ZedBoard Project – Task 1

Xu Shipeng

Suman Deb

Contents

Project Introduction

In this project, we read input from the MIC, calculate the FFT, get the absolute value from FFT output, average between windows and display them on OLED display. We also profile our project.

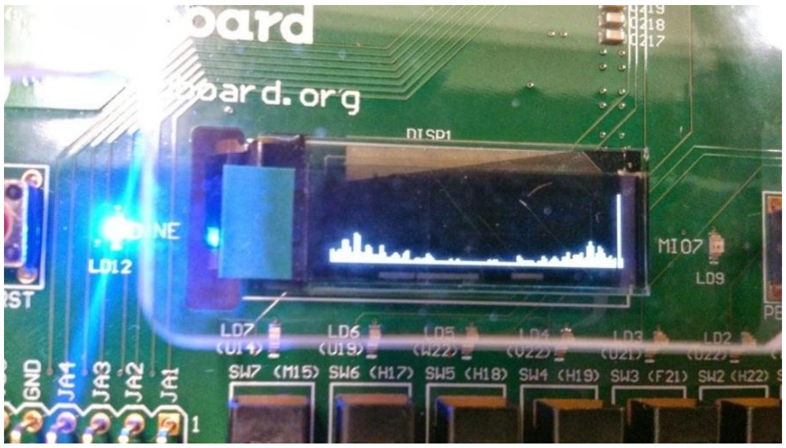


Figure 1 FFT result of human voice from audio input

Video Demo of our system:

http://www.youtube.com/watch?v=vZl1M3n6_3s

Project in GitHub:

https://github.com/billhsu/FPGAFFT/

System Implementation

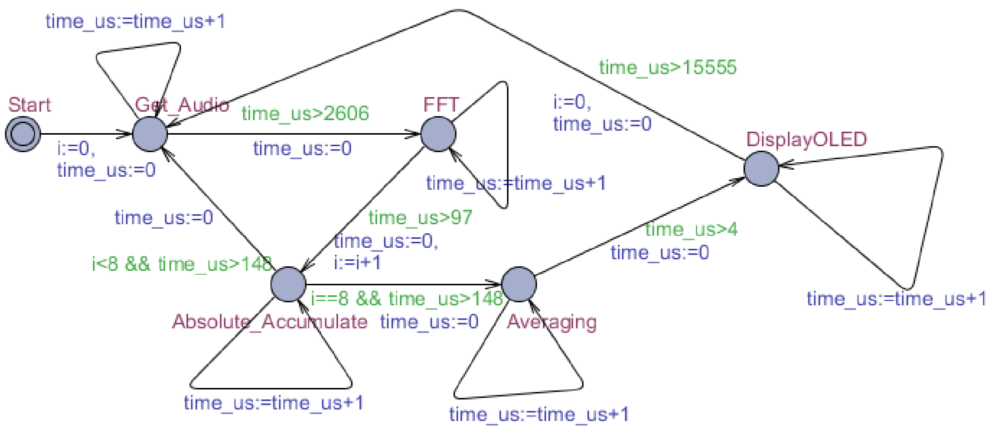


Figure 2 System Flow

Figure 2 is the system flow. It could be descripted as following pseudo-code:

```
while(1){
        for(window = 0; window<WINDOW; ++window)
        {
                get_audio();
                FFT();
                Get_Abs_and_Accum();
        }
        Averaging();
        Display();
}
```

We tried three FFT implementations to compare the performance between them.

The three FFT implementations are:

1. Kiss FFT(http://sourceforge.net/projects/kissfft/)
2. 16 Bit Fixed point FFT (http://www.jjj.de/fft/fftpage.html)
3. Floating point FFT (http://www-ee.uta.edu/eeweb/ip/Courses/DSP_new/Programs/fft.cpp)

Profiling and Timed Automata

Profiling

We profile our system in two ways: using hardware timer counter and using gprof. We use gprof to verify the profiling result by counting hardware timer.

Profiling by Hardware Timer Counter

The hardware timer is defined in xtime_l.h, so firstly we need to include the header file:

```
#include "xtime_l.h"
```

Then we subtract the current counter value with the counter value before entering the function to get the time elapse of this function.

```
XTime_GetTime(&tCur);
fft(datacpx,N,1);
XTime_GetTime(&tEnd);
tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
printf("FFT %d us\r\n",tUsed);
```

1. Profiling result for 16 bit fixed point FFT implementation:

*Get audio input (one time): 2606 us

*FFT (one time): 165 us

*Get absolute value (one time): 37 us

Averaging: 4 us

OLED Display: 15555 us

2. Profiling result for floating point FFT implementation:

*Get audio input (one time): 2606 us

*FFT (one time): 97 us

*Get absolute value (one time): 148 us

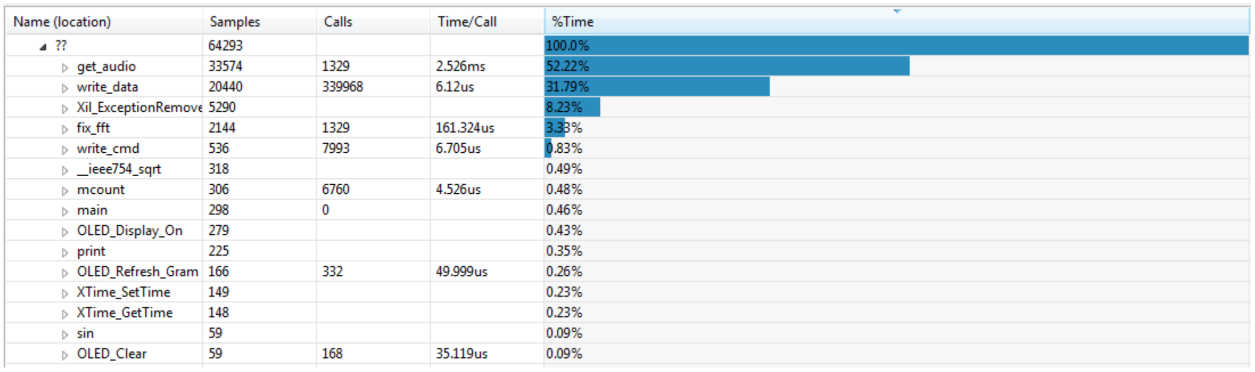
Averaging: 4 us

OLED Display: 15555 us

(*: This is the time elapse for just one time. Totally 8 times are required for one complete loop.)

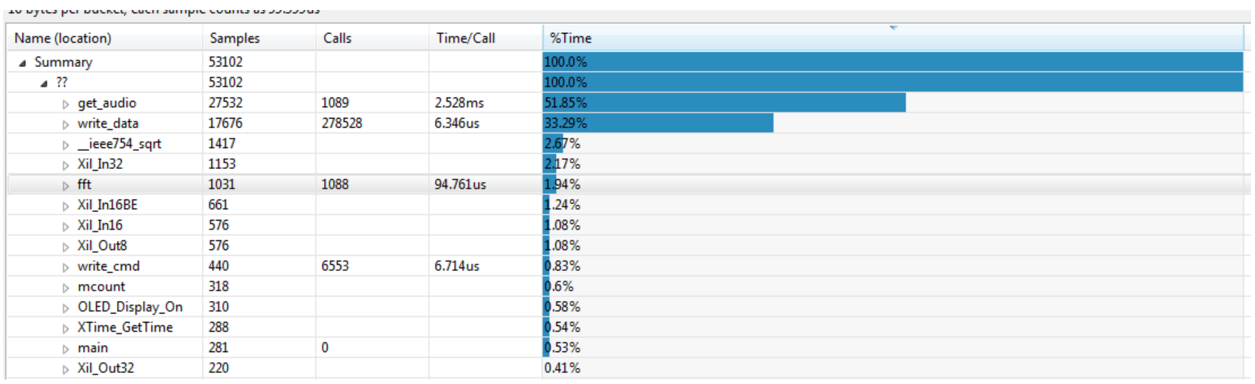
Profiling by gprof

We also used gprof to verify the previous proofing result.



Name (location)	Samples	Calls	Time/Call	%Time
◢ ??	64293			100.0%
▷ get_audio	33574	1329	2.526ms	52.22%
▷ write_data	20440	339968	6.12us	31.79%
▷ Xil_ExceptionRemove	5290			8.23%
▷ fix_fft	2144	1329	161.324us	3.33%
▷ write_cmd	536	7993	6.705us	0.83%
▷ __ieee754_sqrt	318			0.49%
▷ mcount	306	6760	4.526us	0.48%
▷ main	298	0		0.46%
▷ OLED_Display_On	279			0.43%
▷ print	225			0.35%
▷ OLED_Refresh_Gram	166	332	49.999us	0.26%
▷ XTime_SetTime	149			0.23%
▷ XTime_GetTime	148			0.23%
▷ sin	59			0.09%
▷ OLED_Clear	59	168	35.119us	0.09%

Figure 3 gprof result for fixed point implementation



Name (location)	Samples	Calls	Time/Call	%Time
◢ Summary	53102			100.0%
◢ ??	53102			100.0%
▷ get_audio	27532	1089	2.528ms	51.85%
▷ write_data	17676	278528	6.346us	33.29%
▷ __ieee754_sqrt	1417			2.67%
▷ Xil_In32	1153			2.17%
▷ fft	1031	1088	94.761us	1.94%
▷ Xil_In16BE	661			1.24%
▷ Xil_In16	576			1.08%
▷ Xil_Out8	576			1.08%
▷ write_cmd	440	6553	6.714us	0.83%
▷ mcount	318			0.6%
▷ OLED_Display_On	310			0.58%
▷ XTime_GetTime	288			0.54%
▷ main	281	0		0.53%
▷ Xil_Out32	220			0.41%

Figure 4 gprof results for floating point implementation

Form the gprof result, we can see that the fixed point fft is 161us, floating point fft is 94us. So our profiling result with timer is correct.

Analyze

From the above results, what surprises us is that the fixed point version takes longer time in FFT than the floating point version. The algorithm in both implementations is the same. So our conclusion is that the ARM9 core in ZedBoard has a FPU to accelerate the floating point calculations. The fixed point version is slower because it used more instructions than the floating point version to do the computation.

Execution time won't vary depending on the real numbers in the input buffer.

The FFT and get the absolute value calculation seems fast enough with this software only solution. Bottlenecks of this system are getting the audio data and displaying the result to OLED. We may improve the bottlenecks by introducing hardware acceleration technologies.

Timed automata

We created this timed automata with uppaal. Figure 5 and Figure 6 are the screen capture of the timed automata for 16 bit fixed point implementation and floating point implementation. We simulated the model in uppaal and the simulation results verified that our system is working properly.

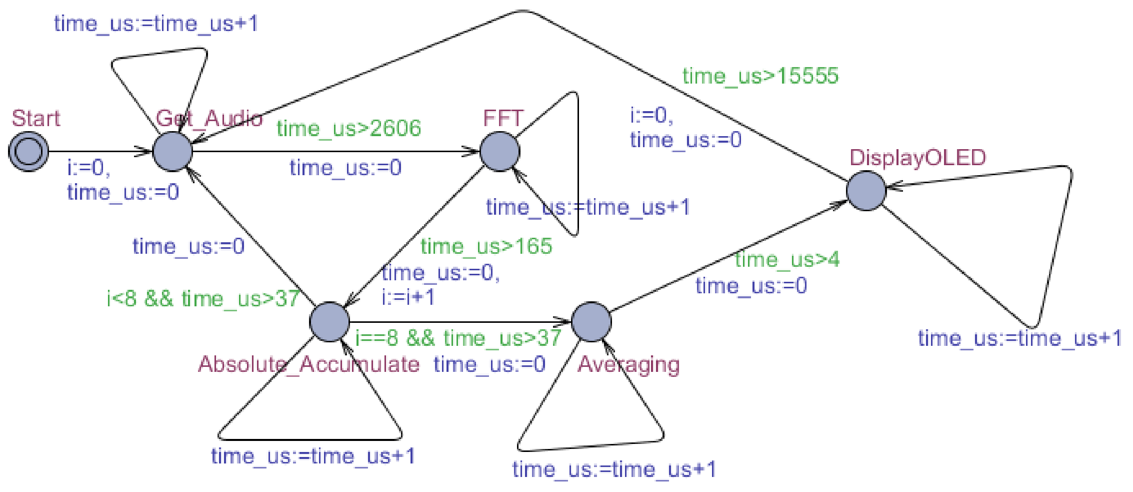


Figure 5 16 bit fixed point implementation

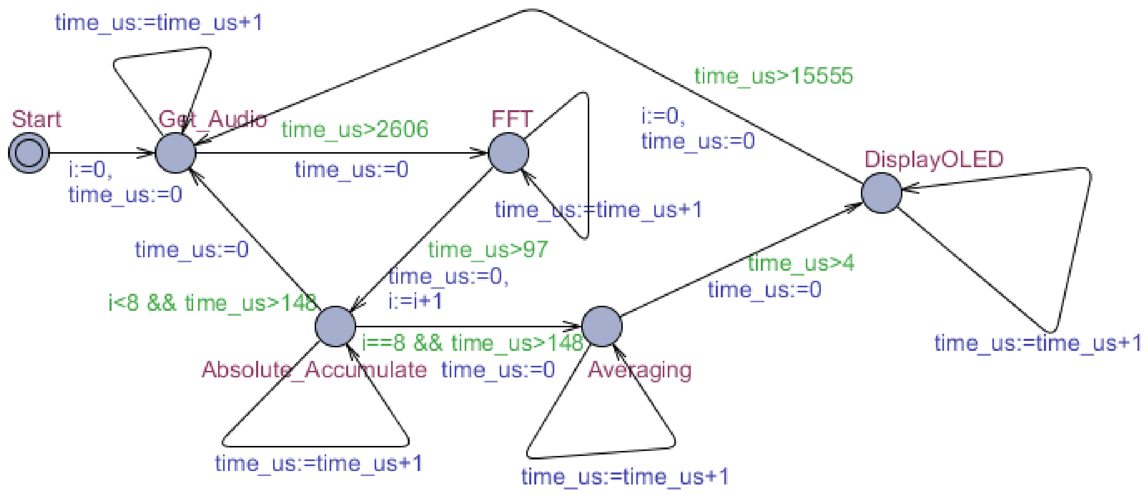


Figure 6 Floating point implementation

Appendix

zynq_audio_fft.c

```c
#include <stdint.h>
#include <stdio.h>
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <math.h>
#include "fft.h"
#include "audio.h"
#include "oled.h"
#include "sleep.h"
#include "xtime_l.h"
#include "xpm_counter.h"
#include "xparameters.h"
#define N 128
#define POW_N 7
#define WINDOW 8
#define FIX_POINT


int main(void)
{
    short real[N], image[N];
    float datacpx[N];
    int outLED[N];
    Xint16 audio_data[128];
    int i,j,window;
    XTime tEnd, tCur;
    u32 tUsed;
    int cycle=0;
    print("FFT start\n");

    u8 *oled_equalizer_buf=(u8 *)malloc(128*sizeof(u8));
         Xil_Out32(OLED_BASE_ADDR,0xff);
         OLED_Init();                          //oled init
         IicConfig(XPAR_XIICPS_0_DEVICE_ID);
         AudioPllConfig(); //enable core clock for ADAU1761
         AudioConfigure();

         xil_printf("ADAU1761 configured\n\r");

         //srand(time(0));
         while(1){
                  OLED_Clear();
                  printf("Cycle %d################################\n",++cycle);
```

```
                    for (i = 0; i < N; i++) outLED[i] = 0;
                    for(window = 0; window<WINDOW; ++window)
                    {
                              //Generate input data
                              XTime_GetTime(&tCur);
                              get_audio(audio_data);

                              for (i = 0; i < N; i++)
                              {
#ifdef FIX_POINT
                                        //real[i] = ((rand()%128-64)/128.0f)*(1<<14);
                                        real[i] = audio_data[i];
                                        image[i] = 0;
#else
                                        //datacpx[2*i]=(rand()%128-64)/128.0f;
                                        datacpx[2*i]=((float)(audio_data[i])/200.0f);
                                        datacpx[2*i+1] = 0;
#endif
                              }

                              XTime_GetTime(&tEnd);
                              tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
                              printf("get input %d us\r\n",tUsed);

                              //FFT
                              XTime_GetTime(&tCur);
#ifdef FIX_POINT
                              fix_fft(real, image, POW_N);
#else
                              fft(datacpx,N,1);
#endif
                              XTime_GetTime(&tEnd);
                              tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
                              printf("FFT %d us\r\n",tUsed);

                              //Conj
                              XTime_GetTime(&tCur);
                              for (i = 0; i < N; i++)
                              {
#ifdef FIX_POINT
                                        int conj_pdt_out =sqrt((real[i]*real[i]) +
(image[i]*image[i]));
#else
                                        int conj_pdt_out =sqrt((datacpx[2*i]*datacpx[2*i]) +
(datacpx[2*i+1]*datacpx[2*i+1]));
#endif
                                        //conj_pdt_out=conj_pdt_out/2;
                                        outLED[i]+=conj_pdt_out;
                    }
                    XTime_GetTime(&tEnd);
                    tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
                    printf("Conj %d us\r\n",tUsed);
                }

                //Averaging
                XTime_GetTime(&tCur);
                for(i=0;i<N;++i)
                {
#ifdef FIX_POINT
                    oled_equalizer_buf[i]=outLED[i]>>2;
#else
                    oled_equalizer_buf[i]=outLED[i]/8;
#endif
                }
                XTime_GetTime(&tEnd);
                tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
                printf("Averaging %d us\r\n",tUsed);

                //Display
                XTime_GetTime(&tCur);
                OLED_Equalizer_128(oled_equalizer_buf);
                OLED_Refresh_Gram();
                XTime_GetTime(&tEnd);
                tUsed = ((tEnd-tCur)*1000000)/(COUNTS_PER_SECOND);
                printf("Display %d us\r\n",tUsed);
        }
    return 0;
}
```

fft.c

```
#include "fft.h"
#define N_WAVE      1024    // full length of Sinewave[]
#define LOG2_N_WAVE 10      // log2(N_WAVE)

// Since we only use 3/4 of N_WAVE, we define only
// this many samples, in order to conserve data space.
const short Sinewave_FIX[N_WAVE-N_WAVE/4] = {
      0,    201,    402,    603,    804,   1005,   1206,   1406,
   1607,   1808,   2009,   2209,   2410,   2610,   2811,   3011,
...
...
 -32609, -32628, -32646, -32662, -32678, -32692, -32705, -32717,
 -32727, -32736, -32744, -32751, -32757, -32761, -32764, -32766,
};
```

```
//16 bit Fixed point FFT
void fix_fft(short fr[], short fi[], short m)
{
         long int mr = 0, nn, i, j, l, k, istep, n, shift;
         short qr, qi, tr, ti, wr, wi;

         n = 1 << m;
         nn = n - 1;
         /* max FFT size = N_WAVE */
         //if (n > N_WAVE) return -1;
         /* decimation in time - re-order data */
         for (m=1; m<=nn; ++m)
         {
                  l = n;
                  do
                  {
                           l >>= 1;
                  } while (mr+l > nn);

                  mr = (mr & (l-1)) + l;
                  if (mr <= m) continue;

                  tr = fr[m];
                  fr[m] = fr[mr];
                  fr[mr] = tr;
                  ti = fi[m];
                  fi[m] = fi[mr];
                  fi[mr] = ti;
         }

         l = 1;
         k = LOG2_N_WAVE-1;

         while (l < n)
         {
                  long int c;
                  short b;

                  istep = l << 1;
                  for (m=0; m<l; ++m)
                  {
                           j = m << k;
                           /* 0 <= j < N_WAVE/2 */
                           wr =  Sinewave_FIX[j+N_WAVE/4];
                           wi = -Sinewave_FIX[j];

                           wr >>= 1;
                           wi >>= 1;
```

```
                        for (i=m; i<n; i+=istep)
                        {
                                j = i + l;

                                // Here I unrolled the multiplications to prevent
overhead
                                // for procedural calls (we don't need to be clever about
                                // the actual multiplications since the pic has an
onboard
                                // 8x8 multiplier in the ALU):

                                // tr = FIX_MPY(wr,fr[j]) - FIX_MPY(wi,fi[j]);
                                c = ((long int)wr * (long int)fr[j]);
                                c = c >> 14;
                                b = c & 0x01;
                                tr = (c >> 1) + b;

                                c = ((long int)wi * (long int)fi[j]);
                                c = c >> 14;
                                b = c & 0x01;
                                tr = tr - ((c >> 1) + b);

                                // ti = FIX_MPY(wr,fi[j]) + FIX_MPY(wi,fr[j]);
                                c = ((long int)wr * (long int)fi[j]);
                                c = c >> 14;
                                b = c & 0x01;
                                ti = (c >> 1) + b;

                                c = ((long int)wi * (long int)fr[j]);
                                c = c >> 14;
                                b = c & 0x01;
                                ti = ti + ((c >> 1) + b);

                                qr = fr[i];
                                qi = fi[i];
                                qr >>= 1;
                                qi >>= 1;

                                fr[j] = qr - tr;
                                fi[j] = qi - ti;
                                fr[i] = qr + tr;
                                fi[i] = qi + ti;
                        }
                }

                --k;
                l = istep;
```

```
        }
}
//floating point FFT
void fft(float data[], int nn, int isign)
{
    int n, mmax, m, j, istep, i;
    float wtemp, wr, wpr, wpi, wi, theta;
    float tempr, tempi;
    n = nn << 1;
    j = 1;
    for (i = 1; i < n; i += 2) {
         if (j > i) {
             tempr = data[j];     data[j] = data[i];     data[i] = tempr;
             tempr = data[j+1]; data[j+1] = data[i+1]; data[i+1] = tempr;
         }
         m = n >> 1;
         while (m >= 2 && j > m) {
             j -= m;
             m >>= 1;
         }
         j += m;
    }
    mmax = 2;
    while (n > mmax) {
         istep = 2*mmax;
         theta = TWOPI/(isign*mmax);
         wtemp = sin(0.5*theta);
         wpr = -2.0*wtemp*wtemp;
         wpi = sin(theta);
         wr = 1.0;
         wi = 0.0;
         for (m = 1; m < mmax; m += 2) {
             for (i = m; i <= n; i += istep) {
                  j =i + mmax;
                  tempr = wr*data[j]   - wi*data[j+1];
                  tempi = wr*data[j+1] + wi*data[j];
                  data[j]   = data[i]   - tempr;
                  data[j+1] = data[i+1] - tempi;
                  data[i] += tempr;
                  data[i+1] += tempi;
             }
             wr = (wtemp = wr)*wpr - wi*wpi + wr;
             wi = wi*wpr + wtemp*wpi + wi;
         }
         mmax = istep;
    }
}
```