4 Observations

4.1 Resistive Adder

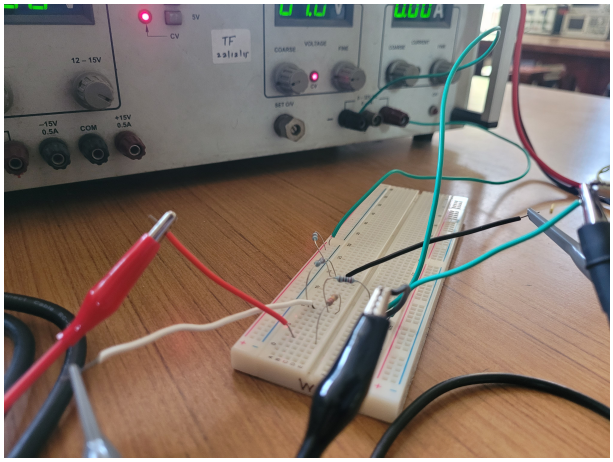

Figure 13: Breadboard design

Figure 14: Results

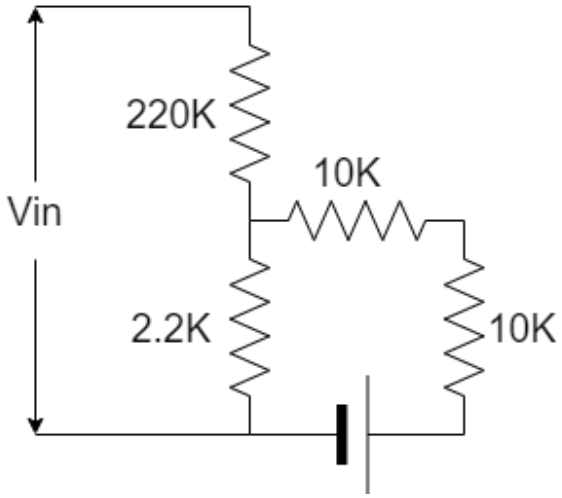

Figure 15: Circuit Diagram

The offset we wanted was 0.5V in the example we were testing but an offset of 0.3V was observed

4.2 Voltage adder using single supply Opamp

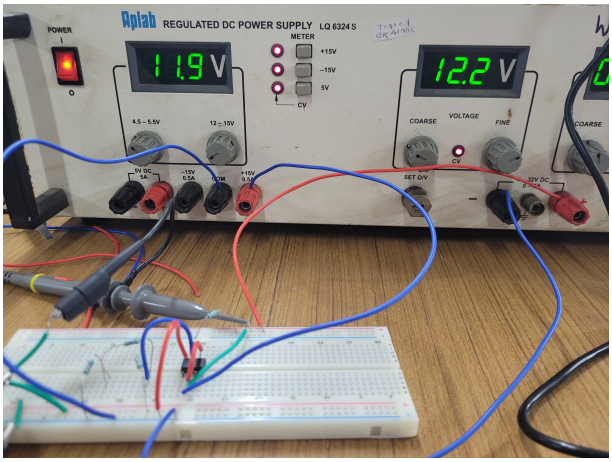

Figure 16: Breadboard design

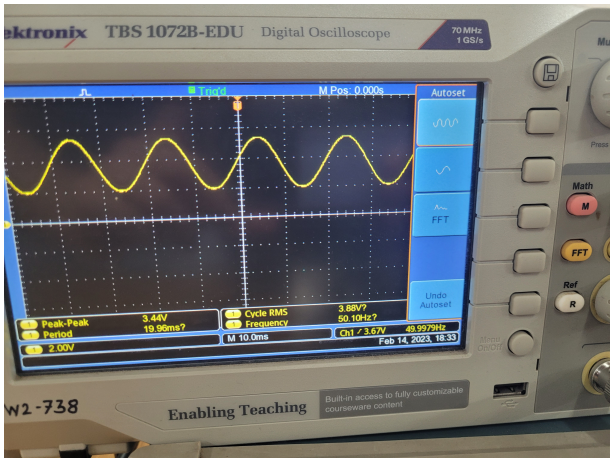

Figure 17: Results

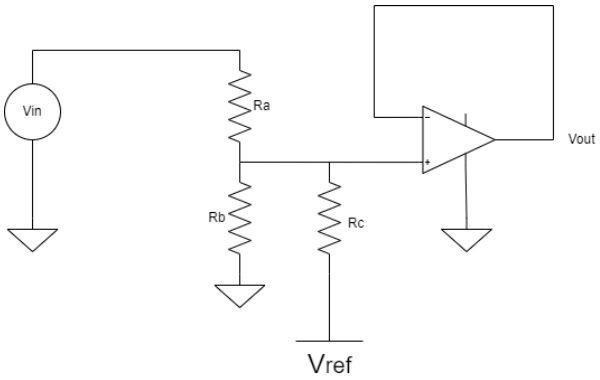

Figure 18: Circuit Diagram

Circuit was workin fine but circuit was clipping at lower voltage of 1.92V instead of expected 0V. Opamp used was LM741

4.3 Video of testing

Video of test of complete circuit and ADC measurement can be found here

5 Packaging

Packaging was done using acrylic board made using makercase.com, the files given for the printing are given in another folder of the zip file.

6 Bill of Material

Here, we have mentioned only the main components we used in our project. A detailed bill of material can be found in the BOM folder of the zip file.

Component	**Number**
Current Transformer AC1050	1
Voltage Transformer 44087	1
LAUNCHXL-F28027	1
LM317	1
Bluetooth Module HC-05	1

7 Conclusion & Future Work

We were able to almost complete the objective. After professor's review on our presentation, we were also able to calculate the frequency and the total energy, which was later shown on the Demo day. Due to some memory constraints we were not able to calculate the total harmonic distortion. Some shortcomings which can be improvised in future are

- Computing THD in the server side using Python
- A better choice of current sensor/transformer could've increased the accuracy of the measurements by a lot
- Inaccuracy in voltage transformer was also an issue
- Due to unavailability of current transformers we could only do single phase. However the PCB can be used for 3-Phase circuits as well

A Appendix A Code

```
// FILE: Voltage_mesurement.c
```

```
// following code samples ADC and measures RMS of that.

//
// Included Files
//
#include "DSP28x_Project.h"     // Device Headerfile and Examples Include File
#include<math.h>    // For sqrt library function
//#include "fft.h"
//#define N 32 //FFT size
////Buffer alignment
//#pragma DATA_SECTION(ipcb, "FFTipcb");
//#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
//long ipcbsrc[N];
//long ipcb[N+2];
///* Create an Instance of FFT module */
//RFFT32 fft=RFFT32_32P_DEFAULTS;
////
// Function Prototypes
//
__interrupt void adc_isr(void);
void Adc_Config(void);

//
// Globals
//
uint16_t LoopCount;
uint16_t ConversionCount;
uint16_t Voltage1[32];
uint16_t Voltage2[32];
uint32_t Vrms1;
float Vrms;
int i;
//
// Main
//
void main(void)
{
    //
    // WARNING: Always ensure you call memcpy before running any functions from
    // RAM InitSysCtrl includes a call to a RAM based function and without a
```

```
    // call to memcpy first, the processor will go "into the weeds"
    //
#ifdef _FLASH
    memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);
#endif

    //
    // Step 1. Initialize System Control:
    // PLL, WatchDog, enable Peripheral Clocks
    // This example function is found in the f2802x_SysCtrl.c file.
    //
    InitSysCtrl();

    //
    // Step 2. Initialize GPIO:
    // This example function is found in the f2802x_Gpio.c file and
    // illustrates how to set the GPIO to it's default state.
    //
    //InitGpio();  // Skipped for this example

    //
    // Step 3. Clear all interrupts and initialize PIE vector table:
    // Disable CPU interrupts
    //
    DINT;

    //
    // Initialize the PIE control registers to their default state.
    // The default state is all PIE interrupts disabled and flags
    // are cleared.
    // This function is found in the f2802x_PieCtrl.c file.
    //
    InitPieCtrl();

    //
    // Disable CPU interrupts and clear all CPU interrupt flags
    //
    IER = 0x0000;
    IFR = 0x0000;
```

```
//
// Initialize the PIE vector table with pointers to the shell Interrupt
// Service Routines (ISR).
// This will populate the entire table, even if the interrupt
// is not used in this example.  This is useful for debug purposes.
// The shell ISR routines are found in f2802x_DefaultIsr.c.
// This function is found in f2802x_PieVect.c.
//
InitPieVectTable();

//
// Interrupts that are used in this example are re-mapped to
// ISR functions found within this file.
//
EALLOW;             // This is needed to write to EALLOW protected register
PieVectTable.ADCINT1 = &adc_isr;
EDIS;      // This is needed to disable write to EALLOW protected registers

//
// Step 4. Initialize all the Device Peripherals
//
InitAdc();  // For this example, init the ADC
AdcOffsetSelfCal();

//
// Step 5. User specific code, enable interrupts:
// Enable ADCINT1 in PIE
//
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Enable INT 1.1 in the PIE
IER |= M_INT1;                                          // Enable CPU Interrupt
EINT;                                                   // Enable Global interr
ERTM;                                                   // Enable Global realti

LoopCount = 0;
ConversionCount = 0;

//
// Configure ADC
// Note: Channel ADCINA4  will be double sampled to workaround the ADC 1st
// sample issue for rev0 silicon errata
```

```
//
EALLOW;

//
// ADCINT1 trips after AdcResults latch
//
AdcRegs.ADCCTL1.bit.INTPULSEPOS        = 1;

AdcRegs.INTSEL1N2.bit.INT1E     = 1;    // Enabled ADCINT1
AdcRegs.INTSEL1N2.bit.INT1CONT  = 0;    // Disable ADCINT1 Continuous mode

//
// setup EOC2 to trigger ADCINT1 to fire
//
AdcRegs.INTSEL1N2.bit.INT1SEL        = 2;

//
// set SOC0 channel select to ADCINA4
//
AdcRegs.ADCSOC0CTL.bit.CHSEL         = 4;

//
// set SOC1 channel select to ADCINA4
//
AdcRegs.ADCSOC1CTL.bit.CHSEL         = 4;

//
// set SOC1 channel select to ADCINA2
//
AdcRegs.ADCSOC2CTL.bit.CHSEL         = 2;

//
// set SOC0 start trigger on EPWM1A, due to round-robin SOC0 converts first
// then SOC1
//
AdcRegs.ADCSOC0CTL.bit.TRIGSEL         = 5;

//
// set SOC1 start trigger on EPWM1A, due to round-robin SOC0 converts first
// then SOC1
```

```
//
AdcRegs.ADCSOC1CTL.bit.TRIGSEL         = 5;

//
// set SOC2 start trigger on EPWM1A, due to round-robin SOC0 converts first
// then SOC1, then SOC2
//
AdcRegs.ADCSOC2CTL.bit.TRIGSEL         = 5;

//
// set SOC0 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
//
AdcRegs.ADCSOC0CTL.bit.ACQPS          = 6;

//
// set SOC1 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
//
AdcRegs.ADCSOC1CTL.bit.ACQPS          = 6;

//
// set SOC2 S/H Window to 7 ADC Clock Cycles, (6 ACQPS plus 1)
//
AdcRegs.ADCSOC2CTL.bit.ACQPS          = 6;
EDIS;

//
// Assumes ePWM1 clock is already enabled in InitSysCtrl();
//
EPwm1Regs.ETSEL.bit.SOCAEN        = 1;                // Enable SOC on A group

//
// Select SOC from from CPMA on upcount
//
EPwm1Regs.ETSEL.bit.SOCASEL        = 4;

EPwm1Regs.ETPS.bit.SOCAPRD         = 1;                    // Generate pulse on 1
EPwm1Regs.CMPA.half.CMPA         = 0x0080;            // Set compare A value
EPwm1Regs.TBPRD                              = 0xFFFF;        // Set period fo
EPwm1Regs.TBCTL.bit.CTRMODE         = 0;                // count up and start
```

```
////    main()
////    {
////    .........
//    // Generate sample waveforms:
//    for(i=0; i < (N); i++)
//    {
//    ipcbsrc[i] =(long)Voltage1[i] //Q31
//    }
////    .........
//    RFFT32_brev(ipcbsrc, ipcb, N); /* Bit reverse */
//    fft.ipcbptr=ipcb; /* FFT computation buffer */
//    fft.init(&fft); /* Twiddle factor pointer init */
//    fft.calc(&fft); /* Compute the FFT */
//    fft.split(&fft); /* perform the split operation to get the N/2 + 1 spectrum*/
```

```
    //
    // Wait for ADC interrupt
    //
    for(;;)
    {
        LoopCount++;
    }

}

//
// adc_isr -
//
__interrupt void
adc_isr(void)
{
```

```
//
// discard ADCRESULT0 as part of the workaround to the 1st sample errata
// for rev0
//
Voltage1[ConversionCount] = AdcResult.ADCRESULT1;
Voltage2[ConversionCount] = AdcResult.ADCRESULT2;

//
// If 32 conversions have been logged, start over
//
if(ConversionCount == 31)
{
    ConversionCount = 0;
    Vrms1 = 0;
    \\compute RMS after sampling 32 values
    for(i =0;i<32;i++)
    {
        Vrms1 += Voltage1[i]*Voltage1[i];
    }
    Vrms = sqrt(Vrms1);
    Vrms = Vrms/32;
}
else
{
    ConversionCount++;
}




//
// Clear ADCINT1 flag reinitialize for next SOC
//
AdcRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;

//
// Acknowledge interrupt to PIE
//
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
```

```
    return;
}




//
// End of File
//
```