1. INTRODUCTION

The demand for wireless service is exponential increasing. Multi-access wireless LAN techniques, enabling users to share the resources, are one of the main components to alleviate this high demand. Four main multiple access technologies are used: frequency division multi-access, time division multi-access, code division multi-access, and space division multi-access. The first three have been extensively studied, while the spacial diversity has risen attention because of emergence of the multiple-input and multiple-out (MIMO) technology.

MIMO significantly boost network capacity by using multiple antennas at both the transmitter and receiver to exploit the spatial properties of wireless channels. MIMO has been incorporated into several wireless standards, including IEEE 802.11n. In single user MIMO system, the capacity improvement is bounded by the number of transmitting or receiving antenna, whichever comes smaller. In practice, the wireless devices generally have only a few antennas. For example, 802.11n devices only equip two antennas. However, access points (AP's) potentially can deploy a larger number of antennas. If we let multiple devices to form a “virtual MIMO” system, which is spacial multiple access in nature, then the network capacity would not be bounded the individual devices. Tse et. El [3] pointed out that the AP may be able to decode all concurrent frames correctly as long as the number of them is less than the number number of antennas at the AP.

In this project, I am planning to design a practical system that enables multiple access by exploiting spacial diversity. Tan et. al. 's work, SAM [1], is my design baseline.

2. PROBLEM DESCRIPTION

One of the main challenges in implementing spatial multiple access is the effect of intersymbol interference (ISI) when trying to decode concurrent transmissions from asynchronous senders in a wireless LAN. SAM applied interference alignment and cancellation [2] techniques to address this challenge.

Let's look at a simple scenario, Illustration 1, where the AP is trying to decode two overlapped frames from two stations at different locations. The AP performs two steps to decode each frame: the interference nullifying (IN) and interference cancellation (IC).
The idea of IN is to find two proper linear transforms for the two received signals, $y_1$ and $y_2$ in Illustration 2, at the two antennas such that, after the transformation, the components ($x_1$ and $x_2$) in these two received signals contributed by one transmitted frame are aligned with the same direction and scale. Then a subtraction of these two transformed signals removes $x_1$ and is ready for decoding. In the next step, IC, the AP regenerates the original signal and cancels out the decoded $x_2$, so that $x_1$ can be decoded.

In summary, the decode procedure is as following:

1. Use the clean preamble of the first arrived frame, $Pa$, of the two overlapped frames to align the signals.
2. Subtract the two signals to get the signal with the information of the second frame, $Pb$; decode $Pb$. 

Illustration 1: A simple scenario

Illustration 2: Illustration of interference nullifying
(3) Re-encode $Pb$, and cancel it out from the original signal, to get the signal with the information of $Pa$; decode $Pa$.

In this project, I am implementing IN and IC for 802.11b on a SDR platform. In the next session, I will describe the processing requirement for the implementation.

3. PROCESSING REQUIREMENT

3.1 802.11b PHY layer requirement

The PHY layer of 802.11b communication system contains four main functional blocks, as shown in Illustration 3. The binary bit stream at 2Mbps coming from the MAC layer will be first scrambled. This operation does not change the bit rate. The bit stream then is fed to QPSK modulation, where every two bits are mapped to one complex symbol. To provide sufficient fidelity, these complex samples require 16-bit quantization for both I and Q components, therefore 32 bits per symbol and the bit rate coming out of QPSK is at 32Mbps. Next operation is Direct Sequence Spread Spectrum (DS-SS) to allow 802.11b provide different data rate. The code length used in 802.11b is 11 chips, so the bit rate coming out of DS-SS will be increased by 11 times. The last operation is up sampling by a factor of four. This is a technique widely used for better performance [6]. Therefore, finally we have 1.4Gbps out of the PHY layer.

Illustration 3: PHY layer in 802.11b transceiver
As for the reception, in order to enable spacial multi-access as described above, even just for the simplest scenario – decoding two overlapped frames, it will require to go the those four basic functional blocks three times: decoding \( Pb \), regenerating \( Pb \) and finally decoding \( Pa \). We can see that the processing requirement for implementing the spacial multi-access is high. Firstly, the interface must be able to sustain 1.4 Gbps throughput. Conventional gigabit ethernet card cannot meet this requirement. Secondly, the computation intensity is high. If there are N operations per bit, this will require 1.4 times N Giga operations per second. Finally, real-time communication requires real-time processing – the processing latency needs to meet response deadlines.

I choose Sora platform to implement this project. Its radio control board has a maximum throughput of PCIe X32, which is equivalent to 64 Gbps. Its software also support both SIMD instructions and multi-core processing.

### 3.2 Sora Platform

Sora is a fully programmable software defined radio (SDR) on commercial personal computer architectures. Sora platform is composed of both software and hardware. The hardware components consist of a radio front-end for reception and transmission and a radio control board for high-throughput data transfer between the radio and the host memory storage. Sora offers dedicated CPU cores programming and SIMD processor extensions for high speed PHY layer processing on general purpose processor. Detailed information about Sora and an example project, SoftWiFi, can be found in [4].

Sora offers the radio control board (RCB) with a radio front-end for transmission and reception. The RCB uses PCIe bus, which support 16.7 Gbps with sub-microsecond latency. On the other side, Sora software explicitly support streamlined multi-core processing. It also efficiently supports table lookups, trading off computation for memory. Moreover, Sora supports single instruction multiple data (SIMD) for exploiting further parallelism.

In this project setup, I choose multi-core system as my computer hardware platform. The system runs Sora software and interfaces with a Sora radio control board (RCB). To simply the project, the system consist only two cores and each core has 32KB instruction, 32KB data L1 caches and a 2MB L2 cache. I use Intel SSE2 as the SIMD instruction set. Intel SSE2 supports a 128-bit packed vector.

### 4. IMPLEMENTATION

#### 4.1 Interference Nullifying

Since we have a clean preamble of \( Pa \), we can estimate the necessary system parameters of \( Pa \),
including symbol timing and carrier frequency offset by using standard mechanisms for single user communication. However, particular attention should be paid to design of the equalizer that removes the distortion due to multi-path fading.

4.1.1 Basic algorithm for channel equalizer

The channel equalizer for interference nullifying is a linear filter, \( c = c_{-L}, \ldots, c_L \), that removes or decreases the effect of a multi-path wireless channel. An optimal equalizer minimizes the difference between the filter output signals and the transmitted signals:

\[
c = \arg \min_c \| x - c \ast y \|,
\]

where \( x \) is the transmitted signal, \( y \) is the received signal, \( \| \cdot \| \) is the 2-norm operation, and \( \ast \) is the convolution operation.

The Normalized Least Mean Square algorithm works sufficiently for synchronized single-user MIMO. However, it is not sufficient when multiple users are transmitting at different symbol rates. To address this problem, a modified algorithm with over-sampling is proposed. This equalizer is trained to minimize the mean square error of all over-sampling points. At the oversampled points, the expected values by using interpolation from its nearby sample point values. For example, when in 802.11b all transmitted samples are shaped with a root-raised cosine function, then the expected values at the oversampled points will be

\[
\hat{x}(nT + \tau) = \sum_{i=-L}^{L} x(nT - iT) R(nT + \tau - iT),
\]

where \( T \) is the symbol interval, \( R(.) \) is the root-raised cosine function, and \( \tau \) is the fraction of the sampling time. In this implementation, four-time over-sampling is used, so the sampling points are \( T/4, 2T/4 \) and \( 3T/4 \).

![Illustration 4: Equalizer block diagram](image)

The following outlined the modified Normalized Least Mean Square algorithm for an adaptive equalizer.

1: \( \text{Initialize } \{c_i\} \):
2: \( c_0 = 1 \)
for each $j \neq 0$
$c_j = 0$
end for
\\ Training:
for each sample index $i$ do
  for $k = 0 \ldots K - 1$ do
    $x_{est_{i+k}} = 0$
    for $l = -L \ldots L$ do
      $x_{est_{i+k}} = x_{est_{i+k}} + c_l y_{i+k-l}$
    end for
    $err_k = x_{i+k} - x_{est_{i+k}}$
    for $j = -L \ldots L$ do
      $d_j = 0$
    end for
    $w = 0$
    for $j = -L \ldots L$ do
      $d_j = d_j + step\ err_k\ y_{i+k-1}^*$
      $w = w + |y_{i+k-j}|^2$
    end for
  end for
  $c_j = c_j - d_j/w$
end for
The algorithm applies block update, where $K$ is the number of samples in a block. To update the coefficient vector, an weighted average over a block $K$ samples are used. This method improves the stability of the equalizer while achieving relative fast convergence, because $step$ remains the same.

4.1.2. Dynamic range and quantization error analysis of the equalizer
The inputs are transmitted signals $\{x_i\}$ and the received signals $\{y_i\}$, and the output is equalizer coefficients $\{c_j\}$. $\{x_i\}$ and $\{y_i\}$ are 16-bit per sample for 256 QAM. To have a good performance equalizer, we choose a 15-tap equalizer, so $L$ is 8 and there are 15 coefficients. [5] suggested that 16 bits fixed point fractional number may be used for the coefficients, with about 10 to 12 of the most significant bits used for arithmetic operations in the equalization of the data. The remaining least significant bits are required to provide the necessary precision for the adaptation process. $step$ is a tunable parameter and it is a 8 bit fixed point fractional number between 0.01 and 0.1 [5].
The length of registers required for the input, outputs, and intermediate results without overflow are summarized in table 1. The third column is the additional fractional bits for intermediate registers in the data path to account for the quantization error due to multiplication with a fractional number. The requirement for the additional fractional bits is that the quantization error after the operation does not exceed 0.5. Since there is no recursion in this algorithm, we do not need to worry about the accumulated quantization errors.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Date type</th>
<th>Dynamic range bits</th>
<th>Additional fractional bits</th>
<th>Total register length</th>
</tr>
</thead>
<tbody>
<tr>
<td>{c_j}</td>
<td>fixed point</td>
<td>16</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>{x_i}</td>
<td>integer</td>
<td>16</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>{y_i}</td>
<td>integer</td>
<td>16</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>{x_{est}}</td>
<td>fixed point</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>{err}</td>
<td>fixed point</td>
<td>21</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>{d}</td>
<td>fixed point</td>
<td>34</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>{w}</td>
<td>integer</td>
<td>36</td>
<td>0</td>
<td>36</td>
</tr>
</tbody>
</table>

4.1.3 FIR filter design

The FIR filter is used in both training and equalizing stages and it is the component used most frequently. We need to pay attention to optimize it. The FIR filter is described as:

\[ y(n) = \sum_{i=0}^{16} c(i) x(n-i) \]

Intel SSE supports 128-bit packed vector, each FIR sample takes 32 bits, so four calculations can be performed simultaneously. In this design, I choose to work on a block of four samples at an iteration, so in each iteration, four samples are loaded in and four results in the end. Table 2 shows the illustration of computation grouping as time zero.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>c_0</th>
<th>c_1</th>
<th>c_2</th>
<th>c_3</th>
<th>c_4</th>
<th>c_5</th>
<th>c_6</th>
<th>c_7</th>
<th>...</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_{18}</td>
<td>x_{17}</td>
<td>x_{16}</td>
<td>x_{15}</td>
<td>(x_{14}</td>
<td>x_{13}</td>
<td>x_{12}</td>
<td>x_{11})</td>
<td>(x_{10}</td>
<td>x_{9}</td>
<td>x_{8}</td>
<td>...</td>
<td>y_{15}</td>
</tr>
<tr>
<td>x_{18}</td>
<td>x_{17}</td>
<td>x_{16}</td>
<td>x_{15}</td>
<td>(x_{14}</td>
<td>x_{13}</td>
<td>x_{12}</td>
<td>x_{11})</td>
<td>(x_{10}</td>
<td>x_{9}</td>
<td>x_{8}</td>
<td>...</td>
<td>y_{16}</td>
</tr>
<tr>
<td>x_{18}</td>
<td>x_{17}</td>
<td>x_{16}</td>
<td>x_{15}</td>
<td>(x_{14}</td>
<td>x_{13}</td>
<td>x_{12}</td>
<td>x_{11})</td>
<td>(x_{10}</td>
<td>x_{9}</td>
<td>x_{8}</td>
<td>...</td>
<td>y_{17}</td>
</tr>
<tr>
<td>x_{18}</td>
<td>x_{17}</td>
<td>x_{16}</td>
<td>x_{15}</td>
<td>(x_{14}</td>
<td>x_{13}</td>
<td>x_{12}</td>
<td>x_{11})</td>
<td>...</td>
<td>...</td>
<td>y_{18}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To facilitate SSE processing, the data layout in memory should be carefully designed. Table 3 shows the memory layout of the FIR coefficients. Each row forms a packed-vector containing 4 coefficients for SIMD operations. The coefficient vector is shifted left in each row, so that the four samples inputs just need to multiply each row in this table.

Table 3. Memory Layout of the FIR Filter Coefficients

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>c₀</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>c₀</td>
<td>c₁</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>c₀</td>
<td>c₁</td>
<td>c₂</td>
<td></td>
</tr>
<tr>
<td>c₀</td>
<td>c₁</td>
<td>c₂</td>
<td>c₃</td>
<td></td>
</tr>
<tr>
<td>c₁</td>
<td>c₂</td>
<td>c₃</td>
<td>c₄</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the following SSE2 code, it takes a vector of four samples, an coefficient array, and outputs four filtered samples. In each iteration, the vector of four samples multiplies the coefficients in each row of Table 3 and adds the result to the corresponding temporary accumulators for next samples to use. Only the first four rows outputs results, as shown in Table 2.

SSE2 code:

```
1:  // Load four 32-bit samples
2:  movdqa xmm0, [esi];
3:
4:  // compute the four results with the first four rows in the FIR filter coefficient table
5:  mov    edx, Coff     // reset coefficient index
6:  mov    edi, Buff     // reset temporary accumulated buffer index
7:
8:  movdqa  xmm1, xmm0;
9:  pmullw  xmm1, [edx];  // edx is coefficient index
10: paddsd xmm1, [edi];   // edi is temporary accumulated buffer index
11:
```
12: movdqa  xmm21, xmm0;
13: pmullw  xmm2, [edx+4];       // edx is coefficient index
14: paddsd  xmm2, [edi+4];      // edi is temporary accumulated buffer index
15:
16: movdqa  xmm3, xmm0;
17: pmullw  xmm3, [edx+8];       // edx is coefficient index
18: paddsd  xmm3, [edi+8];      // edi is temporary accumulated buffer index
19:
20: movdqa  xmm4, xmm0;
21: pmullw  xmm4, [edx+12];      // edx is coefficient index
22: paddsd  xmm4, [edi+12];     // edi is temporary accumulated buffer index
23:
24: // extract output from the four registers and pack them into single 128-bit output
25: paddsd  xmm1, [ecx];        // ecx is the mask index
26: paddsd  xmm2, [ecx+4];
27: paddsd  xmm3, [ecx+8];
28: paddsd  xmm4, [ecx+12];
29: paddsd  xmm1, xmm2;
30: paddsd  xmm1, xmm3;
31: paddsd  xmm1, xmm4;
32: movdqa  [ebx], xmm1;        // ebx is the output memory address
33:
34: mov    xmm1, [edi];
35: // Multiply each of the rest of rows in the FIR filter coefficient table
36: // and update the temporary accumulated buffer
37: mov    eax, 19;              // set total number of iterations
38: loop:
39:     movdqa  xmm1, xmm0;
40:     pmullw  xmm1, [edx+16];
41:     paddsd  xmm1, [edi+16];  // edi is temporary accumulated buffer index
42:     movdqa  [edi], xmm1;     // store the temporary accumulated result
43:     add    edx, 16;          // next coefficient index
44:     add    edi, 32;         // next temporary accumulated buffer index
45:     dec    eax;
46:     jnz    loop;

4.2 Interference Cancellation

After \(P_b\) is decoded, it will be re-encoded and canceled out from the original signals. After that
we can decode $Pa$. The implementation is very similar to the interference nullifying. Due to time constraint, this part is not included in the report. Next we are going to apply multi-core streaming processing to provide sufficient capacity to meet the high processing requirements.

4.3 Multi-core Scheduling

From the processing requirement analysis, we know that single core is not sufficient to provide sufficient capacity, especially the processing requirement linearly increases with the number of overlapped frames the base station able to decode.

As discussed in session 2, PHY layer contains four basic functional blocks in a pipeline. These blocks differ in processing speed and in I/O data rates. A block is only ready to execute when it has sufficient input data from the previous block. Therefore, we need to schedule a block on multiple cores when it is ready.

Generally there are two approaches to schedule multiple core processing. One is to replicate the pipeline and each pipeline run on a core. The scheduler dispatches blocks of samples to the pipeline on a certain core. However, PHY layer processing exhibits data dependency, (for example, the digital filter depends on the 15 preceding samples in the input stream. It is difficult for the scheduler to make an efficient scheduling without knowing all of data dependency and the status of the processing on each core.

Another approach is the implement only one pipeline and dynamically assign ready blocks to available. However, this approach introduces high overhead in implementing multiple synchronized FIFO's for the communication between cores. Such frequent FIFO's and synchronization operations are not suitable for digital signal processing such small computational tasks.

The approach that I choose to implement PHY layer is not the general dynamic scheduling. Instead, I choose the static scheduling, which Sora directly supports. It is a simple scheduling, with the least amount of overhead. In the two core implementation, I statically assign the first two blocks – scrambler and QPSK mod – on one core and assign the last two blocks – DS-SS and Up Sampling – on the other core. And I only need to implement one single synchronized FIFO between these two cores. Illustration 5 shows the static scheduling for the PHY layer.
5. CONCLUSION

Spacial diversity exploitation is a good additional technique to increase the capacity in LAN multi-access. Although diversity gain implies an increase in computational complexity, it can be implemented even with general purpose CPUs.

The project gives me an excellent exercise in implementing algorithms. Prior to this project, I always thought that algorithm design is more important than implementation design. However, after this course and being working on this project, it starts to make me think differently – there are a number of important design decisions to make in order to implement an algorithm. It is first based on the platform that we choose to implement the algorithm. Different platform will have a different design choice. This makes us to study the processing requirement of the algorithm carefully, and take advantage of what the platform supports and optimize the performance.

Reference


