Your Paper

You

March 23, 2018

Abstract

Your abstract.

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 a 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.

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.

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.

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

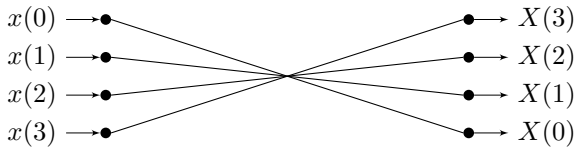
$$W_N^{kn} = e^{\frac{-j2\pi kn}{N}} = \cos(2\pi kn) - j\sin(2\pi kn)$$

$$stuff = kn$$

2.5.2 Coefficients reordering

The following functions are needed because after the computation of the radix 4 FFT butterfly, the coefficients are shuffled in a precise pattern. 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)



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.

- radix4(struct complex *x, int N)
- TODO
- TODO

2.5.4 QAM input

2.5.5 distortion test

2.5.6 main

3 Code

3.1 complex.h

Listing 1: code 1 1

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

    struct complex16
    {
    short r;
    short i;
};

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

Listing 2: code 2

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

    struct complex16{
        int16_t r;
    int16_t i;
};

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

3.2 fft.c

Listing 3: code 3

```
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: code 4

```
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 5: code 5

```
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 6: code 5

```
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[N2+n2].r        >>= scale[stage];
x[(2*N2) + n2].r  >>= scale[stage];
x[(3*N2) + n2].r  >>= scale[stage];
x[n2].i           >>= scale[stage];
x[N2+n2].i        >>= scale[stage];
x[(2*N2) + n2].i  >>= 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);
}
}
```

References