置を検出しているため、この方法では途中から受信したときに復調することができない。そのため、相関を用いた信号の同期処理が必要である。また、パイロット信号を挿入しているものの、復調器でパイロット信号を利用した伝送路特性の補正は行えていないため、振幅の減衰や位相のずれに弱くなっている。パイロット信号を用いた伝送路特性の補正はサブキャリアの振幅とパイロット信号の受信した振幅の大きさの比を計算して補正をするため、除算を行う必要があるが、FPGA で除算を行うには、パイプライン処理を行う必要があり、それの設計・実装が間に合わなかったである。そのため、パイロット信号を用いた伝送路特性の補正も今後の課題である。そしてこれらの課題を解決し、実際の流星バースト通信環境での検証を行いたい。

参考文献

[1] 福田明、”流星バースト通信”、コロナ社、1997

[2] 髙﨑和之、若林良二、亀井利久、高塚徹、三寺史夫、”OFDM を用いた流星バースト通信に関する検討”、信学ソ大、B-1-19(2016)

[3] James W. Cooley, John W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series”、Mathematics of Computation、Vol.19、No90(1965)

[4] 天野英晴 他、”FPGA の原理と構成”、オーム社、2016

[5] 秋月電子通商、Tang Nanno 9K、https://akizukidenshi.com/catalog/g/g117448/ 、最終閲覧日 2025 年 3 月 3 日

[6] Interface 編集部、”Interface 2022 年 12 月号”、CQ 出版社、2022

[7] 秋月電子通商、10bit 2ch AD コンバーター MCP3002-I/P、https://akizukidenshi.com/catalog/g/g102584/、最終閲覧日 2025 年 3 月 3 日

[8] MICROCHIP、MCP3002、https://ww1.microchip.com/downloads/aemDocuments/douments/APID/ProductDocuments/DataSheets/21294E.pdf、最終閲覧日 2025 年 3 月 3 日

[9] 伊丹誠、"OFDM の基礎と応用技術"、電子情報通信学会、Vol.1、No.2(2007)、pp35-43

[10] 唐沢好男、"OFDM は万能選手？～等価伝送路モデルによる符号間干渉誤り解析～"、http://www.radio3.ee.uec.ac.jp/ronbun/TR_YK_055_ETP_OFDM.pdf、2020、最終閲覧日 2025 年 3 月 3 日

[11] 神谷幸宏、"MATLAB によるディジタル無線通信技術"、コロナ社、2008

[12] 猿渡洋、”信号処理理論第二第 3 回”、 https://www.sp.ipc.i.u-tokyo.ac.jp/~saruwatari/SP20_03.pdf 、2020、最終閲覧日 2025 年 3 月 3 日

[13] 小野測器、”計測コラム emm140 号用 基礎からの周波数分析 (9) ― 「高速フーリエ変換 (FFT)」”、https://www.onosokki.co.jp/HP-WK/eMM_back/emm140.pdf 、最終閲覧日 2025 年 3 月 3 日

[14] 大浦拓哉、”FFT (高速フーリエ・コサイン・サイン変換) の概略と設計法”、 https://www.kurims.kyoto-u.ac.jp/~ooura/fftman/ 、最終閲覧日 2025 年 3 月 3 日

[15] 小林一行、” コンボリューション関数（conv）と相関関数（xcorr）と離散フーリエ変換関数（fft）の関係”、 https://www.ikko.k.hosei.ac.jp/~matlab/xcorr.pdf 、2021、最終閲覧日 2025 年 3 月 3 日

[16] 竹下鉄夫、吉川英機、”電気・電子系 教科書シリーズ 23 通信工学”、コロナ社、2010

[17] 一之瀬優、”一陸技・無線工学 A 【無線機器】 完全マスター”、情報通信振興会、2015

[18] Gowin、”GW1NR シリーズ FPGA 製品データシート”、 https://cdn.gowinsemi.com.cn/DS117J.pdf 、2024、最終閲覧日 2025 年 3 月 3 日

付録 A　有限長余弦波のフーリエ変換

長さTの有限長余弦波のフーリエ変換について考える。フーリエ変換は式(2.9)とする。$F\{e^{j2\pi f_0 t}\} = \delta(f - f_0)$より、余弦波のフーリエ変換は式(A.1)で表すことができる。

$$F\{\cos(2\pi f_0 t)\} = F\left\{\frac{1}{2}\left(e^{j2\pi f_0 t} + e^{-j2\pi f_0 t}\right)\right\} = \frac{1}{2}\{\delta(f - f_0) + \delta(f + f_0)\} \tag{A.1}$$

図 A.1 に式(A.2)で表される矩形波$f_{rect}(t)$を示す。$f_{rect}(t)$のフーリエ変換は式(A.3)で表すことができる。

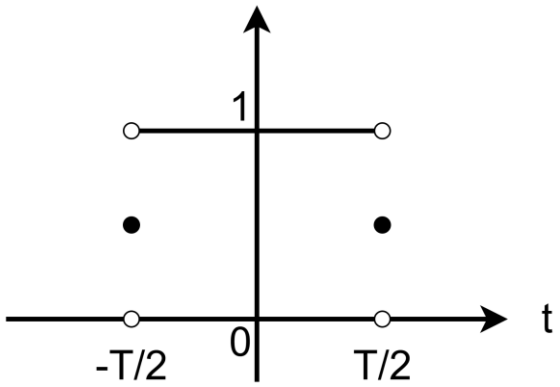


図 A.1 矩形波$f_{rect}(t)$

$$f_{rect}(t) = \begin{cases} 1 & |t| < T/2 \\ 0 & |t| > T/2 \\ 0.5 & |t| = T/2 \end{cases} \tag{A.2}$$

$$\int_{-\infty}^{\infty} f_{rect}(t) \cdot e^{-j2\pi f t} dt = \int_{-T/2}^{T/2} 1 \cdot e^{-j2\pi f t} dt = \frac{\sin(\pi f T)}{\pi f} \tag{A.3}$$

$F\{f_1(t) \cdot f_2(t)\} = F_1(f) * F_2(f)$より、長さ$T$の有限長余弦波のフーリエ変換は矩形波のフーリエ変換と余弦波のフーリエ変換の畳み込み積分となることが分かる。デルタ関数の畳み込み積分は
$\int_{-\infty}^{\infty} f(x) \cdot \delta(x - a) dx = f(a)$の関係より、シンボル長 T の余弦波のフーリエ変換は式(A.4)で表すことができる。

$$\begin{aligned} & F\{f_{rect}(t) \cdot \cos(2\pi f_0 t)\} \\ &= F\{f_{rect}(t)\} * F\{\cos 2\pi f_0 t\} \\ &= \frac{\sin(\pi f T)}{\pi f} * \left[\frac{1}{2}\{\delta(f - f_0) + \delta(f + f_0)\}\right] \\ &= \frac{\sin\{\pi(f + f_0)T\}}{2\pi(f + f_0)} + \frac{\sin\{\pi(f - f_0)T\}}{2\pi(f - f_0)} \end{aligned} \tag{A.4}$$

付録 B　DFT、FFT のプログラム

プログラム B.1　式に忠実に計算した DFT のプログラム

```
import numpy as np
from numpy.typing import NDArray


def dft(x: NDArray[np.complex128]) -> NDArray[np.complex128]:
    N = len(x)
    X = np.zeros(N, dtype=np.complex128)
    for i in range(N):
        for j in range(N):
            X[i] += x[j] * np.exp(-2j * np.pi * i * j / N)
    return X
```

プログラム B.2　FFT のプログラム

```
import cmath
import math
from typing import Any

import numpy as np
from numpy.typing import NDArray


def log2_int(x: int) -> int:
    ans: int = 0
    while x // 2 != 0:
        x = x // 2
        ans += 1
    return ans


def bit_reverse(x: Any) -> Any:
    x = x.copy()
    N = len(x)

    bit: int = log2_int(N)
    for i in range(N):
        k: int = 0
        for j in range(bit):
```

```
            k |= ((i & (0x01 << j)) >> j) << (bit - 1 - j)
        if i < k:
            # swap
            x[i], x[k] = x[k], x[i]
    return x


def fft(_x: NDArray[np.complex128]) -> NDArray[np.complex128]:
    """
    非再帰 FFT(時間間引き)
    """
    # 参照渡しになってしまうと扱いにくいのでコピー
    N = len(_x)

    # reverse
    x: NDArray[np.complex128] = bit_reverse(_x.copy())

    # fft
    step: int = 1
    while step < N:
        half_step: int = step
        step = step * 2
        w_step: complex = cmath.exp(-2j * math.pi / step)
        for k in range(0, N, step):
            w: complex = 1
            for j in range(half_step):
                u: complex = x[k + j]
                t: complex = w * x[k + j + half_step]
                x[k + j] = u + t
                x[k + j + half_step] = u - t
                w *= w_step

    return x
```

付録 C　相関のプログラム

プログラム C.1　式に忠実に計算したときの相関のプログラム

```
import numpy as np


def simple_correlate(x, y):
    R = np.zeros(len(x) + len(y) - 1)
    for i in range(-len(y) + 1, len(x)):
        # j は x のインデックス
        j = 0
        # k は y のインデックス
        k = 0
        if i >= 0:
            j = i
        else:
            k = -i

        while j != len(x) and k != len(y):
            R[i + len(y) - 1] += x[j] * y[k]
            j += 1
            k += 1
            print(i, j, k)
    return R
```

プログラム C.2　FFT を用いた相関のプログラム

```
import math

import numpy as np
from numpy.typing import NDArray


def correlate(
    x: NDArray[np.complex128], y: NDArray[np.complex128]
) -> NDArray[np.complex128]:
    """
    numpy.correlate(mode="full") scipy.signal.correlate()準拠の相関を求める関数
    FFT を用いて計算する。
    len(y) == 0 の場合、x の自己相関関数を求めるようにする。
```

```
    """

    if len(y) == 0:
        y = x
    l_corr = len(x) + len(y) - 1
    # FFT に入力する用の要素数が 2 の冪乗の配列を作成
    l_fft: int = 2 ** math.ceil(math.log2(l_corr))
    x1 = np.zeros(l_fft, dtype=np.complex128)
    y1 = np.zeros(l_fft, dtype=np.complex128)
    # 0 で padding
    for i in range(len(x)):
        x1[len(y) - 1 + i] = x[i]
    for i in range(len(y)):
        y1[i] = y[i]

    X1 = np.fft.fft(x1)
    Y1 = np.fft.fft(y1)
    R1 = np.fft.ifft(X1 * Y1.conj())

    # 必要な部分のみを切り取る
    R = np.zeros(l_corr, dtype=np.complex128)
    for i in range(len(R)):
        R[i] = R1[i]

    return R
```

付録 D　FPGA の FFT のプログラム

プログラム D.1 バタフライ演算器のプログラム

```
// x0 = x0 + x1 * w = (x0re + j * x0im) + ((x1re * wre - x1im * wim) + j(x1im * wre +
x1re * wim)))
// x1 = x0 - x1 * w = (x0re + j * x0im) - ((x1re * wre - x1im * wim) - j(x1im * wre +
x1re * wim)))
// x0re = x0re + x1re * wre - x1im * wim = x0re + A | x1re * wre | - B | x1im * wim |
= x0re + A0 | x1re * wre | + B0 | x1im * wim |
// x0im = x0im + x1im * wre + x1re * wim = x0im + C | x1im * wre | + D | x1re * wim |
= x0im + C0 | x1im * wre | + D0 | x1re * wim |
// x1re = x0re - x1re * wre + x1im * wim = x0re - A | x1re * wre | + B | x1im * wim |
= x0re + A1 | x1re * wre | + B1 | x1im * wim |
// x1im = x0im - x1im * wre - x1re * wim = x0im - C | x1im * wre | - D | x1re * wim |
= x0im + C1 | x1im * wre | + D1 | x1re * wim |
// A~D は掛け算をしたときの符号 (xor で求めることができる)
// 符号は 1 のときはマイナス、0 のときはプラス//  A = sign(x1re  * wre) = x1re ^ wre
// B = sign(x1im * wim) = x1im ^ wim
// C = sign(x1im * wre) = x1im ^ wre
// D = sign(x1re * wim) = x1re ^ wim
// A0~D0、A1~D1 は A~D とバタフライ演算のときに出てくる符号をかけ合わせたもの
// A0 = A
// B0 = ~B
// C0 = C
// D0 = D
// A1 = ~A
// B1 = B
// C1 = ~C
// D1 = ~D
// |=======input=======|                                  |=========output========|
// |x1re| wre|x1im| wim|x1re*wre|x1im*wim|x1im*wre|x1re*wim| x0re| x0im| x1re| x1im|
// |    |    |    |    |    A   |    B   |    C   |    D   |A0|B0|C0|D0|A1|B1|C1|D1|
// |-------------------------------------------------------------------------------|
// |  0 |  0 |  0 |  0 |    0   |    0   |    0   |    0   | 0| 1| 0| 0| 1| 0| 1| 1|
// |-------------------------------------------------------------------------------|
// |  0 |  0 |  0 |  1 |    0   |    1   |    0   |    1   | 0| 0| 0| 1| 1| 1| 1| 0|
// |-------------------------------------------------------------------------------|
// |  0 |  0 |  1 |  0 |    0   |    1   |    1   |    0   | 0| 0| 1| 0| 1| 1| 0| 1|
// |-------------------------------------------------------------------------------|
```

```
// |  0 |  0 |  1 |  1 |    0   |    0   |    1   |    1   | 0| 1| 1| 1| 1| 0| 0| 0|
// |-------------------------------------------------------------------------------|
// |  0 |  1 |  0 |  0 |    1   |    0   |    1   |    0   | 1| 1| 1| 0| 0| 0| 0| 1|
// |-------------------------------------------------------------------------------|
// |  0 |  1 |  0 |  1 |    1   |    1   |    1   |    1   | 1| 0| 1| 1| 0| 1| 0| 0|
// |-------------------------------------------------------------------------------|
// |  0 |  1 |  1 |  0 |    1   |    1   |    0   |    0   | 1| 0| 0| 0| 0| 1| 1| 1|
// |-------------------------------------------------------------------------------|
// |  0 |  1 |  1 |  1 |    1   |    0   |    0   |    1   | 1| 1| 0| 1| 0| 0| 1| 0|
// |-------------------------------------------------------------------------------|
// |  1 |  0 |  0 |  0 |    1   |    0   |    0   |    1   | 1| 1| 0| 1| 0| 0| 1| 0|
// |-------------------------------------------------------------------------------|
// |  1 |  0 |  0 |  1 |    1   |    1   |    0   |    0   | 1| 0| 0| 0| 0| 1| 1| 1|
// |-------------------------------------------------------------------------------|
// |  1 |  0 |  1 |  0 |    1   |    1   |    1   |    1   | 1| 0| 1| 1| 0| 1| 0| 0|
// |-------------------------------------------------------------------------------|
// |  1 |  0 |  1 |  1 |    1   |    0   |    1   |    0   | 1| 1| 1| 0| 0| 0| 0| 1|
// |-------------------------------------------------------------------------------|
// |  1 |  1 |  0 |  0 |    0   |    0   |    1   |    1   | 0| 1| 1| 1| 1| 0| 0| 0|
// |-------------------------------------------------------------------------------|
// |  1 |  1 |  0 |  1 |    0   |    1   |    1   |    0   | 0| 0| 1| 0| 1| 1| 0| 1|
// |-------------------------------------------------------------------------------|
// |  1 |  1 |  1 |  0 |    0   |    1   |    0   |    1   | 0| 0| 0| 1| 1| 1| 1| 0|
// |-------------------------------------------------------------------------------|
// |  1 |  1 |  1 |  1 |    0   |    0   |    0   |    0   | 0| 1| 0| 0| 1| 0| 1| 1|
// |-------------------------------------------------------------------------------|
module butterfly
(
    input [15:0] x0_re,
    input [15:0] x0_im,
    input [15:0] _x1_re,
    input [15:0] _x1_im,
    input [15:0] w_re,
    input w_re_sign,
    input [15:0] w_im,
    input w_im_sign,
    output [15:0] res_x0_re,
    output [15:0] res_x0_im,
    output [15:0] res_x1_re,
```

```
    output [15:0] res_x1_im
);

wire A, A0, A1, B, B0, B1, C, C0, C1, D, D0, D1;
wire [15:0] x1_re;
wire [15:0] x1_im;
wire [15:0] x1re_wre;
wire [15:0] x1im_wim;
wire [15:0] x1im_wre;
wire [15:0] x1re_wim;
wire [31:0] _x1re_wre;
wire [31:0] _x1im_wim;
wire [31:0] _x1im_wre;
wire [31:0] _x1re_wim;

assign A = _x1_re[15] ^ w_re_sign;
assign B = _x1_im[15] ^ w_im_sign;
assign C = _x1_im[15] ^ w_re_sign;
assign D = _x1_re[15] ^ w_im_sign;
assign A0 = A;
assign B0 = ~B;
assign C0 = C;
assign D0 = D;
assign A1 = ~A0;
assign B1 = ~B0;
assign C1 = ~C0;
assign D1 = ~D0;

assign x1_re = _x1_re[15] ? ~_x1_re + 16'd1 : _x1_re;
assign x1_im = _x1_im[15] ? ~_x1_im + 16'd1 : _x1_im;

assign _x1re_wre = x1_re * w_re;
assign _x1im_wim = x1_im * w_im;
assign _x1im_wre = x1_im * w_re;
assign _x1re_wim = x1_re * w_im;

assign x1re_wre = _x1re_wre[14] ? _x1re_wre[30:15] + 16'd1 : _x1re_wre[30:15];
assign x1im_wim = _x1im_wim[14] ? _x1im_wim[30:15] + 16'd1 : _x1im_wim[30:15];
assign x1im_wre = _x1im_wre[14] ? _x1im_wre[30:15] + 16'd1 : _x1im_wre[30:15];
```

```
assign x1re_wim = _x1re_wim[14] ? _x1re_wim[30:15] + 16'd1 : _x1re_wim[30:15];

function [15:0] calc;
input [15:0] x0;
input [15:0] x1w_0;
input [15:0] x1w_1;
input s0;
input s1;
case ({s0, s1})
    2'd0: calc = x0 + x1w_0 + x1w_1;
    2'd1: calc = x0 + x1w_0 + ~x1w_1 + 16'd1;
    2'd2: calc = x0 + ~x1w_0 + 16'd1 + x1w_1;
    2'd3: calc = x0 + ~x1w_0 + ~x1w_1 + 16'd2;
endcase
endfunction

assign res_x0_re = calc(x0_re, x1re_wre, x1im_wim, A0, B0);
assign res_x0_im = calc(x0_im, x1im_wre, x1re_wim, C0, D0);
assign res_x1_re = calc(x0_re, x1re_wre, x1im_wim, A1, B1);
assign res_x1_im = calc(x0_im, x1im_wre, x1re_wim, C1, D1);
endmodule
```

プログラム D.2　FFT のプログラム

```
module fft1024_twindle_factor_index
(
    input [9:0] i,
    output [23:0] res
);

localparam N = 11'd1024;
localparam N4 = 11'd256;
localparam N4_2 = 11'd512;
localparam N4_3 = 11'd768;

function [23:0] calc;
    // calc = {w_re_sign, i_re, w_im_sign, i_im};
    // fft1024 では N=4096 の回転因子を使用するため 4 倍する
    input [9:0] i;
    reg [10:0] _ad_re;
```

```
    reg [10:0] _ad_im;
    reg [10:0] ad_re;
    reg [10:0] ad_im;
    reg sign_re;
    reg sign_im;

    if (0 <= i && i <= N4) begin
        // 第 4 象限
        _ad_re = N4 - i;
        _ad_im = i;
        ad_re = _ad_re << 2;
        ad_im = _ad_im << 2;
        sign_re = 1'd0;
        sign_im = 1'd1;
        calc = {sign_re, ad_re, sign_im, ad_im};
    end
    else if(N4 < i && i <= N4_2) begin
        // 第 3 象限
        _ad_re = i - N4;
        _ad_im = N4_2 - i;
        ad_re = _ad_re << 2;
        ad_im = _ad_im << 2;
        sign_re = 1'd1;
        sign_im = 1'd1;
        calc = {sign_re, ad_re, sign_im, ad_im};
    end
    else if(N4_2 < i && i <= N4_3) begin
        // 第 2 象限
        _ad_re = N4_3 - i;
        _ad_im = i - N4_2;
        ad_re = _ad_re << 2;
        ad_im = _ad_im << 2;
        sign_re = 1'd1;
        sign_im = 1'd0;
        calc = {sign_re, ad_re, sign_im, ad_im};
    end
    else begin
        // 第 1 象限
        _ad_re = i - N4_3;
```

```
        _ad_im = i & (N4 - i);
        ad_re = _ad_re << 2;
        ad_im = _ad_im << 2;
        sign_re = 1'd0;
        sign_im = 1'd0;
        calc = {sign_re, ad_re, sign_im, ad_im};
    end
endfunction

assign res = calc(i);

endmodule

module fft1024
(
    input clk,
    input rst_n,
    input start,
    output reg finish,
    input clear,
    // BSRAM fft0
    input [31:0] dout0,
    output reg oce0,
    output reg ce0,
    output reg wre0,
    output reg [10:0] ad0,
    output reg [31:0] din0,
    // BSRAM fft1
    input [31:0] dout1,
    output reg oce1,
    output reg ce1,
    output reg wre1,
    output reg [10:0] ad1,
    output reg [31:0] din1,
    // BSRAM(prom) w
    input [15:0] dout_w,
    output reg oce_w,
    output reg ce_w,
    output reg [10:0] ad_w
```

```
);

localparam N = 1024;
localparam N2 = N / 2;
localparam N4 = N / 4;
localparam N4_2 = N4 * 2;
localparam N4_3 = N4 * 3;

reg [15:0] w_re;

// x の一次保管用。
// NOTE: 同じ変数を再利用しまくってるので、可読性ゴミなので注意
reg [15:0] x0_re;
reg [15:0] x0_im;
reg [15:0] x1_re;
reg [15:0] x1_im;

reg [9:0] step;
reg [9:0] half_step;
reg [9:0] index;
reg [9:0] i;
reg [9:0] j;
reg [9:0] k;
reg [10:0] prom_i_im;
reg w_re_sign;
reg w_im_sign;
// x0 と x2、x1 と x3 のインデックスは今回の SRAM の構成では同じ(N/2 ずれているため)
reg [9:0] x_index;
reg [9:0] x_half_step_index;

localparam S_IDLE = 2'd0;
localparam S_BUTTERFLY2 = 2'd1;
localparam S_BUTTERFLY1 = 2'd2;
// S_FINISH がある理由はメモリの書き込みが終わるのを待つため。
localparam S_FINISH = 2'd3;

reg [1:0] state;
reg [1:0] next_state;
reg [2:0] clk_cnt;
```

```
wire [63:0] butterfly0_res;
wire [63:0] butterfly1_res;
wire [31:0] butterfly_dout0;

wire [23:0] fft_twindle_factor_index_res;

assign butterfly_dout0 = (state == S_BUTTERFLY1) ? dout1 : dout0;

fft1024_twindle_factor_index fft_twindle_factor_index_instance
(
    i,
    fft_twindle_factor_index_res
);


butterfly butterfly0
(
    x0_re,
    x0_im,
    // state == S_BUTTERFLY2 のときは dout0
    // state == S_BUTTERFLY1 のときは dout1
    // に接続する
    butterfly_dout0[31:16],
    butterfly_dout0[15:0],
    w_re,
    w_re_sign,
    dout_w,
    w_im_sign,
    butterfly0_res[63:48],
    butterfly0_res[47:32],
    butterfly0_res[31:16],
    butterfly0_res[15:0]
);

butterfly butterfly1
(
    x1_re,
    x1_im,
```

```
    dout1[31:16],
    dout1[15:0],
    w_re,
    w_re_sign,
    dout_w,
    w_im_sign,
    butterfly1_res[63:48],
    butterfly1_res[47:32],
    butterfly1_res[31:16],
    butterfly1_res[15:0]
);

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        finish <= 1'd0;
        oce0 <= 1'd0;
        ce0 <= 1'd0;
        wre0 <= 1'd0;
        ad0 <= 11'd0;
        din0 <= 32'd0;
        oce1 <= 1'd0;
        ce1 <= 1'd0;
        wre1 <= 1'd0;
        ad1 <= 11'd0;
        din1 <= 32'd0;
        oce_w <= 1'd0;
        ce_w <= 1'd0;
        ad_w <= 11'd0;

        w_re <= 16'd0;
        x0_re <= 16'd0;
        x0_im <= 16'd0;
        x1_re <= 16'd0;
        x1_im <= 16'd0;
        step <= 10'd0;
        half_step <= 10'd0;
        index <= 10'd0;
        i <= 10'd0;
        j <= 10'd0;
```

```
        k <= 10'd0;
        prom_i_im <= 11'd0;
        w_re_sign <= 1'd0;
        w_im_sign <= 1'd0;
        x_index <= 10'd0;
        x_half_step_index <= 10'd0;
        state <= S_IDLE;
        next_state <= S_IDLE;
        clk_cnt <= 3'd0;
    end
    else begin
        if (clear == 1'd1 && state != S_FINISH) begin
            finish <= 1'd0;
        end
        case (state)
            S_IDLE: begin
                if (start == 1'd1) begin
                    ce_w <= 1'd1;
                    oce_w <= 1'd1;
                    // 他の SRAM もいつでも使用可能にしておく
                    ce0 <= 1'd1;
                    oce0 <= 1'd1;
                    wre0 <= 1'd0;
                    ad0 <= 11'd0;
                    ce1 <= 1'd1;
                    oce1 <= 1'd1;
                    wre1 <= 1'd0;
                    ad1 <= 11'd0;

                    half_step <= 10'd1;
                    step <= 10'd2;
                    index <= N2;
                    i <= 10'd0;
                    j <= 10'd0;
                    k <= 10'd0;
                    state <= S_BUTTERFLY2;
                    next_state <= S_BUTTERFLY2;
                    clk_cnt <= 3'd0;
```

```
                    w_re_sign <= fft_twindle_factor_index_res[23];
                    ad_w <= fft_twindle_factor_index_res[22:12];
                    w_im_sign <= fft_twindle_factor_index_res[11];
                    prom_i_im <= fft_twindle_factor_index_res[10:0];
                end
            end
            // step <= N / 2 のときは 2 つのバタフライ演算器を用いて計算
            S_BUTTERFLY2: begin
                case (clk_cnt)
                    3'd0: begin
                        x_index <= k + j;
                        x_half_step_index <= k + j + half_step;
                        if (j == half_step - 1'd1) begin
                            j <= 10'd0;
                            i <= 10'd0;
                            if (k == N2 - step) begin
                                k <= 10'd0;
                                half_step <= step;
                                step <= step << 1;
                                index <= index >> 1;
                                next_state <= (step == N2) ? S_BUTTERFLY1 :
S_BUTTERFLY2;
                            end
                            else begin
                                k <= k + step;
                            end
                        end
                        else begin
                            j <= j + 1'd1;
                            i <= i + index;
                        end

                        // read
                        wre0 <= 1'd0;
                        wre1 <= 1'd0;
                        // x0
                        ad0 <= {1'd0, k + j};
                        // x2
                        ad1 <= {1'd0, k + j};
```

```
                        // 回転因子のインデックスと符号を計算
                        // {w_re_sign, ad_w, w_im_sign, prom_i_im} <=
fft_twindle_factor_index_res;
                        w_re_sign <= fft_twindle_factor_index_res[23];
                        ad_w <= fft_twindle_factor_index_res[22:12];
                        w_im_sign <= fft_twindle_factor_index_res[11];
                        prom_i_im <= fft_twindle_factor_index_res[10:0];
                        clk_cnt <= 3'd1;
                    end
                    3'd1: begin
                        // read
                        // x1
                        ad0 <= {1'd0, x_half_step_index};
                        // x3
                        ad1 <= {1'd0, x_half_step_index};
                        // w_im
                        ad_w <= prom_i_im;
                        clk_cnt <= 3'd2;
                    end
                    3'd2: begin
                        // 代入
                        // x0
                        x0_re <= dout0[31:16];
                        x0_im <= dout0[15:0];
                        // x2
                        x1_re <= dout1[31:16];
                        x1_im <= dout1[15:0];
                        // w_re
                        w_re <= dout_w;
                        clk_cnt <= 3'd3;
                    end
                    3'd3: begin
                        // バタフライ演算を行い、
                        // din0[31:16] = x0_re
                        // din0[15:0] = x0_im
                        // x0_re = x1_re
                        // x0_im = x1_im
                        // をノンブロッキング代入。butterfly2 も同様。
                        din0 <= butterfly0_res[63:32];
```

```
                x0_re <= butterfly0_res[31:16];
                x0_im <= butterfly0_res[15:0];
                din1 <= butterfly1_res[63:32];
                x1_re <= butterfly1_res[31:16];
                x1_im <= butterfly1_res[15:0];
                // write
                wre0 <= 1'd1;
                wre1 <= 1'd1;
                // x0 = x0 + x1 * w
                ad0 <= {1'd0, x_index};

                // x2 = x2 + x3 * w
                ad1 <= {1'd0, x_index};

                clk_cnt <= 3'd4;
            end
            3'd4: begin
                // write
                // x1 = x0 -  x1 * w
                // x1
                ad0 <= {1'd0, x_half_step_index};
                din0 <= {x0_re, x0_im};
                // x3 = x2 -  x3 * w
                // x3
                ad1 <= {1'd0, x_half_step_index};
                din1 <= {x1_re, x1_im};

                state <= next_state;
                if (next_state == S_BUTTERFLY1) begin
                    half_step <= N2;
                end
                clk_cnt <= 3'd0;
            end
        endcase
    end
    // step == N のときは 1 つのバタフライ演算器を用いて計算
    S_BUTTERFLY1: begin
        case (clk_cnt)
            3'd0: begin
```

```
                        // half step ずれたインデックスを計算
                        x_index <= j;
                        x_half_step_index <= j + half_step;
                        if (j == half_step - 1'd1) begin
                            next_state <= S_FINISH;
                        end
                        else begin
                            j <= j + 1'd1;
                            // index=1'd1
                            i <= i + 1'd1;
                        end

                        // read
                        wre0 <= 1'd0;
                        wre1 <= 1'd0;
                        // x0
                        ad0 <= {1'd0, j};
                        // 回転因子のインデックスと符号を計算
                        // {w_re_sign, ad_w, w_im_sign, prom_i_im} <=
fft_twindle_factor_index_res;
                        w_re_sign <= fft_twindle_factor_index_res[23];
                        ad_w <= fft_twindle_factor_index_res[22:12];
                        w_im_sign <= fft_twindle_factor_index_res[11];
                        prom_i_im <= fft_twindle_factor_index_res[10:0];
                        clk_cnt <= 3'd1;
                    end
                    3'd1: begin
                        // read
                        // x1
                        ad1 <= {1'd0, x_index};
                        // w_im
                        ad_w <= prom_i_im;
                        clk_cnt <= 3'd2;
                    end
                    3'd2: begin
                        // 代入
                        // x0
                        x0_re <= dout0[31:16];
                        x0_im <= dout0[15:0];
```

```
                    // w_re
                    w_re <= dout_w;
                    clk_cnt <= 3'd3;
                end
                3'd3: begin
                    // この時点でバタフライ演算を行い
                    // din0[31:16] = x0_re
                    // din0[15:0] = x0_im
                    // x0_re = x1_re
                    // x0_im = x1_im
                    // を代入する
                    // state == S_BUTTERFLY1 のとき、butterfly_dout0=dout1 なので
注意
                    din0 <= butterfly0_res[63:32];
                    x0_re <= butterfly0_res[31:16];
                    x0_im <= butterfly0_res[15:0];
                    // write
                    // x0 = x0 + x1 * w
                    wre0 <= 1'd1;
                    wre1 <= 1'd0;
                    ad0 <= {1'd0, x_index};
                    clk_cnt <= 3'd4;
                end
                3'd4: begin
                    // write
                    // x1 = x0 -  x1 * w
                    // x1
                    wre0 <= 1'd0;
                    wre1 <= 1'd1;
                    ad1 <= {1'd0, x_index};
                    din1 <= {x0_re, x0_im};
                    state <= next_state;
                    clk_cnt <= 3'd0;
                end
            endcase
        end
        // S_FINISH がある理由はメモリの書き込みが終わるのを待つため。
        S_FINISH: begin
            case (clk_cnt)
```

```
                    3'd0: begin
                        // 念の為、SRAM 関連の変数は 0 にしておく
                        ce_w <= 1'd0;
                        oce_w <= 1'd0;
                        ad_w <= 11'd0;
                        ce0 <= 1'd0;
                        oce0 <= 1'd0;
                        wre0 <= 1'd0;
                        ad0 <= 11'd0;
                        ce1 <= 1'd0;
                        oce1 <= 1'd0;
                        wre1 <= 1'd0;
                        ad1 <= 11'd0;
                        clk_cnt <= 3'd1;
                        // twindle index の関係で、i は 0 に戻しておく
                        i <= 10'd0;
                    end
                    3'd1: begin
                        finish <= 1'd1;
                        state <= S_IDLE;
                        next_state <= S_IDLE;
                    end
                endcase
            end
        endcase
    end
end
endmodule
```

付録 E　プログラム公開リンク

使用したプログラムは GitHub 上に公開している。プログラム公開リンクは次の URL に示す。

https://github.com/yudai0804/sotsuken/tree/master