OpenOFDM Documentation

Release 1.0

Jinghao Shi

Apr 01, 2022

Contents:

OpenOFDM is a open source Verilog implementation of 802.11 OFDM decoder. Highlights are:

- Supports 802.11a/g (all bit rates) and 802.11n (20MHz BW, MCS 0 - 7)
- Modular design, easy to extend
- Fully synthesizable, tested on USRP N210

CHAPTER 1

Overview

Once the RF signals are captured and down-converted to baseband, the decoding pipeline starts, including:

1. Packet detection
2. Center frequency offset correction
3. FFT
4. Channel gain estimation
5. Demodulation
6. Deinterleaving
7. Convolutional decoding
8. Descrambling

This documentation walks through the decoding pipeline and explains how each step is implemented in OpenOFDM.

1.1 Top Level Module

The top level module of OpenOFDM is `dot11.v`. Fig. 1.1 shows its input/output pins. It takes I/Q samples as input, and output 802.11 packet data bytes and various PHY properties.

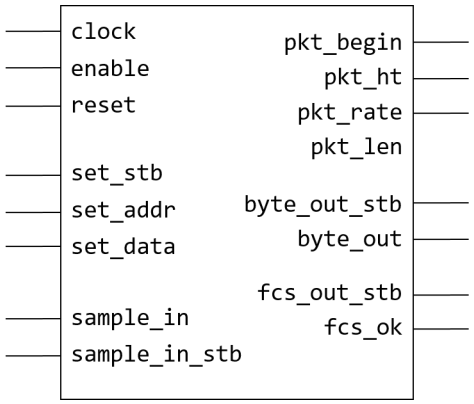

Fig. 1.1: Dot11 Core Schematic Symbol

Table 1.1: Dot11 Module Pinout

Port Name	Port Width	Direction	Description
clock	1	Input	Rising edge clock
enable	1	Input	Module enable (active high)
reset	1	Input	Module reset (active high)
set_stb	1	Input	Setting register strobe
set_addr	8	Input	Setting register address
set_data	32	Input	Setting register value
sample_in	32	Input	High 16 bit I, low 16 bit Q
sample_in_stb	1	Input	Sample input strobe
pkt_begin	1	Output	Signal begin of a packet
pkt_ht	1	Output	HT (802.11n) or legacy (802.11a/g) packet
pkt_rate	8	Output	For HT, the lower 7 bits is MCS. For legacy, the lower 4 bits is the rate bits in SIGNAL
pkt_len	16	Output	Packet length in bytes
byte_out_stb	1	Output	Byte out strobe
byte_out	8	Output	Byte value
fcs_out_stb	1	Output	FCS output strobe
fcs_ok	1	Output	FCS correct (high) or wrong (low)

1.2 Project Structure

In the `verilog` sub-directory, you will find the Verilog implementations of various modules. The implementations were originally targeted for the Xilinx Spartan 3A-DSP 3400 FPGA inside the USRP N210 device, thus there are various dependences to Xilinx libraries and USRP code base. In particular:

- `verilog/Xilinx` contains the Xilinx specific libraries
- `verilog/coregen` contains generated IP cores from Xilinx ISE
- `verilog/usrp2` contains USRP specific modules

However, the project is self-contained and is ready for simulation using Icarus Verilog tool chain, including `iverilog` and `vvp`.

The `scripts` directory contains various Python scripts that:

- Generate look up tables (`gen_atan_lut.py`, `gen_rot_lut.py`, `gen_deinter_lut.py`)
- Convert binary I/Q file into text format so it can be read in Verilog using `readmemh`.
- Consolidate sample files by removing *silent* signals (`condense.py`).
- Test each step of decoding process (`test.py`)
- 802.11 decoder in Python for cross validation (`decode.py`)

It also contains a modified copy of the CommPy library.

The `test.py` script is for cross validation between the Python decoder and OpenOFDM decoder. It first uses the `decode.py` script to decode the sample file and stores the expected output of each step. It then performs Verilog simulation using `vvp` and compare the Verilog output against the expected output step by step.

The `testing_inputs` directory contains various sample files collected in a conducted or over the air setup. These files covers all the bit rates (legacy and HT) supported in OpenOFDM.

1.3 Sample File

Throughout this documentation we will be using a sample file that contains the I/Q samples of a 802.11a packet at 24 Mbps (16-QAM). It'll be helpful to use a interactive iPython session and exercise various steps discussed in the document.

Download the sample file from `here`, the data can be loaded as follows:

```
import scipy

wave = scipy.fromfile('samples.dat', dtype=scipy.int16)
samples = [complex(i, q) for i, q in zip(wave[::2], wave[1::2])]
```

CHAPTER 2

Packet Detection

802.11 OFDM packets start with a short PLCP Preamble sequence to help the receiver detect the beginning of the packet. The short preamble duration is 8 us. At 20 MSPS sampling rate, it contains 10 repeating sequence of 16 I/Q samples, or 160 samples in total. The short preamble also helps the receiver for coarse frequency offset correction , which will be discussed separately in *Frequency Offset Correction*.

2.1 Power Trigger

- **Module**: `power_trigger.v`
- **Input**: `sample_in` (16B I + 16B Q), `sample_in_strobe` (1B)
- **Output**: `trigger` (1B)
- **Setting Registers**: `SR_POWER_THRES`, `SR_POWER_WINDOW`, `SR_SKIP_SAMPLE`.

The core idea of detecting the short preamble is to utilize its repeating nature by calculating the auto correlation metric. But before that, we need to make sure we are trying to detect short preamble from "meaningful" signals. One example of "un-meaningful" signal is constant power levels, whose auto correlation metric is also very high (nearly 1) but obviously does not represent packet beginning.

The first module in the pipeline is the `power_trigger.v`. It takes the I/Q samples as input and asserts the `trigger` signal during a potential packet activity. Optionally, it can be configured to skip the first certain number of samples before detecting a power trigger. This is useful to skip the spurious signals during the initial hardware stabilization phase.

The logic of the `power_trigger` module is quite simple: after skipping certain number of initial samples, it waits for significant power increase and triggers the `trigger` signal upon detection. The `trigger` signal is asserted until the power level is smaller than a threshold for certain number of continuous samples.

2.2 Short Preamble Detection

- **Module**: `sync_short.v`

- **Input**: `sample_in` (16B I + 16B Q), `sample_in_strobe` (1B)
- **Output**: `short_preamble_detected` (1B)
- **Setting Registers**: `SR_MIN_PLATEAU`

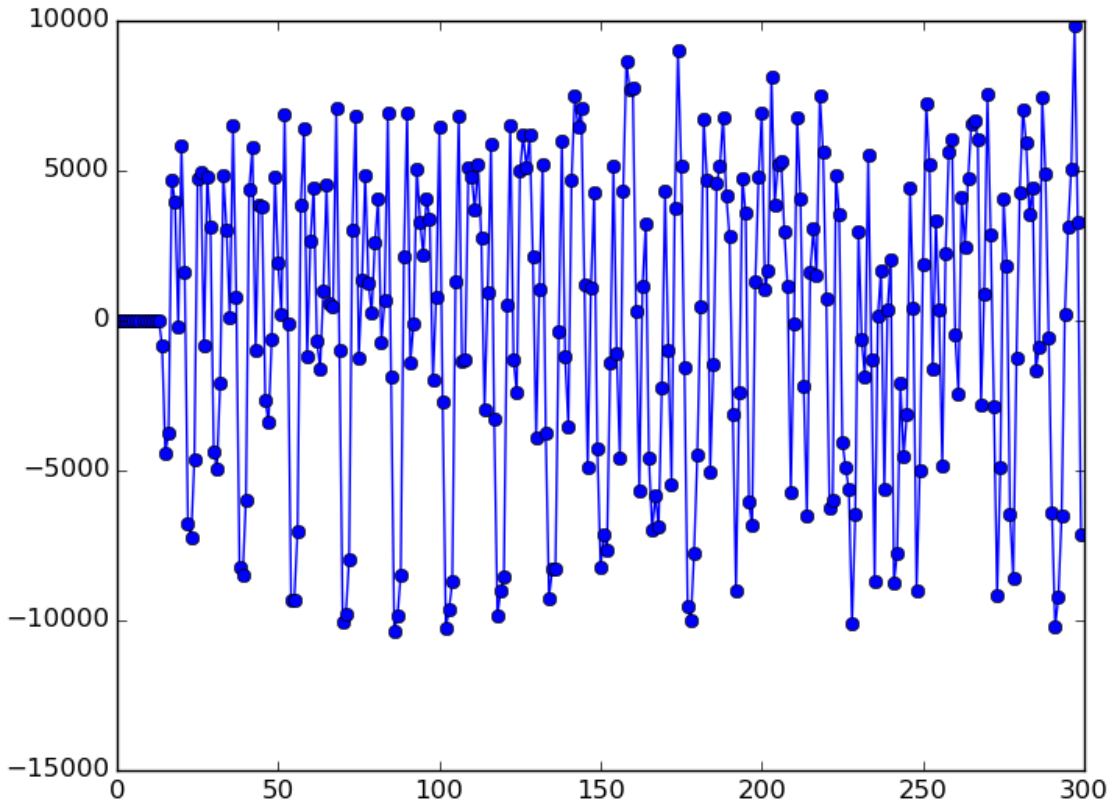

Fig. 2.1: In-Phase of Short Preamble.

Fig. 2.1 shows the in-phase of the beginning of a packet. Some repeating patterns can clearly be seen. We can utilize this characteristic and calculate the auto correlation metric of incoming signals to detect such pattern:

$$corr[i] = \frac{\left\| \sum_{i=0}^{N} S[i] * \overline{S[i+16]} \right\|}{\sum_{i=0}^{N} S[i] * \overline{S[i]}} \tag{2.1}$$

where $S[i]$ is the $\langle I, Q \rangle$ sample expressed as a complex number, and $\overline{S[i]}$ is its conjugate, N is the correlation window size. The correlation reaches 1 if the incoming signal is repeating itself every 16 samples. If the correlation stays high for certain number of continuous samples, then a short preamble can be declared.

To plot Fig. 2.2, load the samples (see *Sample File*), then:

```
from matplotlib import pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
ax[0].plot([s.real for s in samples[:500]], '-bo')
ax[1].plot([abs(sum([samples[i+j]*samples[i+j+16].conjugate()
```

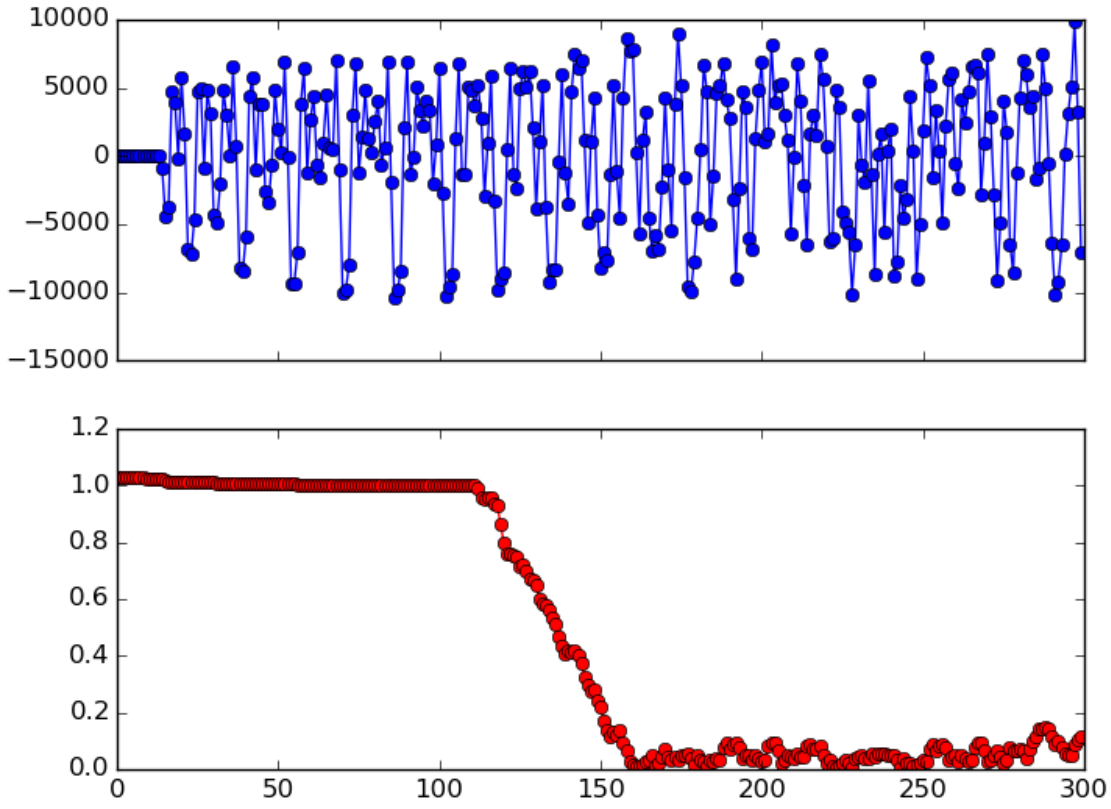

Fig. 2.2: Auto Correlation of the Short Preamble samples (N=48).

```
    for j in range(0, 48)]))/
    sum([abs(samples[i+j])**2 for j in range(0, 48)])
    for i in range(0, 500)], '-ro')
plt.show()
```

Fig. 2.2 shows the auto correlation value of the samples in Fig. 2.1. We can see that the correlation value is almost 1 during the short preamble period, but drops quickly after that. We can also see that for the very first 20 samples or so, the correlation value is also very high. This is because the silence also repeats itself (at arbitrary interval)! That's why we first use the `power_trigger` module to detect actual packet activity and only perform short preamble detection on non-silent samples.

A straight forward implementation would require both multiplication and division. However, on FPGAs devision consumes a lot of resources so we really want to avoid it. In current implementation, we use a fixed threshold (0.75) for the correlation so that we can use bit-shift to achieve the purpose. In particular, we calculate `numerator>>1 + numerator>>2` and compare that with the denominator. For the correlation window size, we set $N = 16$.

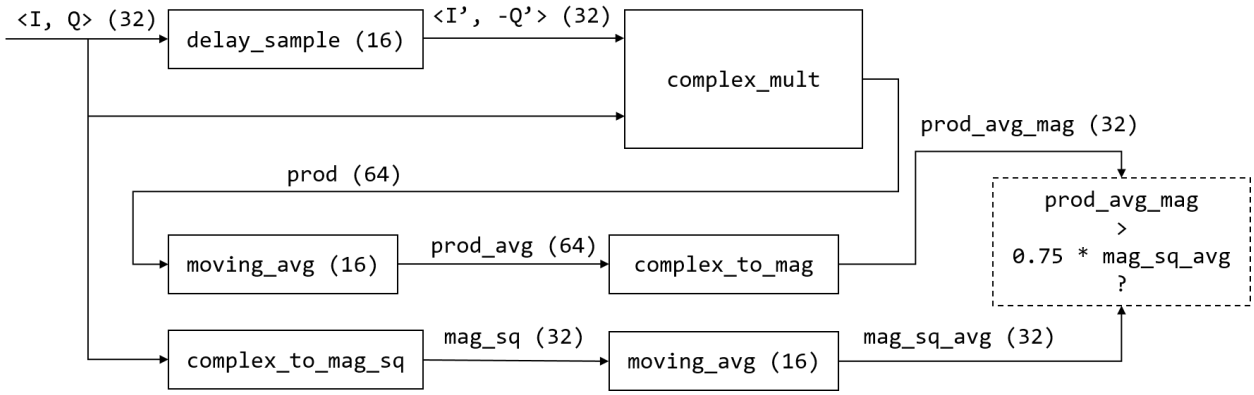

Fig. 2.3: `sync_short` Module Diagram

Fig. 2.3 shows the internal module diagram of the `sync_short` module. In addition to the number of consecutive samples with correlation larger than 0.75, the `sync_short` module also checks if the incoming signal has both positive (> 25%) and negative (> 25%) samples to further eliminate false positives (e.g., when the incoming signals are constant non-zero values). Again, the thresholds (25%) are chosen so that we can use only bit-shifts for the calculation.

CHAPTER 3

Frequency Offset Correction

`This paper`[1] explains why frequency offset occurs and how to correct it. In a nutshell, there are two types of frequency offsets. The first is called **Carrier Frequency Offset (CFO)** and is caused by the difference between the transmitter and receiver's Local Oscillator (LO). This symptom of this offset is a phase rotation of incoming I/Q samples (time domain). The second is **Sampling Frequency Offset (SFO)** and is caused by the sampling effect. The symptom of this offset is a phase rotation of constellation points after FFT (frequency domain).

The CFO can be corrected with the help of short preamble (Coarse) long preamble (Fine). And the SFO can be corrected using the pilot sub-carriers in each OFDM symbol. Before we get into how exactly the correction is done. Let's see visually how each correction step helps in the final constellation plane.

Fig. 3.1 to Fig. 3.4 shows the constellation points of a 16-QAM modulated 802.11a packet.

3.1 Coarse CFO Correction

The coarse CFO can be estimated using the short preamble as follows:

$$\alpha_{ST} = \frac{1}{16} \angle \left(\sum_{i=0}^{N-1} \overline{S[i]} S[i+16] \right) \tag{3.1}$$

where $\angle(\cdot)$ is the phase of complex number and $N \leq 144(160-16)$ is the subset of short preambles utilized. The intuition is that the phase difference between S[i] and S[i+16] represents the accumulated CFO over 16 samples.

After getting α_{ST}, each following I/Q samples (starting from long preamble) are corrected as:

$$S'[m] = S[m] e^{-jm\alpha_{ST}}, m = 0, 1, 2, \ldots \tag{3.2}$$

In OpenOFDM, the coarse CFO is calculated in the `sync_short` module, and we set $N = 64$. The `prod_avg` in Fig. 2.3 is fed into a `moving_avg` module with window size set to 64.

[1] Sourour, Essam, Hussein El-Ghoroury, and Dale McNeill. "Frequency Offset Estimation and Correction in the IEEE 802.11 a WLAN." Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th. Vol. 7. IEEE, 2004.

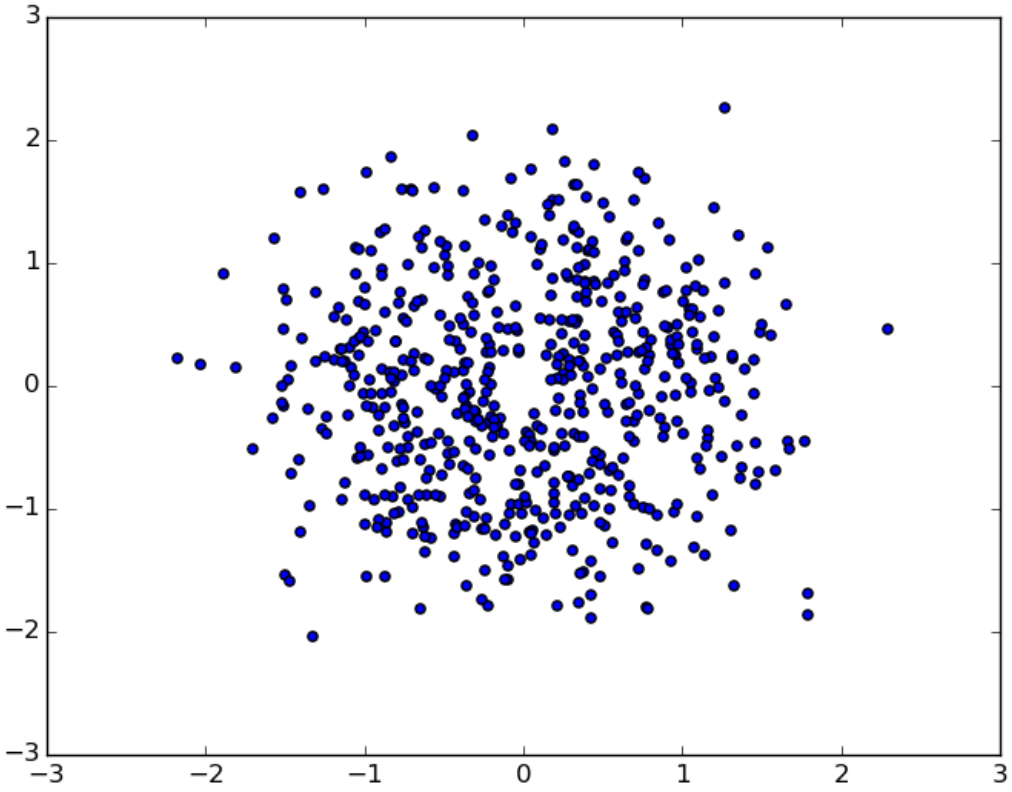

Fig. 3.1: Constellation Points Without Any Correction

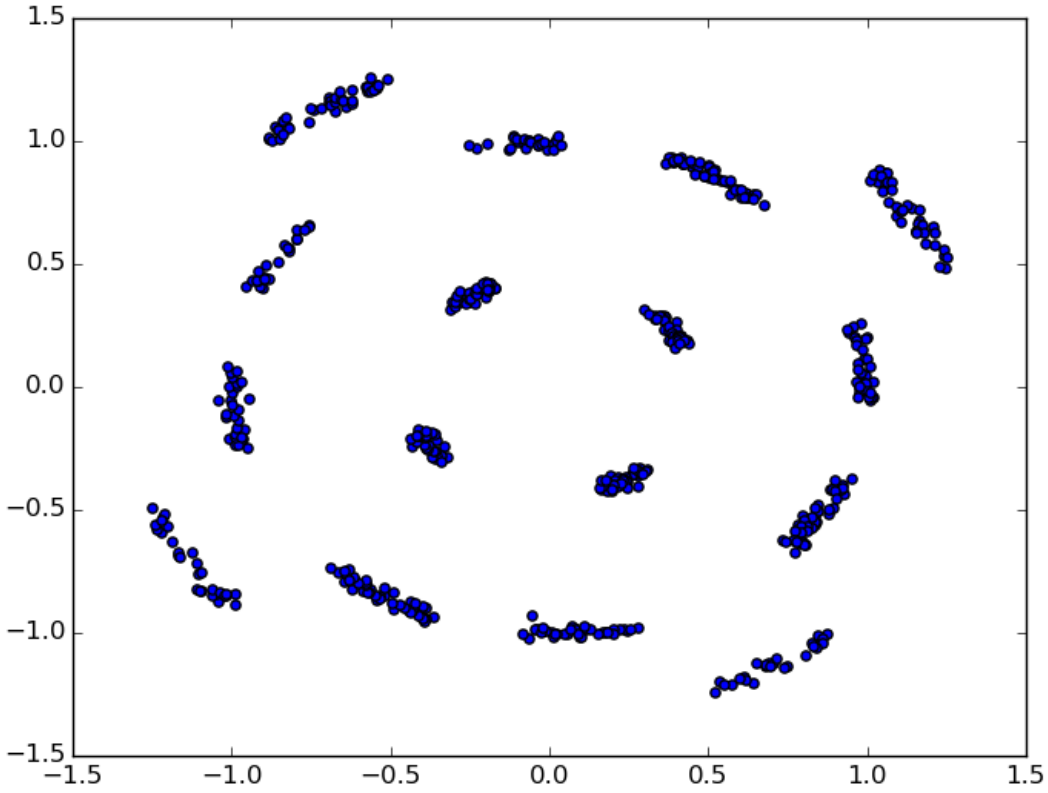

Fig. 3.2: Constellation Points With Only Coarse Correction

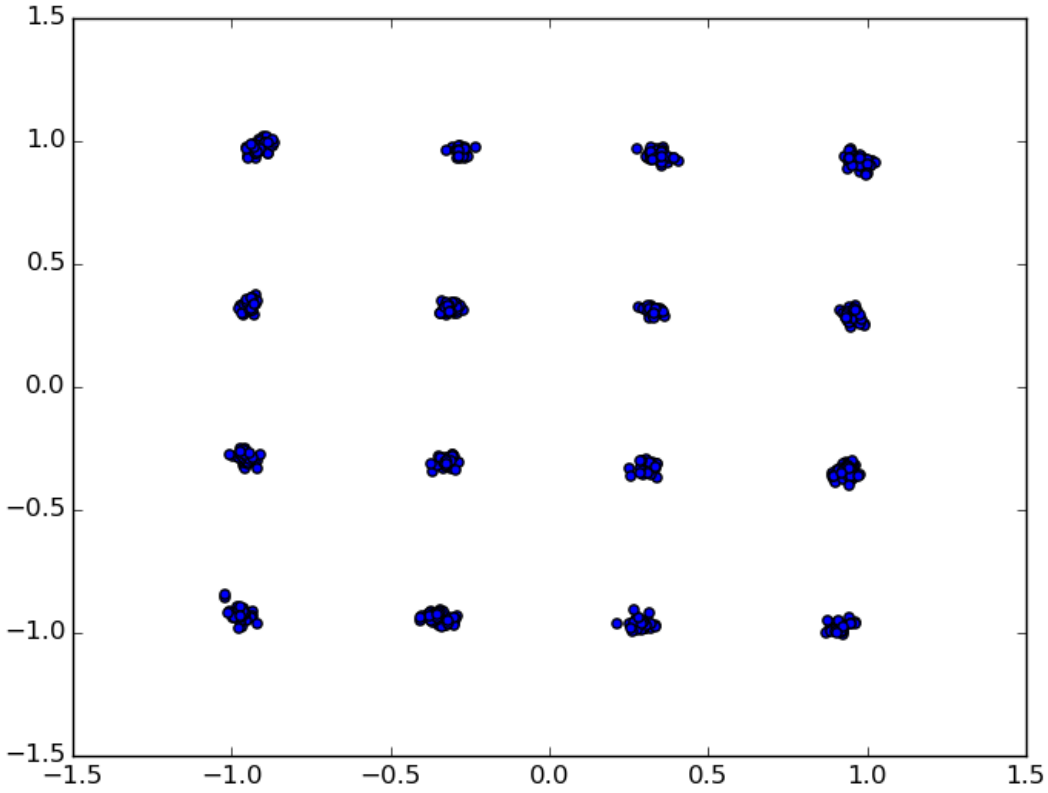

Fig. 3.3: Constellation Points With both Coarse and Fine Correction

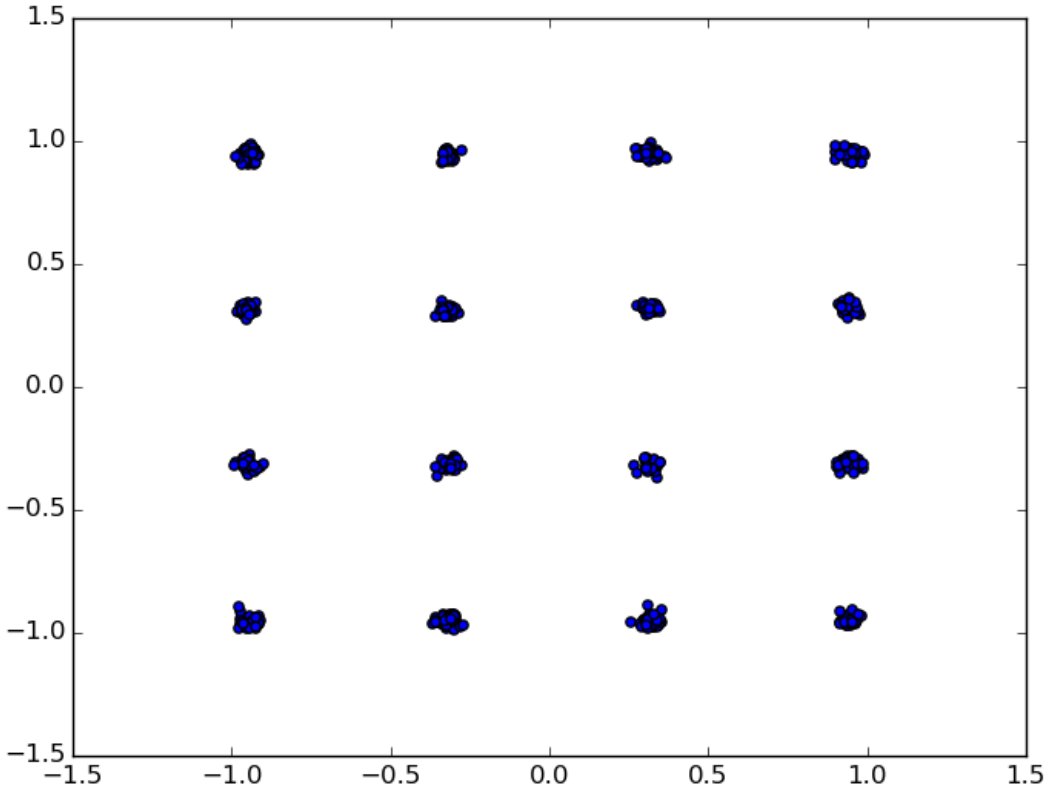

Fig. 3.4: Constellation Points With Coarse, Fine and Pilot Correction

3.2 Fine CFO Correction

A finer estimation of the CFO can be obtained with the help of long training sequence inside the long preamble.

The long preamble contains two identify training sequence (64 samples each at 20 MSPS), the phase offset can be calculated as:

$$\alpha_{LT} = \frac{1}{64} \angle \left(\sum_{i=0}^{63} \overline{S[i]} S[i+64] \right) \tag{3.3}$$

This step is omitted in OpenOFDM due to the limited resolution of phase estimation and rotation in the look up table.

CHAPTER 4

Symbol Alignment

- **Module**: `sync_long.v`
- **Input**: `I (16), Q (16), phase_offset (32), short_gi (1)`
- **Output**: `long_preamble_detected (1), fft_re (16), fft_im (16)`

After detecting the packet, the next step is to determine precisely where each OFDM symbol starts. In 802.11, each OFDM symbol is 4 μs long. At 20 MSPS sampling rate, this means each OFDM symbol contains 80 samples. The task is to group the incoming streaming of samples into 80-sample OFDM symbols. This can be achieved using the long preamble following the short preamble.

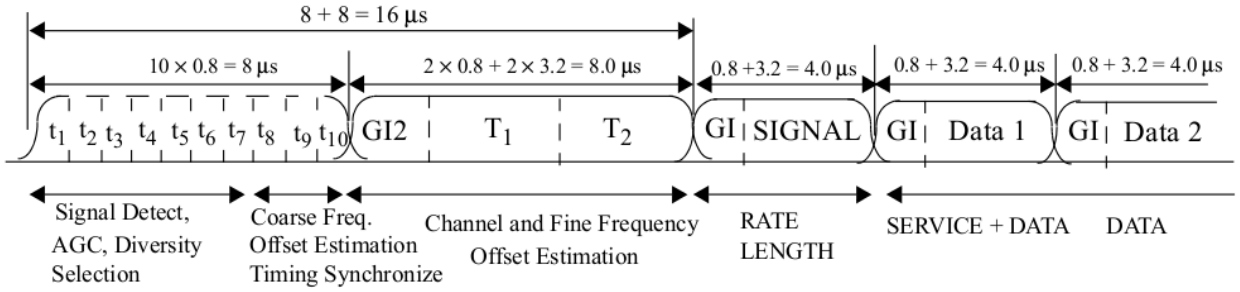

Fig. 4.1: 802.11 OFDM Packet Structure (Fig 18-4 in 802.11-2012 Std)

As shown in Fig. 4.1, the long preamble duration is 8 μs (160 samples), and contains two identical long training sequence (LTS), 64 samples each. The LTS is known and we can use cross correlation to find it.

The cross validation *score* at sample i can be calculated as follows.

$$Y[i] = \sum_{k=0}^{63} (S[i+k]\overline{H[k]}) \tag{4.1}$$

where H is the 64 sample known LTS in time domain, and can be found in Table L-6 in `802.11-2012 std` (index 96 to 159). A numpy readable file of the LTS (64 samples) can be found `here`, and can be read like this:

```
>>> import numpy as np
>>> lts = np.loadtxt('lts.txt').view(complex)
```

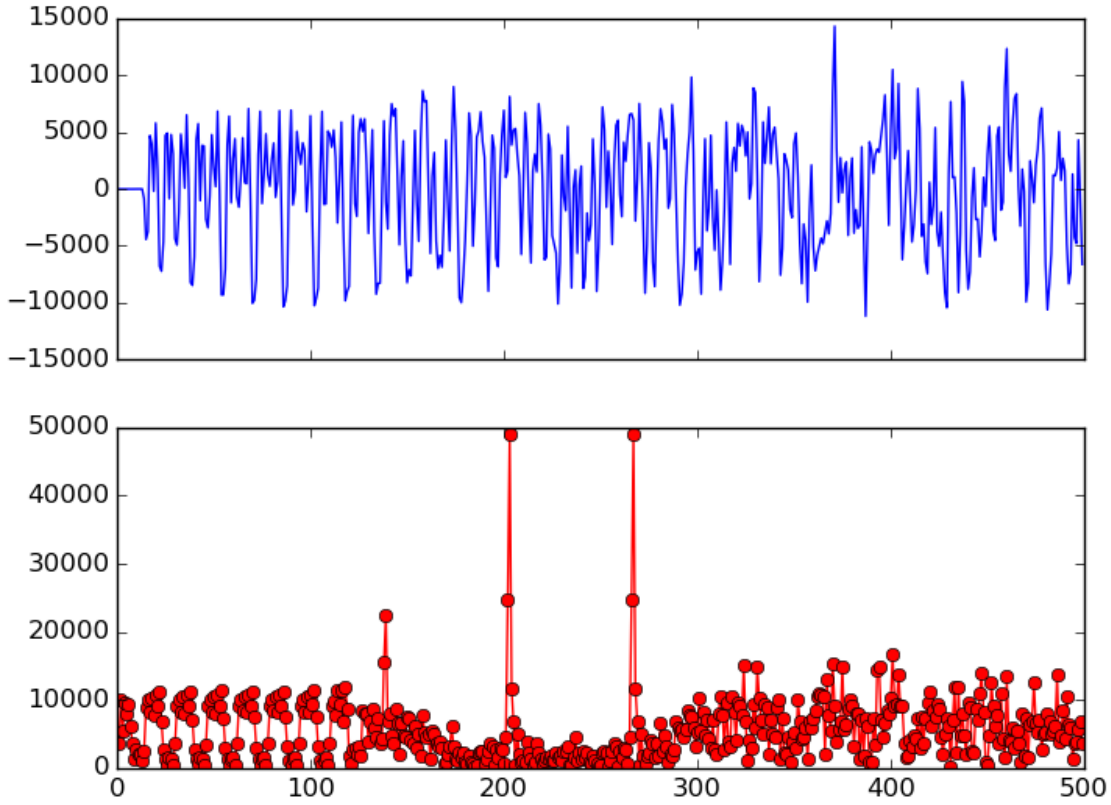

Fig. 4.2: Long Preamble and Cross Correlation Result

To plot Fig. 4.2, load the data file (see *Sample File*), then:

```
# in scripts/decode.py
import decode
import numpy as np
from matplotlib import pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
ax[0].plot([c.real for c in samples][:500])
# lts is from the above code snippet
ax[1].plot([abs(c) for c in np.correlate(samples, lts, mode='valid')][:500], '-ro')
plt.show()
```

Fig. 4.2 shows the long preamble samples and also the result of cross correlation. We can clearly see two spikes corresponding the two LTS in long preamble. And the spike width is only 1 sample which shows exactly the beginning of each sequence. Suppose the sample index if the first spike is N, then the 160 sample long preamble starts at sample $N - 32$.

This all seems nice and dandy, but as it comes to Verilog implementation, we have to make a compromise.

From (4.1) we can see for each sample, we need to perform 64 complex number multiplications, which would consume a lot FPGA resources. Therefore, we need to reduce the size of cross validation. The idea is to only use a portion instead of all the LTS samples.

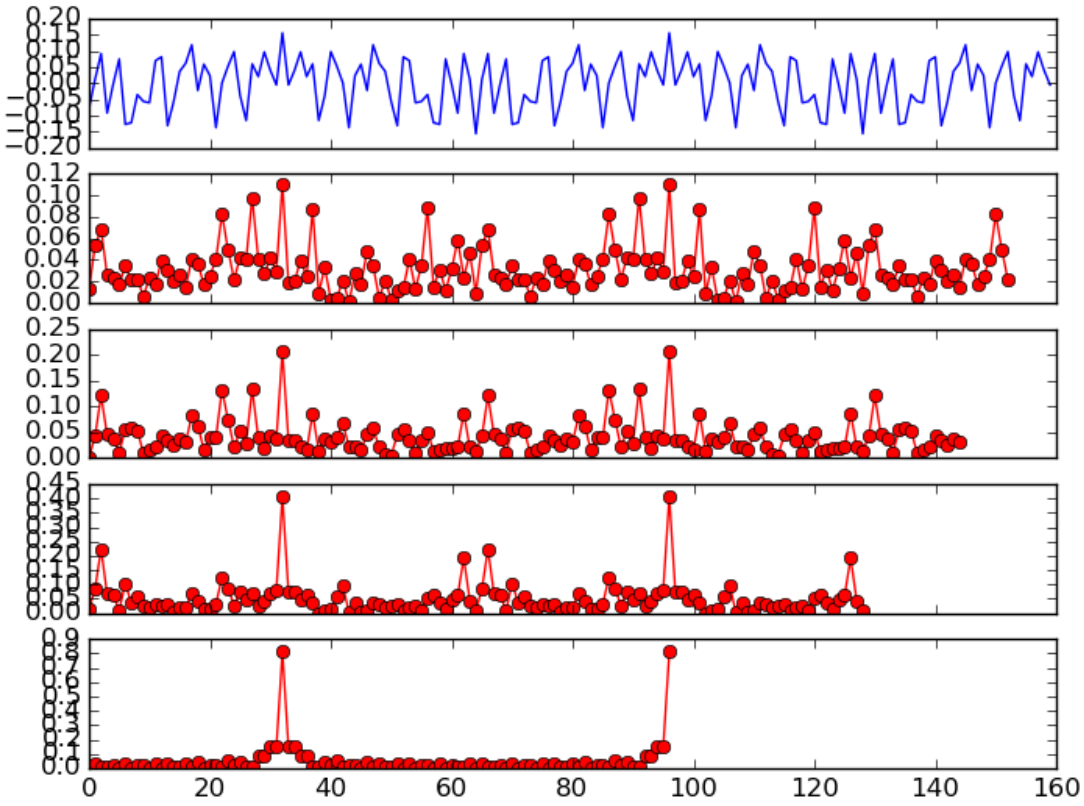

Fig. 4.3: Cross Correlation with Various Size (8, 16, 32, 64)

Fig. 4.3 can be plotted as:

```
lp = decode.LONG_PREAMBLE

fig, ax = plt.subplots(nrows=5, ncols=1, sharex=True)
ax[0].plot([c.real for c in lp])
ax[1].plot([abs(c) for c in np.correlate(lp, lts[:8], mode='valid')], '-ro')
ax[2].plot([abs(c) for c in np.correlate(lp, lts[:16], mode='valid')], '-ro')
ax[3].plot([abs(c) for c in np.correlate(lp, lts[:32], mode='valid')], '-ro');
ax[4].plot([abs(c) for c in np.correlate(lp, lts, mode='valid')], '-ro')
plt.show()
```

Fig. 4.3 shows the long preamble (160 samples) as well as cross validation with different size. It can be seen that using the first 16 samples of LTS is good enough to exhibit two narrow spikes. Therefore, OpenOFDM use cross correlation of first 16 samples of LTS for symbol alignment. To confirm, Fig. 4.4 shows the cross correlation of the first 16 samples of LTS on the actual packet. The two spikes are not as obvious as the ones in Fig. 4.2, but are still clearly visible.

To find the two spikes, we keep a record of the max correlation sample for the first 64 samples (since the first spike is

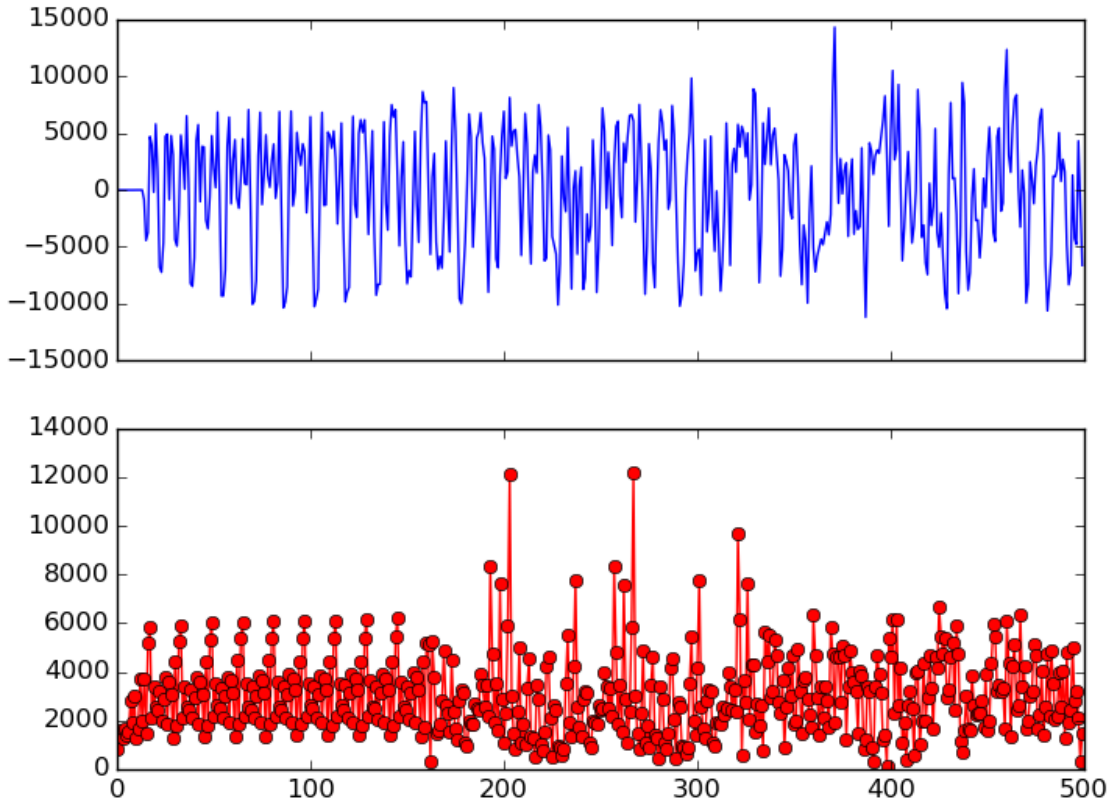

Fig. 4.4: Cross Validation using the First 16 Samples of LTS

supposed to be at the 32th sample). Similarly, we also keep a record of the max correlation sample for the second 64 samples. For further eliminate false positives, we also check if the two spike sample indexes are 64 ± 1 apart.

4.1 FFT

Now we have located the start of each OFDM symbol, the next task is to perform FFT on the last 64 data samples inside each symbol. For this we utilize the XFFT core generated by Xilinx ISE. Depend on if short guard interval (SGI) is used, the first 16 or 8 samples of each OFDM symbol need to be skipped.

But before performing FFT, we need to first apply the frequency offset correction (see *Frequency Offset Correction*). This is achieved via the `rotate` module (see *Rotation*).

CHAPTER 5

Sub-carrier Equalization and Pilot Correction

- **Module**: `equalizer.v`
- **Input**: `I (16), Q (16)`
- **Output**: `I (16), Q (16)`

This is the first module in frequency domain. There are two main tasks: sub-carrier gain equalization and correcting residue phase offset using the pilot sub-carriers.

5.1 Sub-carrier Structure

The basic channel width in 802.11a/g/n is 20 MHz, which is further divided into 64 sub-carriers (0.3125 MHz each).

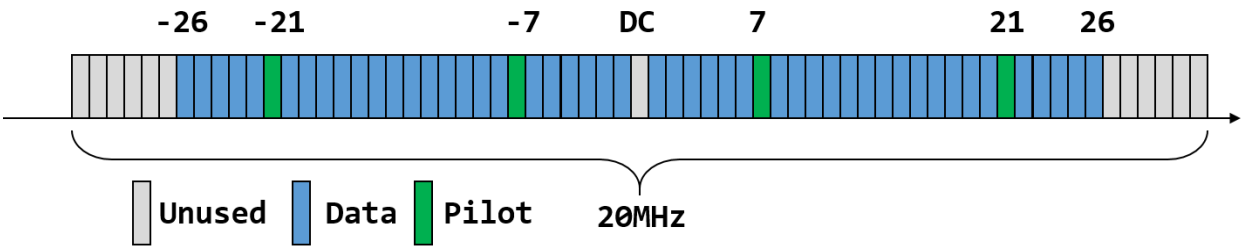

Fig. 5.1: Sub-carriers in 802.11 OFDM

Fig. 5.1 shows the sub-carrier structure of the 20 MHz band. 52 out of 64 sub-carriers are utilized, and 4 out of the 52 (-7, -21, 7, 21) sub-carriers are used as pilot sub-carrier and the remaining 48 sub-carriers carries data. As we will see later, the pilot sub-carriers can be used to correct the residue frequency offset.

Each sub-carrier carries I/Q modulated information, corresponding to the output of 64 point FFT from `sync_long.v` module.

5.2 Sub-Carrier Equalization

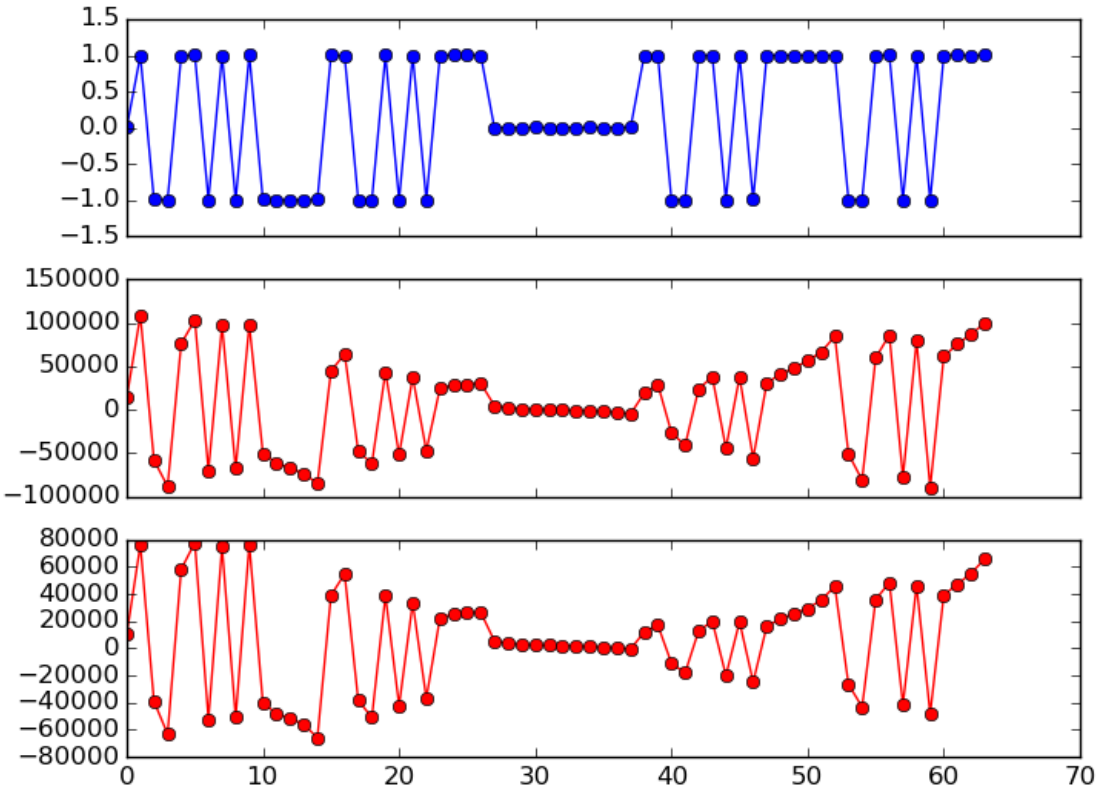

Fig. 5.2: FFT of the Perfect and Two Actual LTS

To plot Fig. 5.2:

```
lts1 = samples[11+160:][32:32+64]
lts2 = samples[11+160:][32+64:32+128]
fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True);
ax[0].plot([c.real for c in np.fft.fft(lts)], '-bo');
ax[1].plot([c.real for c in np.fft.fft(lts1)], '-ro');
ax[2].plot([c.real for c in np.fft.ff t(lts2)], '-ro');
plt.show()
```

Fig. 5.2 shows the FFT of the perfect LTS and the two actual LTSs in the samples. We can see that each sub-carrier exhibits different magnitude gain. In fact, they also have different phase drift. The combined effect of magnitude gain and phase drift (known as *channel gain*) can clearly be seen in the I/Q plane shown in Fig. 5.3.

To map the FFT point to constellation points, we need to compensate for the channel gain. This can be achieved by normalize the data OFDM symbols using the LTS. In particular, the mean of the two LTS is used as channel gain (H):

$$H[i] = \frac{1}{2}(LTS_1[i] + LTS_2[i]) \times L[i], i \in [-26, 26] \tag{5.1}$$

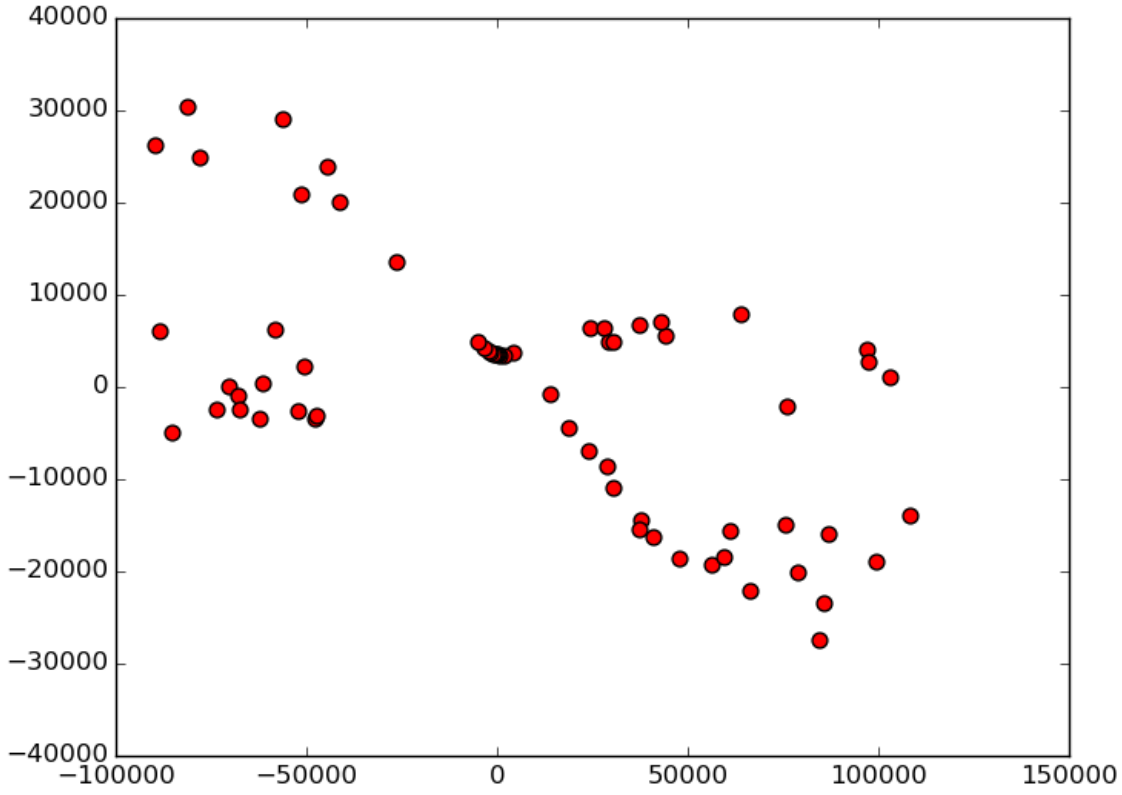

Fig. 5.3: FFT in I/Q Plane of The Actual LTS

where $L[i]$ is the sign of the LTS sequence:

$$\begin{aligned} L_{-26,26} = \{&1,1,-1,-1,1,1,-1,1,-1,1,1,1,1,1,1,-1,-1,1,\\ &1,-1,1,-1,1,1,1,1,0,1,-1,-1,1,1,-1,1,-1,1,\\ &-1,-1,-1,-1,-1,1,1,-1,-1,1,-1,1,-1,1,1,1,1\} \end{aligned} \tag{5.2}$$

And the FFT output at sub-carrier i is normalized as:

$$Y[i] = \frac{X[i]}{H[i]}, i \in [-26, 26] \tag{5.3}$$

where $X[i]$ is the FFT output at sub-carrier i.

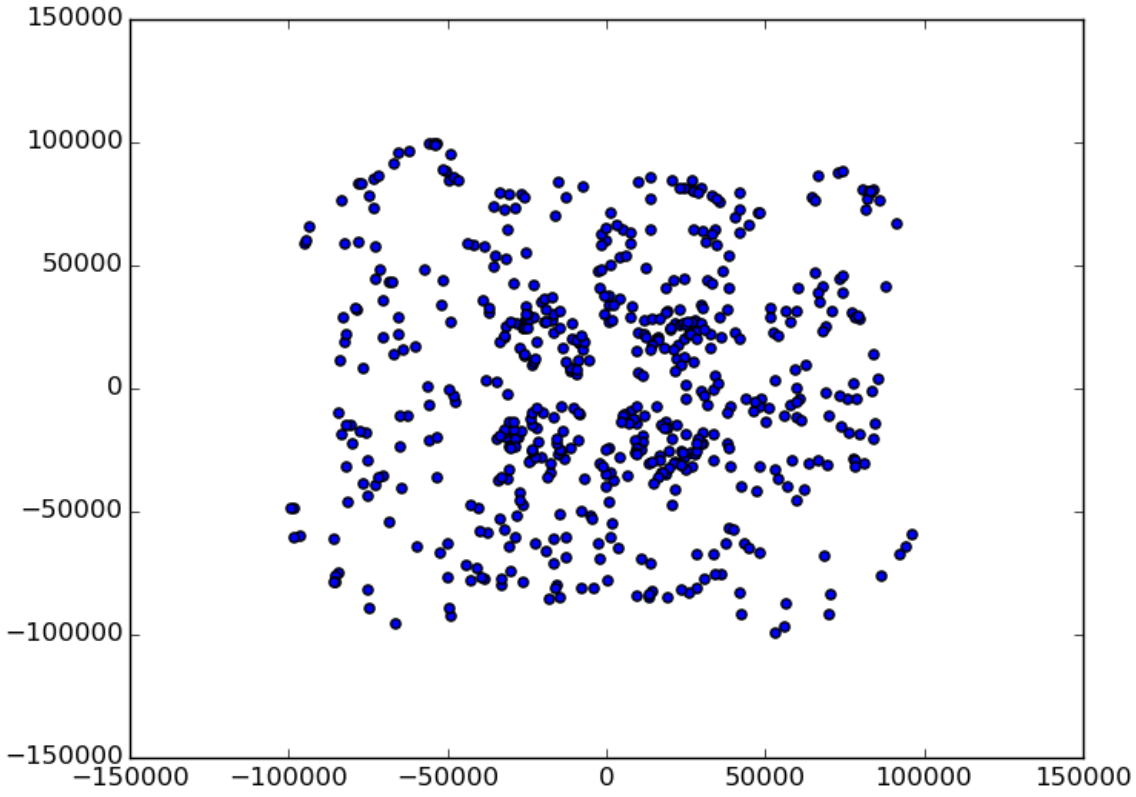

Fig. 5.4: FFT Without Normalization

Fig. 5.4 and Fig. 5.5 shows the FFT before and after normalization using channel gain.

5.3 Residual Frequency Offset Correction

We can see from Fig. 5.5 that the FFT output is tilted slightly. This is caused by residual frequency offset that was not compensated during the coarse CFO correction step.

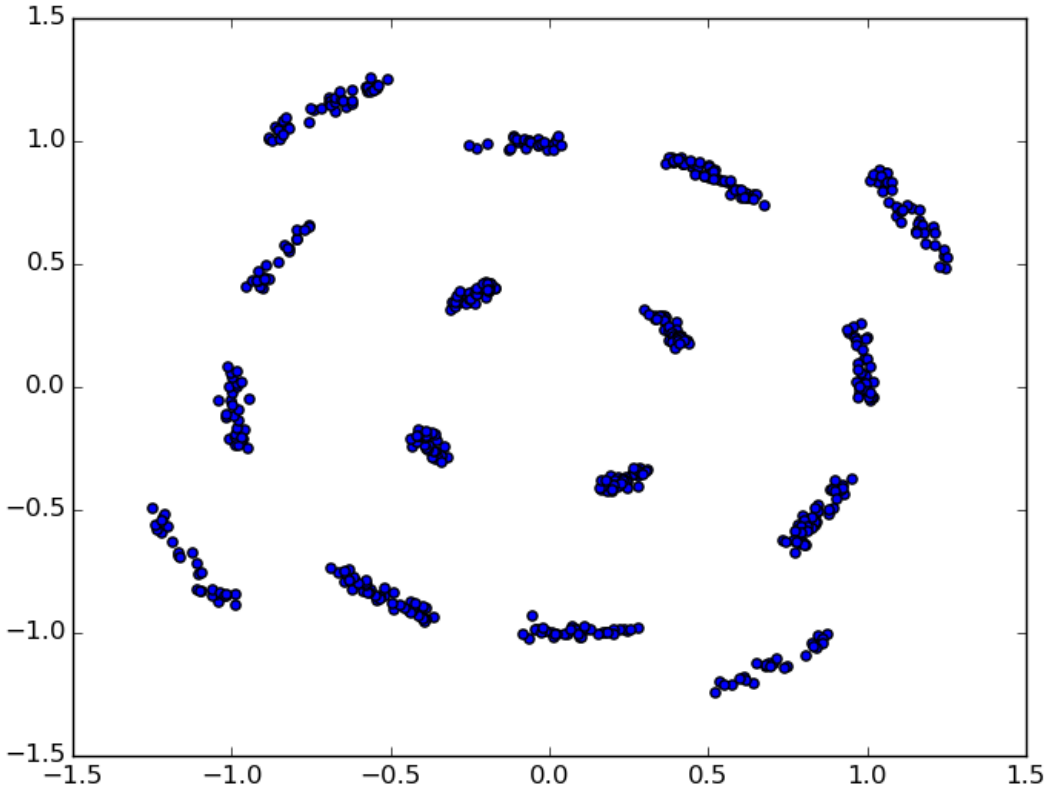

Fig. 5.5: FFT With Normalization

This residual CFO can be corrected either by *Fine CFO Correction*, or/and by the pilot sub-carriers. Ideally we want to do both, but since the fine CFO is usually beyond the resolution of the phase look up table, we skip it in the `sync_long.v` module and only rely on the pilot sub-carriers.

Regardless of the data sub-carrier modulation, the four pilot sub-carriers (-21, -7, 7, 21) always contains BPSK modulated pseudo-random binary sequence.

The polarity of the pilot sub-carriers varies symbol to symbol. For 802.11a/g, the pilot pattern is:

$$\begin{aligned} p_{0,\ldots,126} = \{&1,1,1,1,-1,-1,-1,1,-1,-1,-1,-1,1,1,-1,1,-1,-1,1,1,-1,1,1,-1,1,\\ &1,1,1,1,1,-1,1,1,1,-1,1,1,-1,-1,1,1,1,-1,1,-1,-1,-1,1,-1,\\ &1,-1,-1,1,-1,-1,1,1,1,1,1,-1,-1,1,1,-1,-1,1,-1,1,-1,1,\\ &1,-1,-1,-1,1,1,-1,-1,-1,-1,1,-1,-1,1,-1,1,1,1,1,-1,1,-1,1,-1,\\ &1,-1,-1,-1,-1,-1,1,-1,1,1,-1,1,-1,1,1,1,-1,-1,1,-1,-1,-1,1,1,\\ &1,-1,-1,-1,-1,-1,-1,-1\} \end{aligned} \tag{5.4}$$

And the pilot sub-carriers at OFDM symbol n (starting at 0 from the first symbol after the long preamble) is then:

$$P^{(n)}_{-21,-7,7,21} = \{p_{n\%127}, p_{n\%127}, p_{n\%127}, -p_{n\%127}\} \tag{5.5}$$

For 802.11n at 20MHz bandwidth with single spatial stream, the n'th pilot sub-carriers are:

$$P^{(n)}_{-21,-7,7,21} = \{\Psi_{n\%4}, \Psi_{(n+1)\%4}, \Psi_{(n+2)\%4}, \Psi_{(n+3)\%4}\} \tag{5.6}$$

And:

$$\Psi_{0,1,2,3} = \{1,1,1,-1\} \tag{5.7}$$

In other words, the pilot sub-carries of the first few symbols are:

$$\begin{aligned} P^{(0)}_{-21,-7,7,21} &= \{1,1,1,-1\}\\ P^{(1)}_{-21,-7,7,21} &= \{1,1,-1,1\}\\ P^{(2)}_{-21,-7,7,21} &= \{1,-1,1,1\}\\ P^{(3)}_{-21,-7,7,21} &= \{-1,1,1,1\}\\ P^{(4)}_{-21,-7,7,21} &= \{1,1,1,-1\}\\ &\ldots \end{aligned} \tag{5.8}$$

For other configurations (e.g., spatial stream, bandwidth), the pilot sub-carrier pattern can be found in Section 20.3.11.10 in `802.11-2012 std`.

The residual phase offset at symbol n can then be estimated as:

$$\theta_n = \angle\Big(\sum_{i\in\{-21,-7,7,21\}} \overline{X^{(n)}[i]} \times P^{(n)}[i] \times H[i]\Big) \tag{5.9}$$

Combine this phase offset and the previous channel gain correction together, the adjustment to symbol n is:

$$Y^{(n)}[i] = \frac{X^{(n)}[i]}{H[i]} e^{j\theta_n} \tag{5.10}$$

Fig. 5.6 shows the effect of correcting the residual CFO using pilot sub-carriers. Each sub-carrier can then be mapped to constellation points easily.

In OpenOFDM, the above tasks are implemented by the `equalizer.v` module. It first stores the first LTS, and then calculates the mean of the two LTS and store it as channel gain.

For each incoming OFDM symbol, it first obtains the polarity of the pilot sub-carriers in current symbol, then calculates the residual CFO using the pilot sub-carriers and also performs the channel gain correction.

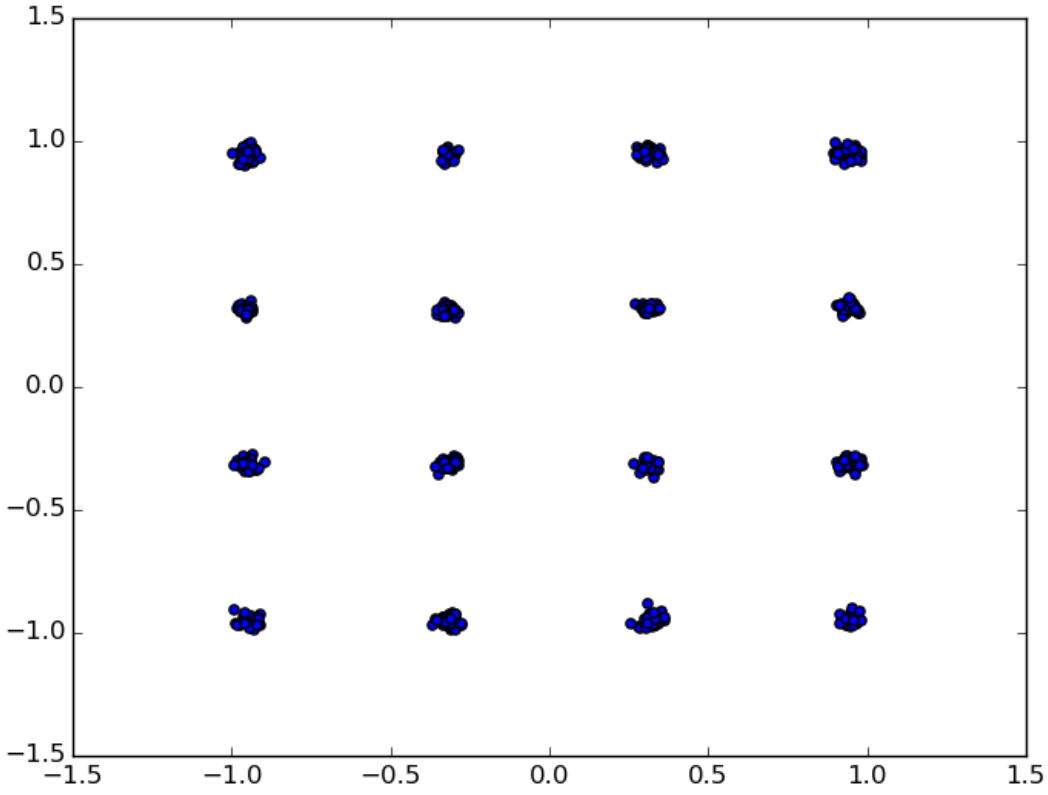

Fig. 5.6: Residual CFO Correction Using Pilot Sub-Carriers

CHAPTER 6

Decoding

Now we have corrected the residual CFO and also have corrected the channel gain, the next step is to map the FFT output to actual data bits. This is the reverse process of encoding a packet.

1. demodulation: complex number to bits
2. deinterleaving: shuffle the bits inside each OFDM symbol
3. Convolution decoding: remove redundancy and correct potential bit errors
4. Descramble.

Step 1 and 3 depend on the modulation and coding scheme, which can be obtained from the SIGNAL field. The SIGNAL field is encoded in the first OFDM symbol after the long preamble and is always BPSK modulated regardless of the actual modulation. Recall that in 802.11a/g, one OFDM symbol contains 48 data sub-carriers, which corresponds to 48 data bits in BPSK scheme. The SIGNAL field is also convolutional encoded at 1/2 rate so there are 24 actual data bits in the SIGNAL field.

Next, we first go through the decoding process and then explain the format of both legacy (802.11a/g) and the HT (802.11n) SIGNAL format.

6.1 Demodulation

- **Module**: `demodulate.v`
- **Input**: `rate (7), cons_i (16), cons_q (16)`
- **Output**: `bits (6)`

This step maps the complex number in the FFT plane into bits. Fig. 6.1 shows the constellation encoding schemes for BPSK, QPSK, 16-QAM and 64-QAM. also supported in OpenOFDM.

Inside each OFDM symbol, each sub-carrier is mapped into 1, 2, 4 or 6 bits depending on the modulation.

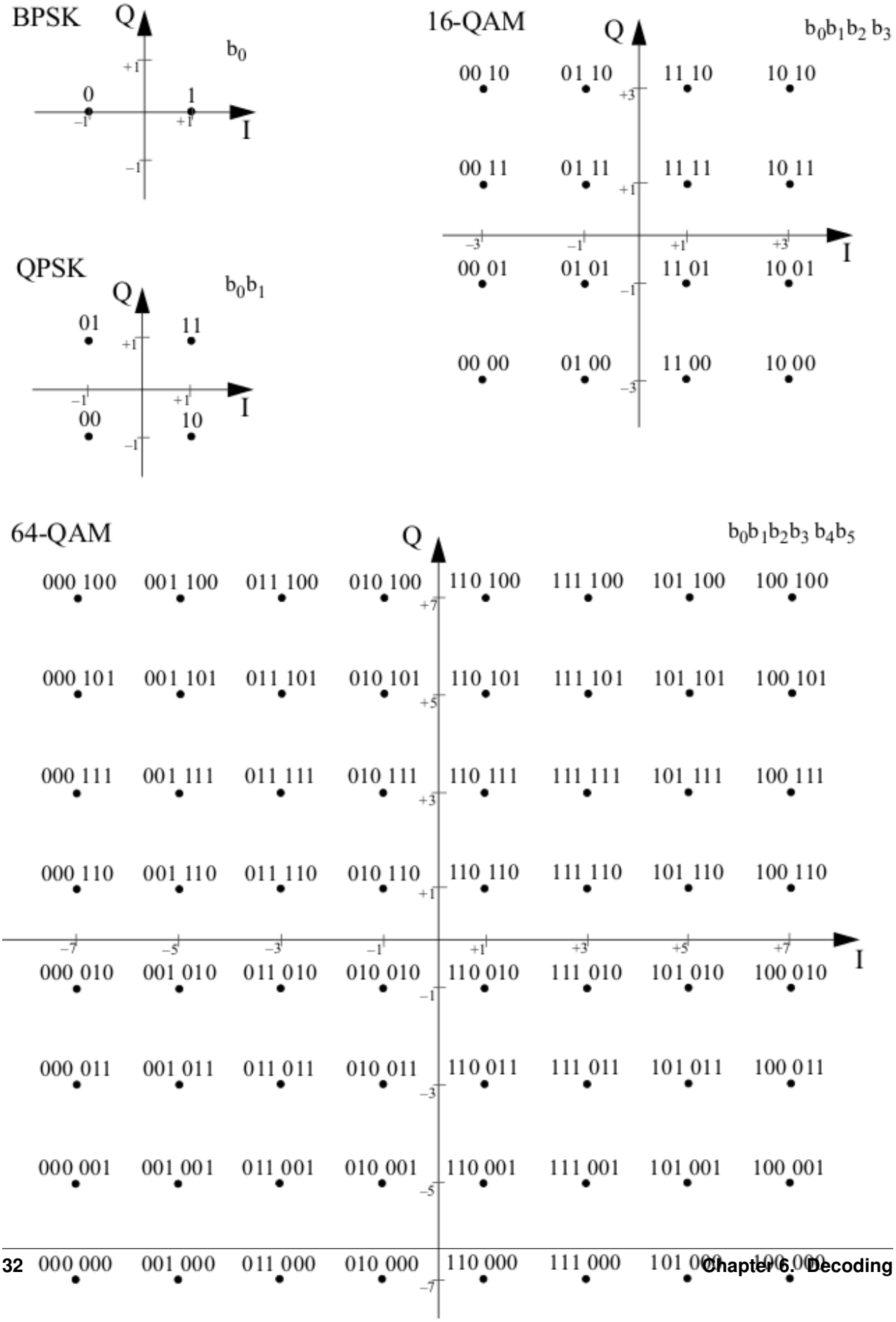

Fig. 6.1: BPSK, QPSK, 16-QAM and 64-QAM Constellation Bit Encoding

6.2 Deinterleaving

- **Module**: `deinterleave.v`
- **Input**: `rate (7), in_bits (6)`
- **Output**: `out_bits (2), erase (2)`

Inside each OFDM symbol, the encoded bits are interleaved. To understand how the block interleaver works, first we need to define a few parameters. Here we only consider 802.11a/g and 802.11n single spatial stream mode.

Table 6.1: Modulation Dependent Parameters (802.11a/g)

Modulation	Coding Rate	Bit-Rate	N_{BPSC}	N_{CBPS}	N_{DBPS}
BPSK	1/2	6	1	48	24
BPSK	3/4	9	1	48	36
QPSK	1/2	12	2	96	48
QPSK	3/4	18	2	96	72
16-QAM	1/2	24	4	192	96
16-QAM	3/4	36	4	192	144
64-QAM	2/3	48	6	288	192
64-QAM	3/4	54	6	288	216

where:

- N_{BPSC}: number of bits per sub-carrier
- N_{CBPS}: number of coded bits per OFDM symbol
- N_{DBPS}: number of data bits per OFDM symbol

Let $s = max(N_{BPSC}/2, 1)$ be the number of bits along the real (or imaginary) axis in the constellation plane. The interleaver is based on writing the data bits in rows and reading them out in columns.

Table 6.2: Row and Columns of 802.11 Interleaver

	802.11a/g	802.11n 20MHz
N_{COL}	16	13
N_{ROW}	$3 \times N_{BPSC}$	$4 \times N_{BPSC}$

The interleaving process involves two permutations. Let k be the index of the bit index before the first permutation, i be the index after the first but before the second permutation, and j be the index after the second permutation.

The first permutation ($k \rightarrow i$) of interleaving ensures adjacent code bits are mapped to non-adjacent sub-carriers, and is defined as:

$$i = N_{ROW} \times (k \bmod N_{COL}) + \lfloor \frac{k}{N_{COL}} \rfloor \tag{6.1}$$

And the second permutation ($i \rightarrow j$) ensures that adjacent code bits are mapped alternatively to less or more significant bits in constellation point, and is defined as:

$$j = s \times \lfloor \frac{i}{s} \rfloor + (i + N_{CBPS} - \lfloor N_{COL} \times \frac{i}{N_{CBPS}} \rfloor) \bmod s \tag{6.2}$$

The deinterleaving process involves two permutations as well to reverse the two permutations in interleaving process.

First, to reverse the second permutation ((6.2)):

$$i = s \times \lfloor \frac{j}{s} \rfloor + (j + \lfloor N_{COL} \times \frac{j}{N_{CBPS}} \rfloor) \bmod s \tag{6.3}$$

And to reverse the first permutation:

$$k = N_{COL} \times i - (N_{CBPS} - 1) \times \lfloor \frac{i}{N_{ROW}} \rfloor \tag{6.4}$$

In OpenOFDM, the deinterleaving is performed using look up table. First, the bits in one OFDM symbol are stored in a two-port RAM. Then the bits are read according to the look up table.

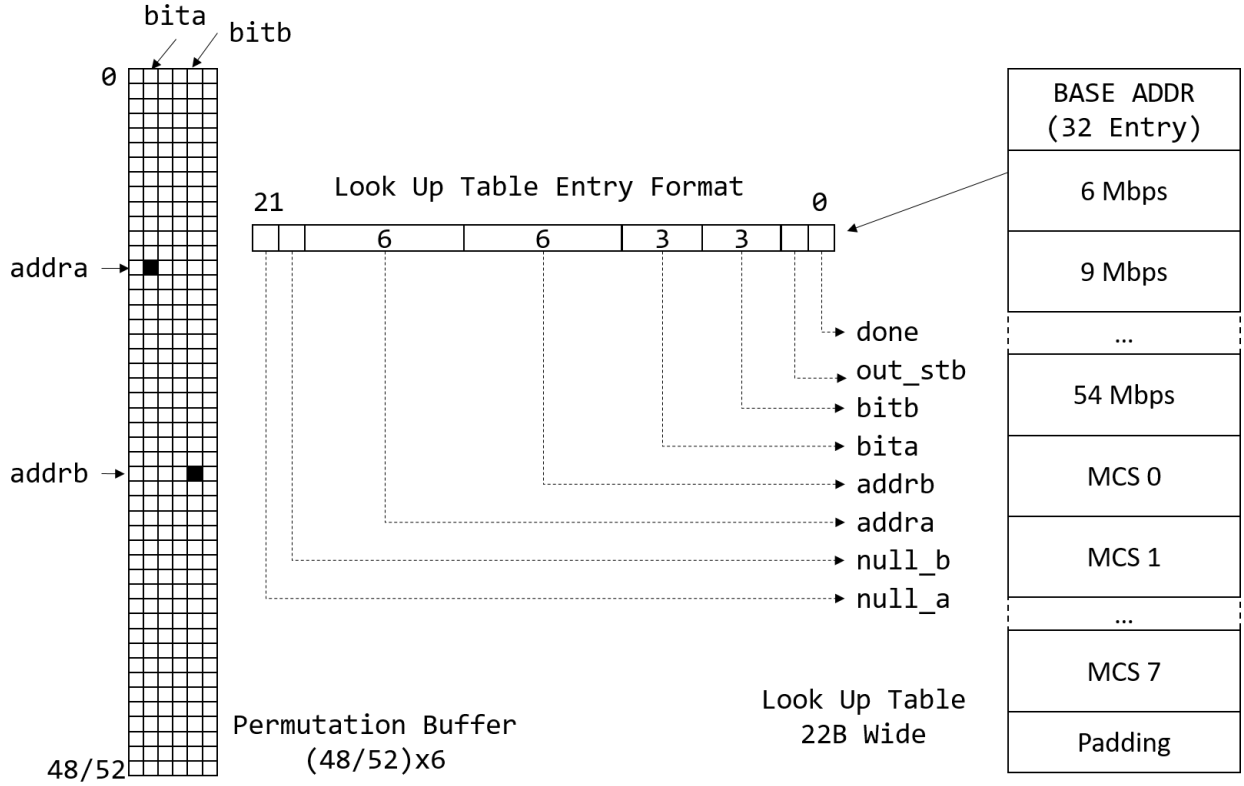

Fig. 6.2: Deinterleave Look Up Table

As shown in Fig. 6.2, the raw bits of one OFDM symbol is first stored in the permutation buffer. The buffer entry is 6-bit wide to accommodate 64-QAM. For other modulations, only the lower N_{BPSC} bits are valid. The buffer has 48 (802.11a/g) or 52 (802.11n) rows depend on whether HT is used.

After all the bits inside one OFDM symbol are written to the permutation buffer, we first get the base address of the sub look up table for current modulation scheme. For 802.11a/g, the key is the rate bits inside the SIGNAL field. For 802.11n, the key is $mcs + 16$. The next step is to read the look up table to determine which bits to output next.

The lookup table entry is 22 bits wide, which contains:

- `null_a/null_b`: whether the current bit is valid (for punctuation in Viterbi decoding next)
- `addra/bita`: the first bit to output
- `addrb/bitb`: the second bit to output
- `out_stb`: output strobe
- `done`: end of sub-LUT for current modulation

Note that the deinterleave module output 2 bits at each clock cycle. The look up table is generated by `scripts/gen_deinter_lut.py`.

For non 1/2 modulation rates, we need to compensate for the punctuation in the deinterleaving step to make following Viterbi decoding easier. This is achieved by inserting dummy bits (via the `null_a/null_b` bits) accordingly. The exact punctuation pattern can be found in Figure 18-9 in `802.11-2012 std`.

6.3 Viterbi Decoding

The transmitted bits are convolutional encoded which adds redundancy to the bits and help the receiver fix bit errors. The decoding can be performed using Viterbi algorithm. We utilize the Viterbi IP core provided by Xilinx. It is not free but you can obtain a evaluation license. The limitation of the evaluation license is that the core will stop working after certain time (several hours) after the FPGA is powered up.

The Viterbi core handles most of the heavy lifting and we only need to feed it with the de-punctured bits output from the deinterleave module.

For SIGNAL or HT-SIG fields, the decoding stops here. For data symbols, the last step is to descramble.

6.4 Descrambling

The scrambling step at the transmitter side is to avoid long consecutive sequences of 0s or 1s. The scrambling and descrambling process can be realized using the same logic.

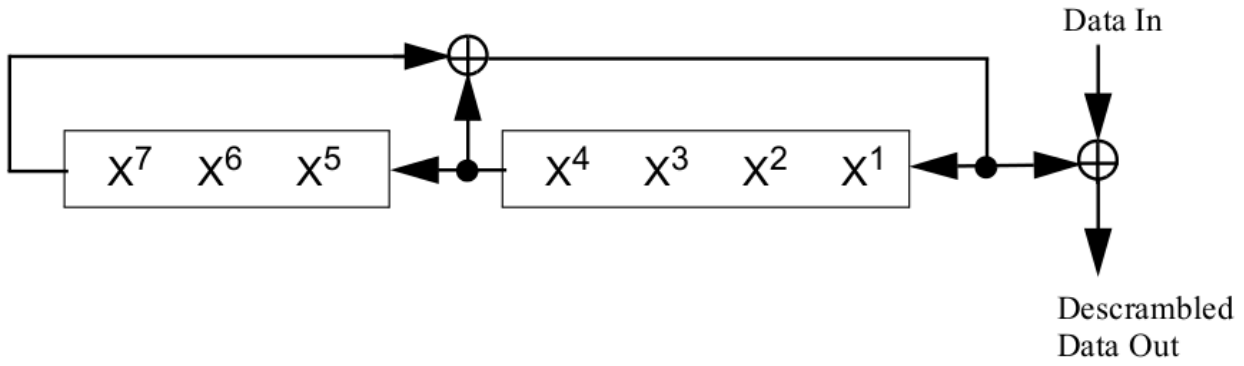

Fig. 6.3: Scrambler/Descrambler Logic

Suppose the current input bit is B_n, the scrambled bit B_n^s and the internal state of the scrambler is updated as follows:

$$\begin{aligned} B_n^s &\leftarrow X_n^1 \oplus B_n \\ X_{n+1}^1 &\leftarrow X_n^7 \oplus X_n^4 \\ X_{n+1}^i &\leftarrow X_n^{i-1}, i = 2, 3, \ldots, 7 \end{aligned} \tag{6.5}$$

where X_n^i is the scrambler state before the nth input bit, $n = 0, 1, 2, \ldots$.

At the transmitter side, for each packet, the scrambler is initialized with pseudo random value. The very first 7 bits of the data bits is preset to zero before scrambling, so that the receiver can estimate the value using the scrambled bits.

Now let's see how the receiver recovers the initial state of the transmitter's scrambler. There are two ways to interpret this.

First, we can *calculate* the initial state. Since the first 7 un-scrambled bits (B_0 to B_6) are all zeros, the scrambled bits can be obtained by:

$$\begin{aligned}
B_0^s &= X_0^7 \oplus X_0^4 \\
B_1^s &= X_1^7 \oplus X_1^4 = X_0^6 \oplus X_0^3 \\
B_2^s &= X_2^7 \oplus X_2^4 = X_0^5 \oplus X_0^2 \\
B_3^s &= X_3^7 \oplus X_3^4 = X_0^4 \oplus X_0^1 \\
B_4^s &= X_4^7 \oplus X_4^4 = X_0^3 \oplus B_0^s \\
B_5^s &= X_5^7 \oplus X_5^4 = X_0^2 \oplus B_1^s \\
B_6^s &= X_6^7 \oplus X_6^4 = X_0^1 \oplus B_2^s
\end{aligned} \tag{6.6}$$

From which we can reverse calculating the value of X as follows:

$$\begin{aligned}
X_0^1 &= B_6^s \oplus B_2^s \\
X_0^2 &= B_5^s \oplus B_1^s \\
X_0^3 &= B_4^s \oplus B_0^s \\
X_0^4 &= B_3^s \oplus X_0^1 = B_3^s \oplus B_6^s \oplus B_2^s \\
X_0^5 &= B_2^s \oplus X_0^2 = B_2^s \oplus B_5^s \oplus B_1^s \\
X_0^6 &= B_1^s \oplus X_0^3 = B_1^s \oplus B_4^s \oplus B_0^s \\
X_0^7 &= B_0^s \oplus X_0^4 = B_0^s \oplus B_3^s \oplus B_6^s \oplus B_2^s
\end{aligned} \tag{6.7}$$

This interpretation does not lead to efficient Verilog implementation since we need to first buffer the first 7 bits, calculate the initial state and then descramble from the first 7 bits again.

The second interpretation is that: **the first 7 scrambled bits are the state after scrambling the 7 bits**. In other words, we have:

$$\begin{aligned}
X_7^7 &= B_0^s \\
X_7^6 &= B_1^s \\
X_7^5 &= B_2^s \\
X_7^4 &= B_3^s \\
X_7^3 &= B_4^s \\
X_7^2 &= B_5^s \\
X_7^1 &= B_6^s
\end{aligned} \tag{6.8}$$

For instance, take a look at X_7^7,

$$X_7^7 = X_6^6 = \ldots = X_1^1 = X_0^7 \oplus X_0^4 \tag{6.9}$$

We also know that:

$$\begin{aligned}
B_0^s &= X_0^7 \oplus X_0^4 \oplus B_0 \\
&= X_0^7 \oplus X_0^4 \oplus 0 \\
&= X_0^7 \oplus X_0^4
\end{aligned} \tag{6.10}$$

Therefore $X_7^7 = B_0^s$. This way we directly get the state to descramble the next bit B_7^s, resulting a very simple Verilog implementation.

CHAPTER 7

SIGNAL and HT-SIG

The first OFDM symbol after long preamble is the SIGNAL field, which contains the modulation rate and length of the packet. These information are needed to determine how many OFDM symbols to decode and how to decode them.

7.1 Legacy SIGNAL

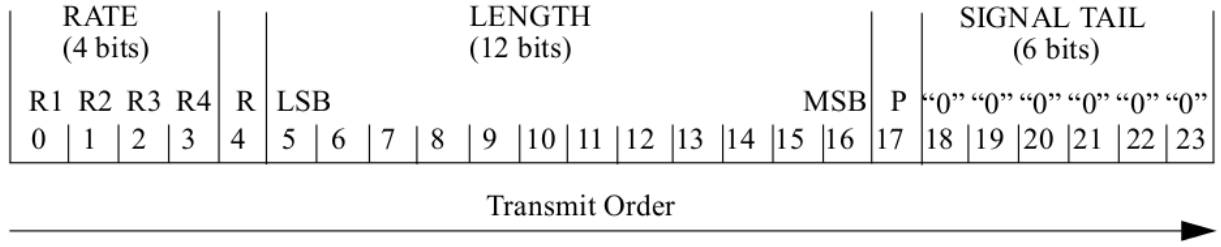

Fig. 7.1: SIGNAL field of 802.11a/g

For 802.11a/g, the SIGNAL field is 24-bits, which expands to 48 bits after 1/2 convolutional encoding and fits precisely into one OFDM symbol. Fig. 7.1 shows the format of SIGNAL.

In OpenOFDM, we check the following properties to make sure the SIGNAL field is decoded properly.

- Parity. Bit 17 is a even parity bit of the previous 17 bits.
- Reserved bit. Bit 4 is reserved, and should be 0.
- Tail bits. The last 6 bits should be all 0.

If any checking failed, we stop decoding immediately and wait for next power trigger.

7.2 HT-SIG

For backward compatibility, 802.11n shares the same preambles and SIGNAL field with 802.11a/g so that legacy stations can also decode the SIGNAL field and back-off accordingly (see NAV).

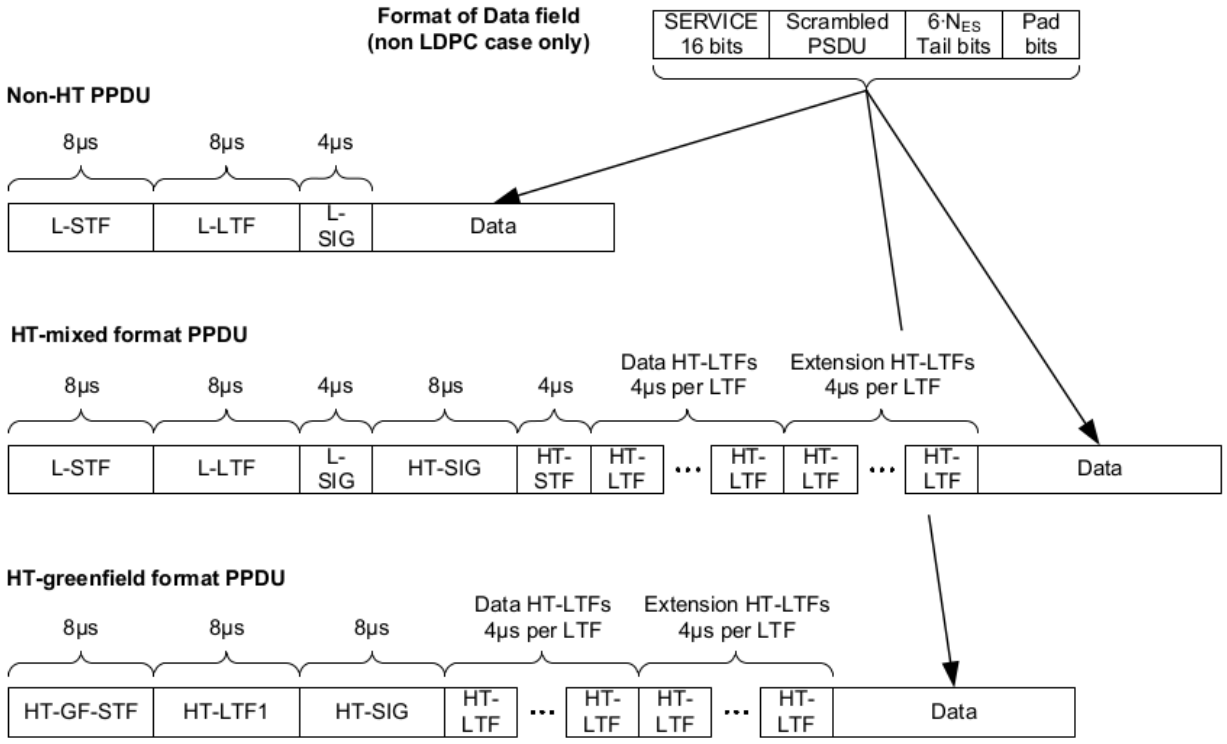

Fig. 7.2: PPDU Format of 802.11n

As shown in Fig. 7.2, there are actually three PPDU formats supported in 802.11n. The legacy mode is identical to 802.11a/g. The HT-mixed mode provides backward compatibility, and is mostly widely used. Finally, the HT-greenfield mode is pure 802.11n and does not have backward compatibility. OpenOFDM supports HT-mixed mode only.

In HT-mixed mode, the rate field in SIGNAL (or L-SIG) is always 6 Mbps, and the LENGTH is adjusted accordingly so that it reflects the actual packet air duration.

From receiver's point of view, after decoding the SIGNAL field, if the rate is not 6 Mbps, then this is a 802.11a/g packet and we continue to decoding the DATA bits. However, if the rate is 6 Mpbs, then we need to first check if this is a 802.11n packet by detecting the HT-SIG field. This is achieved by examine the BPSK constellation points of the OFDM symbol after SIGNAL.

As shown in Fig. 7.3, HT-SIG is BPSK modulated using the Quadrature component instead of the In-phase component. Therefore, we check the number of samples in which the quadrature component is larger than in-phase, and claim a HT-SIG if enough such samples are detected (4 in OpenOFDM).

The HT-SIG field spans two OFDM symbols, and has 48 data bits (96 coded bits) in total. The constellation points are rotated 90 degrees clockwise before decoding.

Fig. 7.4 shows the format of HT-SIG. The following fields are checked in OpenOFDM:

- MCS: only supports 0 - 7.
- CBW 20/40: channel bandwidth. OpenOFDM only supports 20 MHz channel (0).

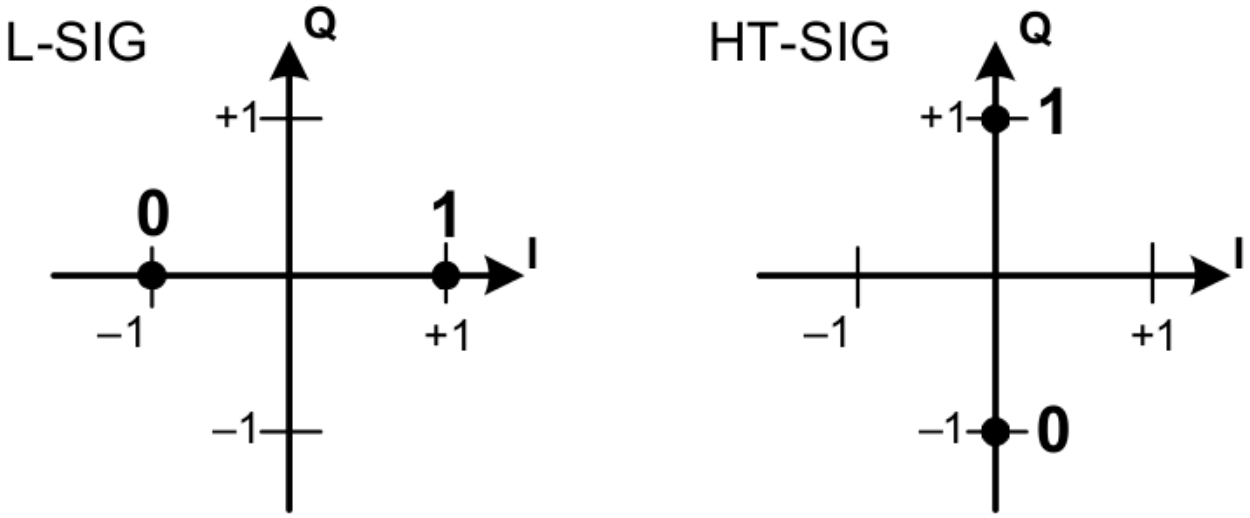

Fig. 7.3: Constellation Points of HT-SIG vs. SIGNAL

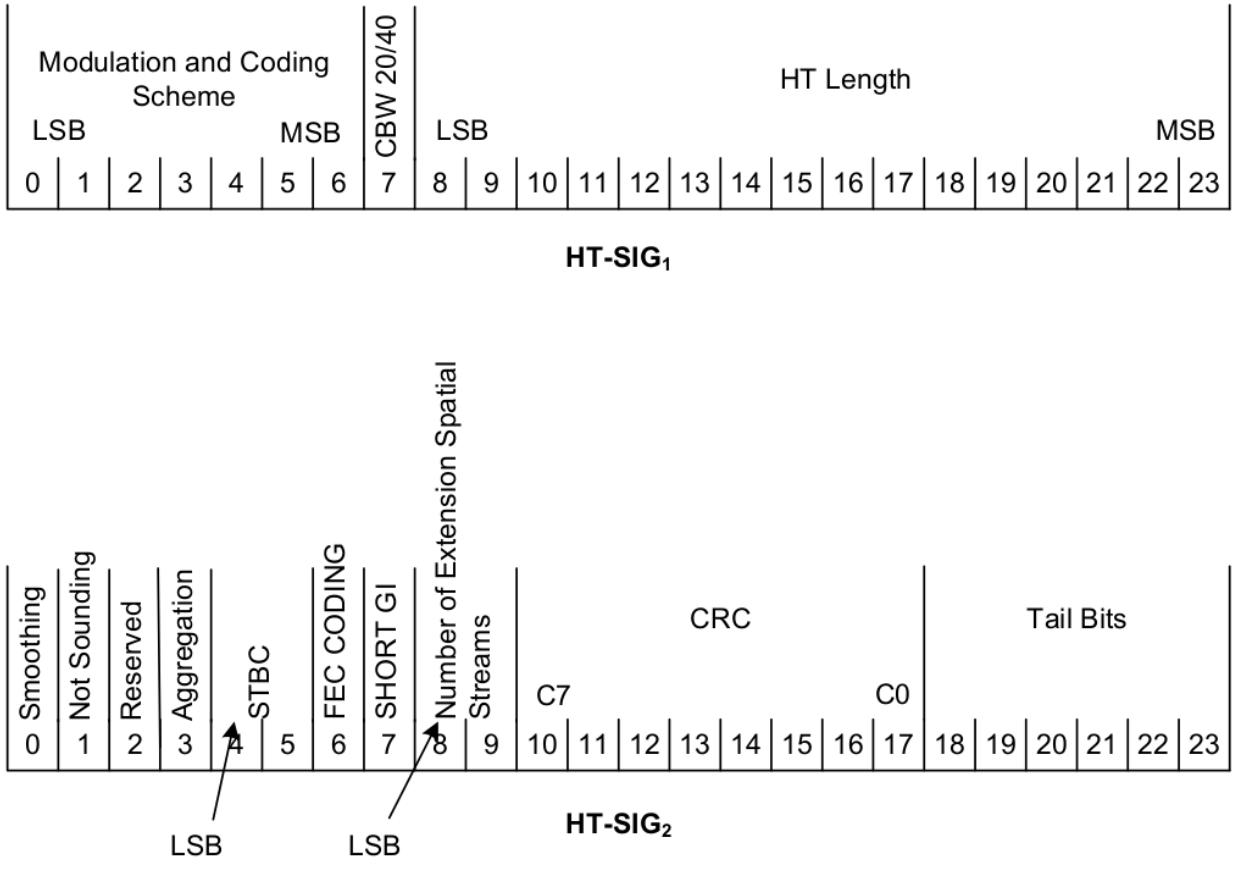

Fig. 7.4: HT-SIG Format

- Reserved: must be 0.
- STBC: number of space time block code. OpenOFDM only supports 00 (no STBC).
- FEC coding: OpenOFDM only supports BCC (0).
- Short GI: whether short guard interval is used.
- Number of extension spatial streams: only 0 is supported.
- CRC: checksum of previous 34 bits.
- Tail bits: must all be 0.

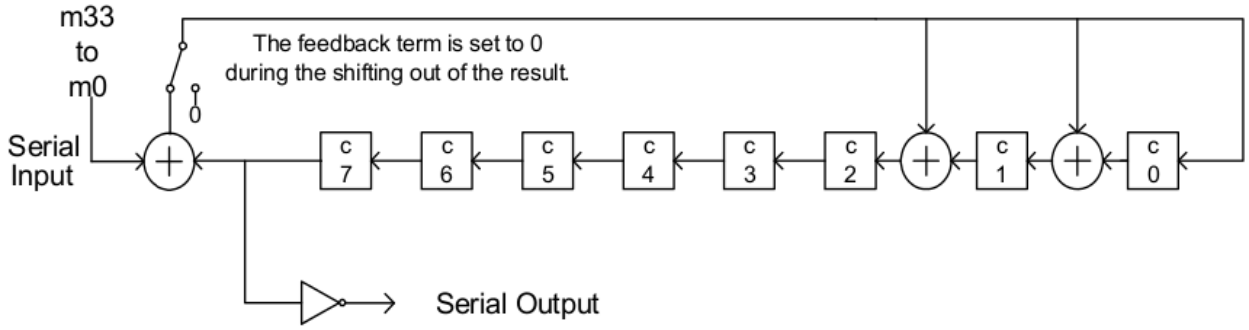

Fig. 7.5: CRC Calculation of HT-SIG

Fig. 7.5 shows the logic to calculate the CRC in HT-SIG. The shift registers $C_0, \ldots, C_7$ are initialized with all ones. For each data bit $m_0, \ldots, m_{33}$, the shift register is updated as:

$$
\begin{aligned}
C_7^{i+1} &= C_6^i \\
C_6^{i+1} &= C_5^i \\
C_5^{i+1} &= C_4^i \\
C_4^{i+1} &= C_3^i \\
C_3^{i+1} &= C_2^i \\
C_2^{i+1} &= C_1^i \oplus C_7^i \oplus m_i \\
C_1^{i+1} &= C_0^i \oplus C_7^i \oplus m_i \\
C_0^{i+1} &= C_7^i \oplus m_i
\end{aligned}
\tag{7.1}
$$

The CRC is then $\overline{C_7^{34}}, \ldots, \overline{C_0^{34}}$. Note the bits are inverted.

The next OFDM symbol after HT-SIG is HT short preamble, which is skipped in OpenOFDM. The following OFDM symbol contains HT long training sequence, which replaces the legacy channel gain inside `equalizer.v` module. The rest decoding logic is similar to 802.11a/g, except the number of data sub-carriers is adjusted from 48 to 52.

CHAPTER 8

Setting Registers

- **Module**: `usrp/setting_reg.v`
- **Input**: `set_stb`, `set_addr` and `set_data`
- **Output**: `out`, `changed`

To enable dynamic configuration of decoding parameters at runtime, the USRP N210 provides the setting register mechanism. Most modules in OpenOFDM have three common inputs for such purpose:

- `set_stb (1)`: asserts high when the setting data is valid
- `set_addr (8)`: register address (256 registers possible in total)
- `set_data (32)`: the register value

Here is a list of setting registers in OpenOFDM.

Table 8.1: List of Setting Registers in OpenOFDM.

Name	Addr	Module	Bit Width	Default Value	Description
SR_POWRE_THRES	3	power_trigger.v	16	100	Threshold for power trigger
SR_POWER_WINDOW	4	power_trigger.v	16	80	Number of samples to wait before reset the trigger signal
SR_SKIP_SAMPLE	5	power_trigger.v	32	5000000	Number of samples to skip initially
SR_MIN_PLATEAU	6	sync_short.v	32	100	Minimum number of plateau samples to declare a short preamble

CHAPTER 9

Verilog Hacks

Because of the limited capability of FPGA computation, compromises often need to made in the actual Verilog implementation. The most used techniques include quantization and look up table. In OpenOFDM, these approximations are used.

9.1 Magnitude Estimation

- **Module**: `complex_to_mag.v`
- **Input**: `i (32), q (32)`
- **Output**: `mag (32)`

In the `sync_short` module, we need to calculate the magnitude of the `prod_avg`, whose real and imagine part are both 32-bits. To avoid 32-bit multiplication, we use the Magnitude Estimator Trick from DSP Guru. In particular, the magnitude of complex number $\langle I, Q \rangle$ is estimated as:

$$M \approx \alpha * max(|I|, |Q|) + \beta * min(|I|, |Q|) \tag{9.1}$$

And we set $\alpha = 1$ and $\beta = 0.25$ so that only simple bit-shift is needed.

Fig. 9.1 shows the waveform of the `complex_to_mag` module. In the first clock cycle, we calculate `abs_i` and `abs_q`. In the second cycle, `max` and `min` are determined. In the final cycle, the magnitude is calculated.

9.2 Phase Estimation

- **Module**: `phase.v`
- **Input**: `i (32), q (32)`
- **Output**: `phase (32)`
- **Note**: The returned phase is scaled up by 512 (i.e., $int(\theta * 512)$)

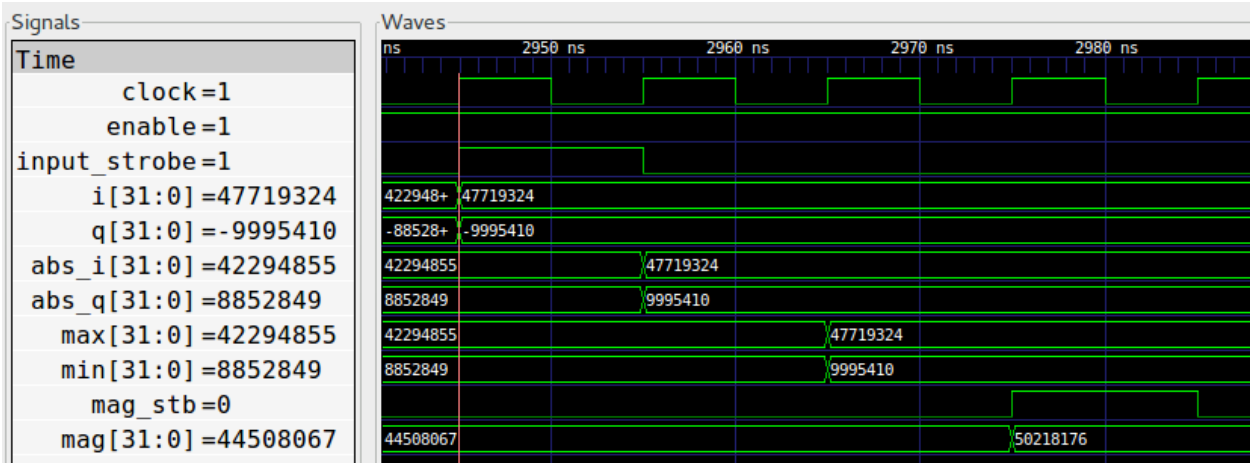

Fig. 9.1: Waveform of `complex_to_mag` Module

When correcting the frequency offset, we need to estimate the phase of a complex number. The *right* way of doing this is probably using the CORDIC algorithm. In OpenOFDM, we use look up table.

More specifically, we calculate the phase using the $arctan$ function.

$$\theta = \angle(\langle I, Q \rangle) = arctan(\frac{Q}{I}) \tag{9.2}$$

The overall steps are:

1. Project the complex number to the $[0, \pi/4]$ range, so that the $tan(\theta)$ range is $[0, 1]$.
2. Calculate $arctan$ (division required)
3. Looking up the quantized $arctan$ table
4. Project the phase back to the $[-\pi, \pi)$ range

Here we use both quantization and look up table techniques.

Step 1 can be achieved by this transformation:

$$\langle I, Q \rangle \rightarrow \langle max(|I|, |Q|), min(|I|, |Q|) \rangle \tag{9.3}$$

In the lookup table used in step 3, we use $int(tan(\theta) * 256)$ as the key, which effectively maps the $[0.0, 1.0]$ range of tan function to the integer range of $[0, 256]$. In other words, we quantize the $[0, \pi/4]$ quadrant into 256 slices.

This $arctan$ look up table is generated using the `scripts/gen_atan_lut.py` script. The core logic is as follows:

```
SIZE = 2**8
SCALE = SIZE*2
data = []
for i in range(SIZE):
    key = float(i)/SIZE
    val = int(round(math.atan(key)*SCALE))
    data.append(val)
```

Note that we also scale up the $arctan$ values to distinguish adjacent values. This also systematically scale up π in OpenOFDM. In fact, π is defined as $1608 = int(\pi * 512)$ in `verilog/common_params.v`.

The generated lookup table is stored in the `verilog/atan_lut.coe` file (see COE File Syntax). Refer to this guide on how to create a look up table in Xilinx ISE. The generated module is stored in `verilog/coregen/atan_lut.v`.

9.3 Rotation

- **Module**: `/verilog/rotate.v`
- **Input**: `i (16), q (16), phase (32)`
- **Output**: `out_i (16), out_q (16)`
- **Note**: The input phase is assumed to be scaled up by 512.

To rotate a complex number $C = I + jQ$ by θ degree, we can multiply it by $e^{j\theta}$, as shown in (9.4).

$$C' = (I + jQ) \times (\cos(\theta) + j\sin(\theta)) \tag{9.4}$$

Again, this can be done using the CORDIC algorithm. But similar to *Phase Estimation*, we use the look up table.

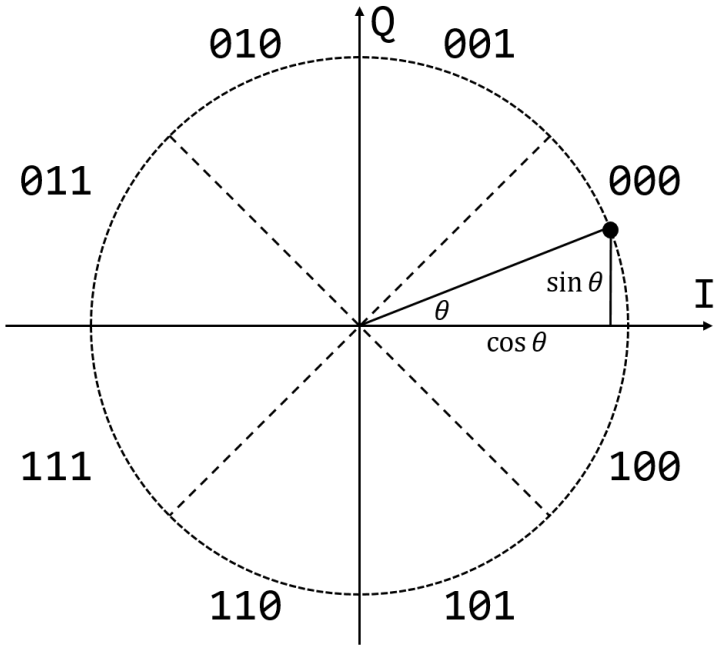

Fig. 9.2: Quadrant in I/Q Plane

As shown in Fig. 9.2, we split the I/Q plane into 8 quadrants, $\pi/4$ each. To avoid storing nearly duplicate entries in the table, we first map the phase to be rotated ($[-\pi, \pi]$) into the $[0, \pi/4]$ range. Next, since the incoming phase is scaled up by 512, each quadrant is further split into $402 = int(\pi/4 * 512)$ sectors. And the $\cos(\theta)$ and $\sin(\theta)$ values (scaled up by 2048) are stored in the look up table. The table is generated by the `scripts/gen_rot_lut.py`.

CHAPTER 10

Integration with USRP

OpenOFDM was originally developed on Ettus Research USRP N210 platform. This short guide explains how to modify the USRP N210's FPGA code base to accommodate OpenOFDM.

10.1 USRP N2x0 FPGA Overview

The top level model of USRP N2x0 (N200 and N210) can be found in `top/N2x0/u2plus.v`. It instantiates the `u2plus_core` module, which contains the core modules such as the receiver and transmit chain. In particular, the receive chain includes `rx_frontend`, `ddc_chain` and `vita_rx_chain`. Similarly, the transmit chain includes `vita_tx_chain`, `duc_chain` and `tx_frontend`.

The code base contains placeholder modules (`dsp_rx_glue` and `dsp_tx_glue`) for extension. These modules are controlled by Verilog compilation flags and by default they are simply pass-through and have no effect on the signal processing at all.

10.2 Enable Custom Modules

Take the receive chain as an example, inside `dsp_rx_glue` module, it checks the `RX_DSP0_MODULE` macro and instantiates it if found. The macro can be defined in a customized Makefile. Make a copy of the `top/N2x0/Makefile.N210R4`, name it to `top/N2x0/Makefile.N210R4.custom`. And then make these changes.

- Change `BUILD_DIR` to `$(abspath build$(ISE)-N210R4-custom)`. This will create a new build directory for our custom build.
- Comment out `CUSTOM_SRCS` and `CUSTOM_DEFS`. We will define them in a separate Makefile.
- Find `Verilog Macros` and change it to "`LVDS=1|RX_DSP0_MODULE=custom_dsp_rx|RX_DSP1_MODULE=custom`
  This defines the macros to so that the custom modules are instantiated by the glue modules mentioned earlier.

After these changes, the two modules in `custom/custom_dsp_rx.v` and `custom/custom_dsp_tx.v` will be instantiated. By default they are simply pass-through. For instance, the output of RF frontend are directly connnected to the input of DDC, and the output of DDC are directly connected to the VITA RX module.

To integrate OpenOFDM, we only need to *insert* it after the DDC but before VITA RX module. That is, the `sample_in/sample_in_strobe` of the `dot11` module should be connected to the `ddc_out/ddc_out_strobe` signal.

Also note that two receive chains are defined in `u2plus_core` module, so that the two antenna ports can be configured in TX/RX or RX/RX mode. To save FPGA resource, you may want to comment out one of the RX chains to make more room for OpenOFDM.