Fixed-Point FFT Signal Processing Technologies

CompMeth Spring 2018

Davide Pola

April 30, 2018

1 Introduction

The goal of this report is to illustrate the work done on a FFT simulator in for the CompMeth course in EURECOM. The functions developed are able to compute the FFT of a given signal, changing the input signal type, amplitude, number of points and some other parameters that will be described later.

2 Code Explanation

2.1 taus.c

The purpose of this file is to generate random variable starting from a seed. This seed could be either fixed or a randomly generated value.

modifications: As it was, the seed was generated from the current system time, down to a precision of 1 second. This is not exactly good as two consecutive simulations could run in less than one second and, as a consequence, show some correlation. This is the reason why now the random seed is initialized down to a precision of 1 microsecond. Also, in order to have a full deterministic behavior, another initialization function is able to set the seed to a fixed value.

2.2 rangen_double.c

This is used to generated normally distributed random value trough the function *gaussdouble*.

modifications: This piece of code was originally not working because it was developed for a 32-bit machine. Fixing the data types solved this issue.

2.3 complex.h

This file contains the definitions of data types used in the entire project. These can be double precision floating point numbers, 32-bit integers or 16-bit integers.

modifications: Like the previous file, this was not fully portable too, due to machine-dependent data types. Setting *int32_t* and *int16_t* should fix this portability issue for any machine. A comparison between the original and modified code can be found in Listings 1,2.

2.4 fixed_point.c

Here we define the functions used to compute fixed point saturated addition and multiplication. These functions are available both in a 16-bit and 25-bit version.

modifications: Nothing was changed here with the exception of data types.

2.5 fft.c

This is the main core of the application, where all the processing of the FFT is done. Each function will be separately covered in this section, together with some brief explanation of their meaning in the overall FFT computation.

2.5.1 Twiddle factor

The functions:

- void twiddle(struct complex *W, int N, double stuff)
- void twiddle_fixed(struct complex16 *W, int N, double stuff)
- void twiddle_fixed_Q17(struct complex32 *W, int N, double stuff)

Are used to compute the twiddle factor. As the prototype indicates, each function is required whether we are using floating points, Q15 or Q24 representations.
Following these formula, it's straightforward to understand the computations done in this function and the role of the *stuff* parameter.

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_k W_N^{kn} \tag{1}$$

$$W_N^{kn} = e^{\frac{-j2\pi kn}{N}} = \cos(2\pi kn) - j\sin(2\pi kn)$$

$$stuff = kn$$

modifications: At the beginning, only the floating point twiddle factor function was present. The code can be found at the end of this document in Listing 3

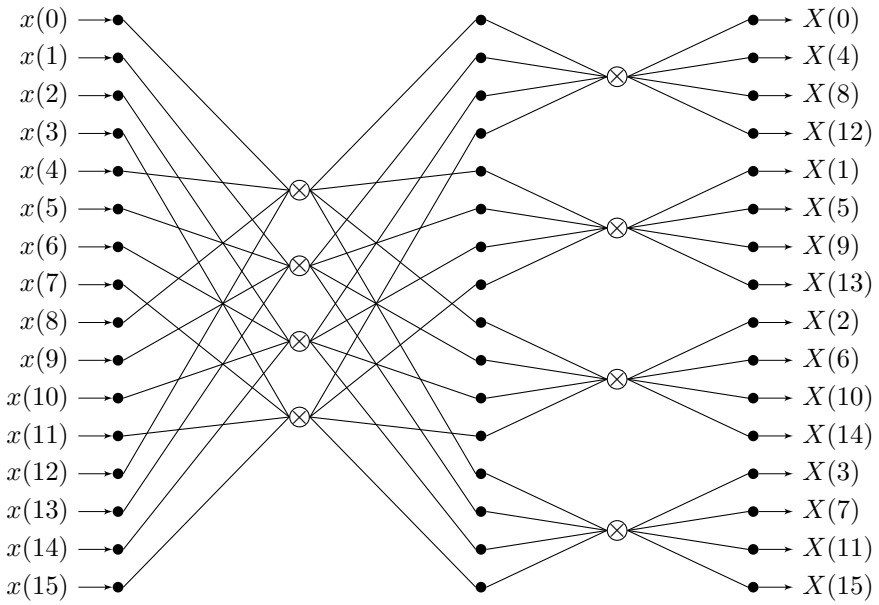
quantization: As can be seen in the code, in the fixed point version of the function, a scaling factor is needed to convert from floating to fixed point. (i.e. ± 32767.0 for 16-bit). The range for a Q15 fixed point number goes from -1 to $1 - 2^{-15}$, as a consequence, it is intrinsically not possible to represent exactly the number 1.
Radix-4 twiddle factors have the characteristic that the real part ranges from 0 to 1, and the imaginary part from 0 to -1. The programmer is left with two options: either produce an unbiased result, using ± 32767.0, or use -32768.0 only for imaginary part and $+32767.0$ for the real part.
Trying both of them led to a slightly better result for the biased version, so it was the one left in the code.

Another important choice is between rounding and truncation. The C standard for typecasting is truncation, thus in order to produce a rounded value it would be necessary to add ± 0.5 to the final result before casting.

2.5.2 Coefficients reordering

The following functions are needed because after the recursion of the radix 4 FFT butterfly, the coefficients are shuffled in a precise pattern. An simple example of a 16 points radix 4 butterfly can be seen in the following image.
Again there are three functions used depending on the data type:

- bit_r4_reorder(struct complex *W, int N)
- bit_r4_reorder_fixed_Q15(struct complex16 *W, int N)
- bit_r4_reorder_fixed_Q24(struct complex32 *W, int N)



modifications A weird bug was discovered and fixed in the fixed point version of the function. As can be seen in the two code listings, the original code was applying the scaling only to those coefficients that were swapped. Fixing this bug strongly improved the output SNR, especially in the low dB input where the scaling is more likely to be executed at the reordering phase. A comparison between the two codes can be found in Listings 4,5.

2.5.3 Radix 4

This is the core function where the FFT coefficients are actually calculated. The function is recursive, and calls itself back until no more splitting is possible(when there is only one butterfly left), every time cutting the number of iterations by ¼.

shifting At each stage, the input data might be right shifted by a scale value, ranging from 0 to $log_2(\sqrt{N})$. This is because, according to (1), final coefficients must be divided by a factor $\sqrt{N}$. This shift can be split in all the stages of the Radix-4 FFT in order to achieve the maximum possible precision with the least number of saturations.

- radix4(struct complex *x, int N)
- void radix4_fixed_Q15(struct complex16 *x, int N, uns char *scale, uns char stage)
- void radix4_fixed_Q24(struct complex32 *x, int N, uns char *scale, uns char stage)

As for the other functions this one is also present three times, depending on input data type.

modifications Other than constructing the functions for fixed point arithmetics, a simple control on the twiddle factor was developed in order to skip multiplications when the twiddle factor is equal to 1. This would slightly reduce computation time and, more important, avoid possible precision losses. The code of the 16-bit fixed point version can be found in Listing 6

2.5.4 QAM input

This helper function is called in the two cases when PSK or QAM input is needed, it will simply fill the input vector with the right numbers. This function was left untouched.

2.5.5 distortion test

The purpose of this function is to run a single simulation of an FFT both in double precision arithmetic and one between 16-bit or 25-bit fixed point arithmetic and produce an output SNR value between the two. The formula used to calculate the SNR is:

$$SNR = 10 \log_{10} \left(\frac{\sum_{n=0}^{N-1} |X_n^f - X_n^d|^2}{\sum_{n=0}^{N-1} |X_n^f|^2} \right)$$

Where X_n^f is the n-th value of the floating point FFT and X_n^d is the fixed point version of the same value.

Also at the end of the computation, if the current SNR is better than the stored maximum, that value is updated(*maxSNR*) and the current scaling is saved(*maxscale*).

As shown in the prototype, many other parameters are required. Most of them are pretty self explained by the comments. One parameter worth mentioning is *test*. It is an integer value passed from the calling function which is used to generate the input signal according to the following scheme:

0 → fixed frequency cosine signal

1 → PSK white noise signal

2 → 16QAM white noise signal

3 → Gaussian white noise signal

Most of the code in this function was left untouched, with the exception of the extension to Q24 fixed point structure, leading to the inclusion of *data_t* and *shift* parameters. The first is a simple selector of data type, while the latter will be discussed later in 3.4.

```
void fft_distortion_test(
int N,                          // dimension of FFT under test
char test,                      // type of test
double input_dB,                // strength of input
char *scale,                    // pointer to scaling schedule
double *maxSNR,                 // pointer best signal-to-noise ratio
char *maxscale,                 // pointer to best scaling schedule
struct complex *data,           // pointer to floating point data vector
struct complex16 *data16,       // pointer to Q15 data vector
struct complex32 *data32,       // pointer to Q17 data vector
int data_t,                     // 16 or 25 bit input
int shift)                      // 25-bit only pre/post shift
```

2.5.6 main

The purpose of the main function, other than obviously interfacing with the user and extrapolate input parameters, is to actually launch the executions of the simulation. Considering that one strong factor of the output quality is the scaling schedule(previously mentioned in 2.5.3), the main function simply tries all the possible combinations in an exhaustive manner and prints out only the best one. This is also the reason why the final execution time is so long, the complexity of the search is:

$$\frac{(2n)!}{(n!)^2}$$

where:

$$n = log_2 \sqrt{N}$$

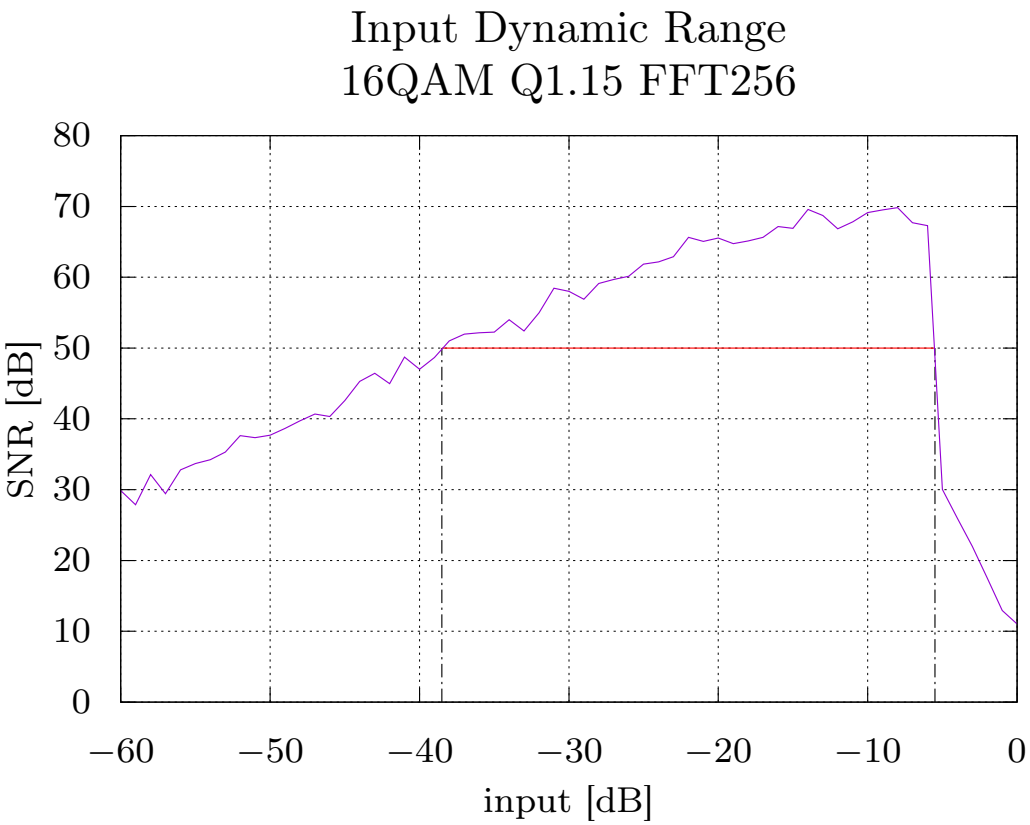


Figure 1: Input dynamic range calculation.

3 Results

As suggested in the instruction, it is interesting to see the input signal strength versus the signal-to-distortion ratio to find the practical dynamic range. As there are no given numbers to use to decide the acceptable SNR threshold, a 50dB value has been chosen.

3.1 Calculate the Input Dynamic Range

As show in Figure 1, calculating the input dynamic range is not difficult. Every input signal shows almost the same behavior: the SNR is roughly increasing linearly with the input dB, while at a certain point there is a sudden drop on the output quality. The input dynamic range is simply the range for which the output is higher than the threshold. In the example it spaces from -38dB to -6dB.

3.2 Fixed Point Saturation

One particular drawback of fixed point representation is saturation. In our case this particular phenomenon can happen in case of additions for which the result inevitably fall outside of the possible range. Fortunately, as discussed in 2.5.3, FFT shifting helps to overcome this problem. However, at some point, even after the maximum scaling applied directly at the input, the output will saturate. Moving to a 25-bit representation could enlarge the input dynamic range and provide good results where the 16-bit reaches its limit.

Figure 2 shows on the left the cosine input signal, and on the right the same FFT computed in three different bit representations. At this point signal input strength Q1.15 is no more able to avoid saturations, and the resulting FFT spectrum is unavoidably compromised. Q10.15 instead is still in its dynamic range and provide an almost identical spectrum.

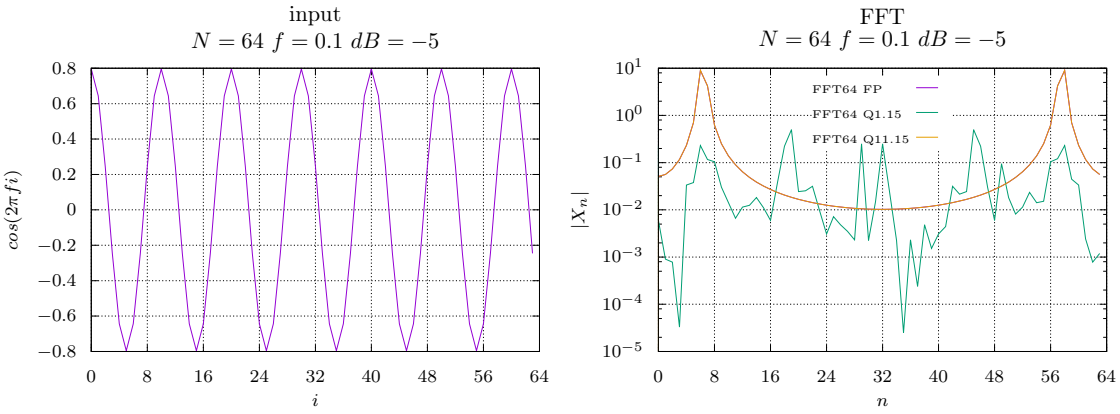


Figure 2: Input dynamic range calculation.

3.3 FFT Output Comparison

First of all it is interesting to see the output of the program with respect to the variation of input signal and FFT size. Figure 3 shows the output of the 16-bit fixed point simulation.
One notable aspect is that the input dynamic range of the cosine is more dependent on the FFT size compared to the others. This is probably due to the particular Fourier transform of the cosine, which is composed of only two components. These component are noticeably higher than those that could be found in the FFT of a pure noise signal, and clearly will produce many more saturations.

3.4 25-bit Decimal Representation

In order to properly compare the two fixed point representations, the input source needs to be exactly the same(in this case is set as 16 bits of precision). In order to put these 16 bits into a 25-bit numbers without cutting out digits, many different solutions are possible. If we want to keep the same magnitude as the 16-bit version, then a Q1.24 is the only solution, but it is possible to assign more bits to the integer value, such as Q2.23 or Q5.20, down to Q10.15. The results of this comparison are shown in Figure 4 (please note that the input range is different than the others graphs found in this report).

As can be seen from the graph, the maximum acceptable input strength increases together with the number of bits of the integer part, topping at 35dB with Q10.15. One less evident but still important aspect is that, apart from saturation, with the same input signal, having more fractional bits produces better results in terms of SNR.

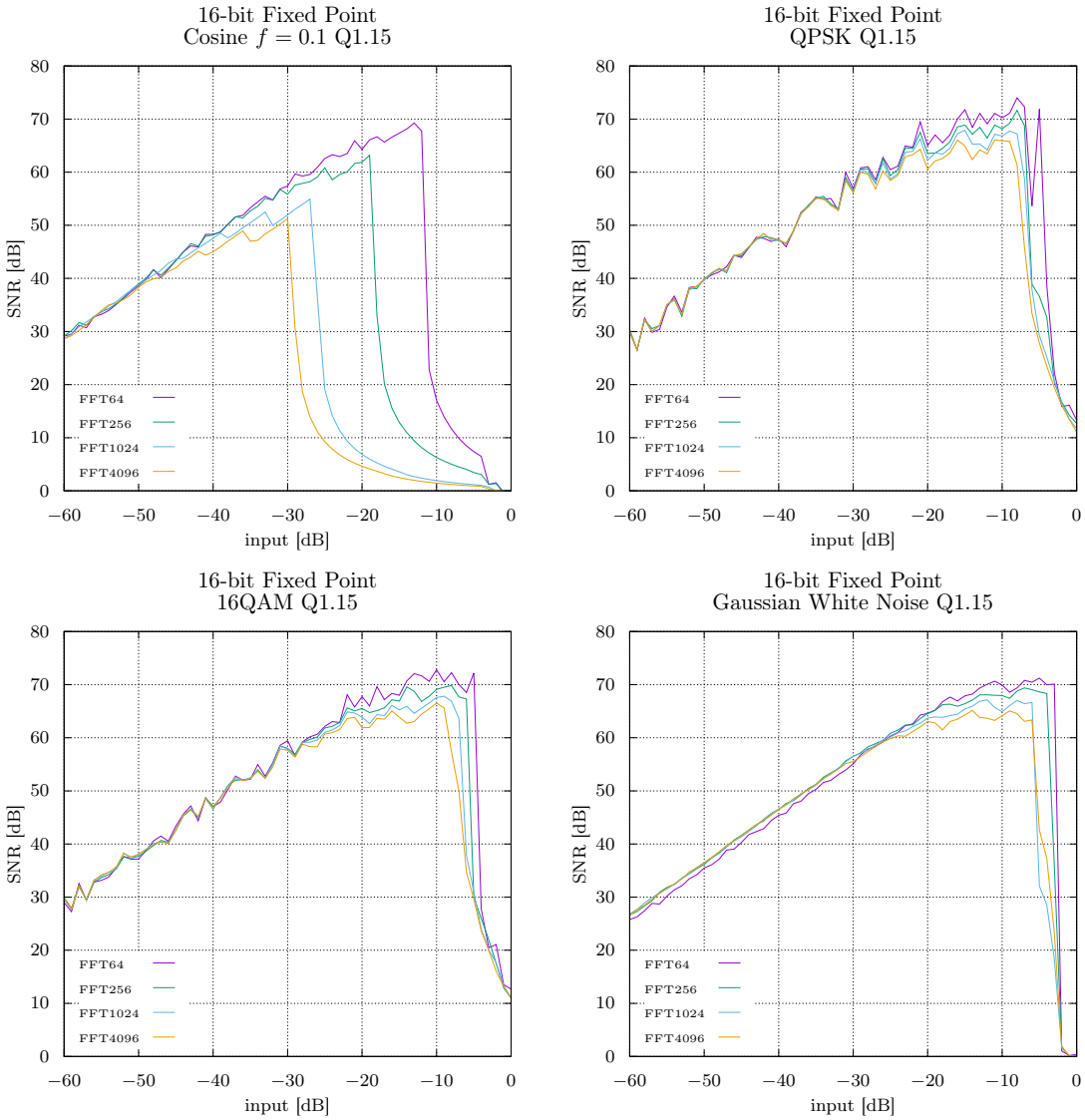


Figure 3: Output comparison.

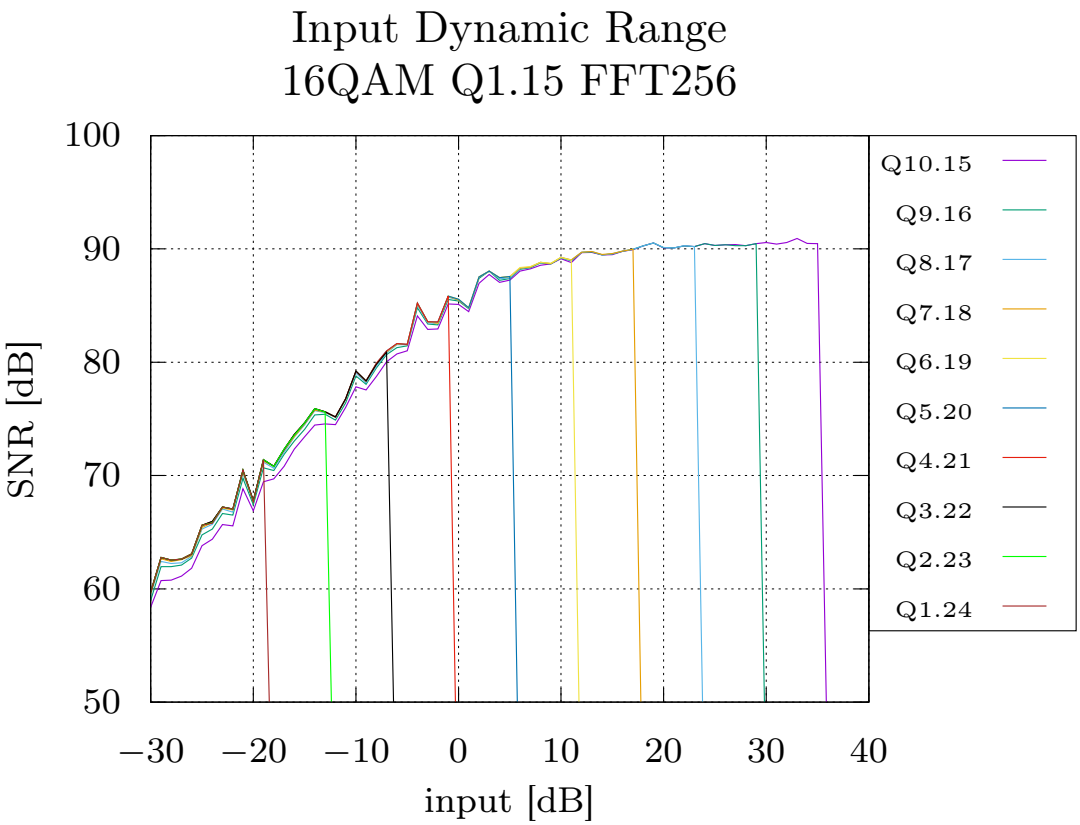


Figure 4: Comparison of 25-bit fixed point representations.

4 Code

Listing 1: complex.h - original

```
struct complex{
    double r;
    double i;
    };

struct complex16{
    short r;
    short i;
};

struct complex32{
    int r;
    int i;
};
```

Listing 2: complex.h - modified

```
struct complex{
    double r;
    double i;
};

struct complex16{
    int16_t r;
    int16_t i;
};

struct complex32{
    int32_t r;
    int32_t i;
};
```

Listing 3: fft.c - twiddle factors

```
void twiddle(struct complex *W, int N, double stuff){
    W->r=cos(stuff*2.0*PI/(double)N);
    W->i=-sin(stuff*2.0*PI/(double)N);
}

void twiddle_fixed(struct complex16 *W, int N, double stuff){
    W->r=(int16_t)(32767.0*cos(stuff*2.0*PI/(double)N));
    W->i=(int16_t)(-32768.0*sin(stuff*2.0*PI/(double)N));
    }

void twiddle_fixed_Q17(struct complex32 *W, int N, double stuff){
    W->r=(int32_t)(131071.0*cos(stuff*2.0*PI/(double)N));
W->i=(int32_t)(-131072.0*sin(stuff*2.0*PI/(double)N));
}
```

Listing 4: fft.c - bit reordering - original

```
void bit_r4_reorder_fixed_Q15(
struct complex16 *W,
int N,
char scale)
{
  int bits, i, j, k;
  short tempr, tempi;

  for (i=0; i<MAXPOW; i++)
    if (pow_2[i]==N) bits=i;

  for (i=0; i<N; i++){
    j=0;
    for (k=0; k<bits; k+=2){
      ...
    }

    if (j>i){
      tempr=W[i].r>>scale;
      tempi=W[i].i>>scale;
      W[i].r=W[j].r>>scale;
      W[i].i=W[j].i>>scale;
      W[j].r=tempr;
      W[j].i=tempi;
    }
  }
}
```

Listing 5: fft.c - bit reordering - modified

```
void bit_r4_reorder_fixed_Q15(
struct complex16 *W,
int N,
char scale)
{
  int bits, i, j, k;
  int16_t tempr, tempi;

  for(i=0; i<N; i++){
    W[i].r=W[i].r>>scale;
      W[i].i=W[i].i>>scale;
      }

  for (i=0; i<MAXPOW; i++)
    if (pow_2[i]==N) bits=i;

  for (i=0; i<N; i++){
    j=0;
      for (k=0; k<bits; k+=2){
        ...
      }

    if (j>i){
      tempr=W[i].r;
      tempi=W[i].i;
      W[i].r=W[j].r;
      W[i].i=W[j].i;
      W[j].r=tempr;
      W[j].i=tempi;
    }
  }
}
```

Listing 6: fft.c - radix4 Q15

```
void radix4_fixed_Q15(struct complex16 *x,    // Input in Q15 format
int N,                    // Size of FFT
unsigned char *scale,   // Pointer to scaling schedule
unsigned char stage)    // Stage of fft
{
int     n2, k1, N1, N2;
struct complex16 W, bfly[4];

N1=4;
N2=N/4;

// Do 4 Point DFT
for (n2=0; n2<N2; n2++){
    // scale Butterfly input
x[n2].r           >>= scale[stage];
...
x[(3*N2) + n2].i >>= scale[stage];

// Radix 4 Butterfly
bfly[0].r = SAT_ADD16(  SAT_ADD16(x[n2].r, x[N2 + n2].r) ,
            SAT_ADD16(x[2*N2+n2].r, x[3*N2+n2].r));
bfly[0].i = SAT_ADD16(  SAT_ADD16(x[n2].i, x[N2 + n2].i) ,
            SAT_ADD16(x[2*N2+n2].i, x[3*N2+n2].i));
bfly[1].r = SAT_ADD16(  SAT_ADD16(x[n2].r, x[N2 + n2].i) ,
          -SAT_ADD16(x[2*N2+n2].r, x[3*N2+n2].i));
bfly[1].i = SAT_ADD16(  SAT_ADD16(x[n2].i, -x[N2 + n2].r) ,
            SAT_ADD16(-x[2*N2+n2].i, x[3*N2+n2].r));
bfly[2].r = SAT_ADD16(  SAT_ADD16(x[n2].r, -x[N2 + n2].r) ,
            SAT_ADD16(x[2*N2+n2].r, -x[3*N2+n2].r));
bfly[2].i = SAT_ADD16(  SAT_ADD16(x[n2].i, -x[N2 + n2].i) ,
            SAT_ADD16(x[2*N2+n2].i, -x[3*N2+n2].i));
bfly[3].r = SAT_ADD16(  SAT_ADD16(x[n2].r, -x[N2 + n2].i) ,
            SAT_ADD16(-x[2*N2+n2].r, x[3*N2+n2].i));
bfly[3].i = SAT_ADD16(  SAT_ADD16(x[n2].i, x[N2 + n2].r) ,
            SAT_ADD16(-x[2*N2+n2].i, -x[3*N2+n2].r));

// In-place results
x[n2].r = bfly[0].r;
x[n2].i = bfly[0].i;

for (k1=1; k1<N1; k1++){
    twiddle_fixed(&W, N, (double)k1*(double)n2);
    x[n2 + N2*k1].r = SAT_ADD16( FIX_MPY(bfly[k1].r, W.r) ,
                      -FIX_MPY(bfly[k1].i, W.i) );
    x[n2 + N2*k1].i = SAT_ADD16( FIX_MPY(bfly[k1].i, W.r) ,
                       FIX_MPY(bfly[k1].r, W.i) );
}
}

// Don't recurse if we're down to one butterfly
if (N2!=1)
for (k1=0; k1<N1; k1++){
    radix4_fixed_Q15(&x[N2*k1], N2,scale,stage+1);
}
}
```