Activité Pratique:

DSP

Filtre IIR

Réalisé Par:

TERRASSI Ayoub

KADDOURI Nassira

Encadré Par:

Pr. M. Najoui

2022-2023

Contents

1 Introduction:

IIR filters or infinite Impulse Response are a type of digital filter used in Digital Signal Processing (DSP) applications. They are known as recursive digital filters due to their feedback mechanism, which uses previous output values along with current and previous input values to compute the filter's output.

IIR filters can be useful for designing many types of filters including the standard ones such as low-pass, high-pass, and band-stop filters. Compared to Finite Impulse Response (FIR filters), IIR filters can achieve a given filtering characteristic using less memory and calculations, not to mention that IIR filters are usually less power hungry, which makes them more efficient and suitable for embedded systems. Regarding the importance of the IIR filters in data signal processing it was preferable to study their behavior and try to implement them in DSP which is the goal this practical activity is aiming for. In this rapport we will resume our work in three major phases which are:

The implementation of IIR filter in DSP.

The Simulation and interpretation of the results.

The measurement of filtering time in different types of memory.

2 Filter implementation procedure in C6678:

2.1 Tools

To program the C6678 DSP we worked with CCS studio provided by Ti.

We used the following provided header files:

- **Input.h**: header file that contains the input samples, Input[800] array.
- **IIR.h**: header file containing the filter's coefficients.
- **IIR_output_ref.h**: header file containing the filter's coefficients.

2.2 IIR function in C

IIR mathematical formula:

$$y(n)=\sum_{i=0}^{M} b_i x(n-i)-\sum_{i=1}^{N} a_i y(n-i)$$

We converted the math formula of the IIR filter to the following C code:

```
float iir(float xn)
{
    int i;
    X_BUF[0]= xn;
    float y=X_BUF[0]*b_coeff[0];
    for (i=1;i<=7;i++)
        {
        y+=X_BUF[i]*b_coeff[i]-Y_BUF[i]*a_coeff[i];
        }
    Y_BUF[0]=y;
    for(i=7;i>=1;i--)
        {
        X_BUF[i]=X_BUF[i-1];
        Y_BUF[i]=Y_BUF[i-1];
        }
    return y ; }
```

3 Simulation and result interpretation:

3.1 Graphs

The input signal is represented by the array **Input**[**800**], that contains the samples.

The input signal contains the following frequencies: F1 = 50 Hz,
F2 = 300 Hz et F3 = 2KHz.

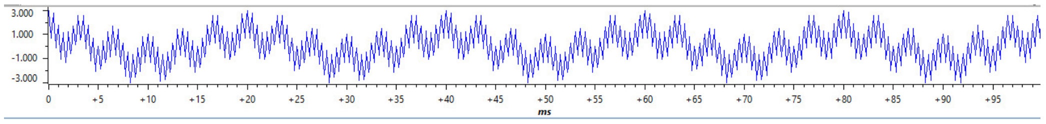


Figure 1: Input signal.

The result signal (output) is represented by the array **output**[**800**] that contains the samples.

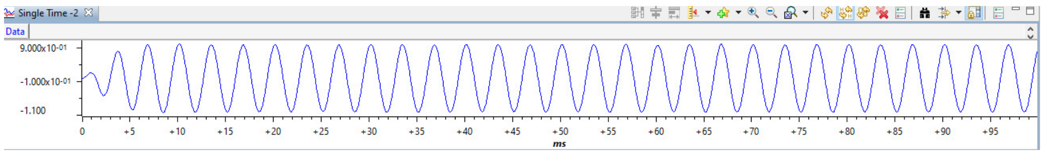


Figure 2: Output signal.

3.2 Interpretation

After analyzing the output signal period, we can see that the period is: $T = 3.375s$ which is approximatly equivalent to a frequency of **F = 300Hz**.

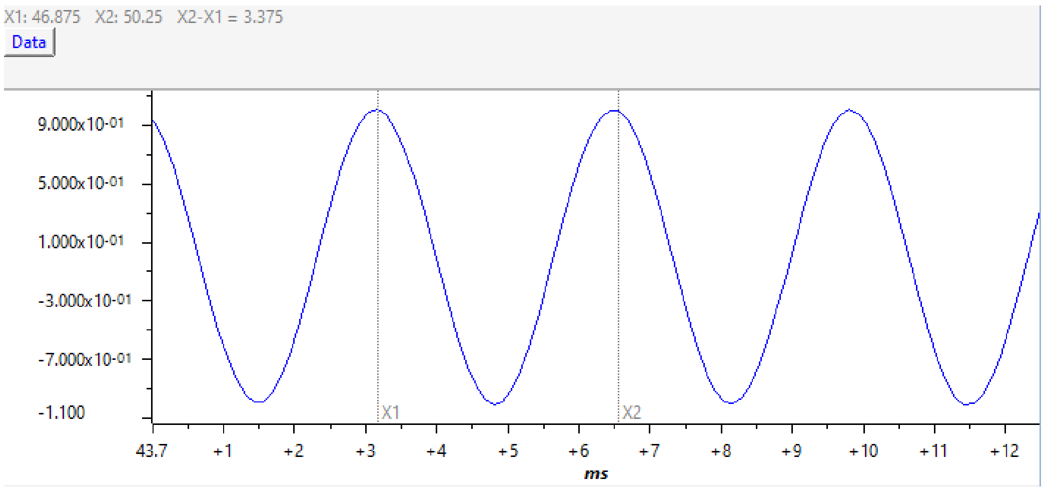


Figure 3: output signal period.

From the original three frequency components of the input signal only the F = 300Hz was conserved, we can say then that we are dealing with a: **Band pass filter**.

4 Filtering time measurement:

4.1 Tools

To calculate the time necessary to display the execution time we used the internal timer library:

- **csl_tsc.h**.

To map data we used:

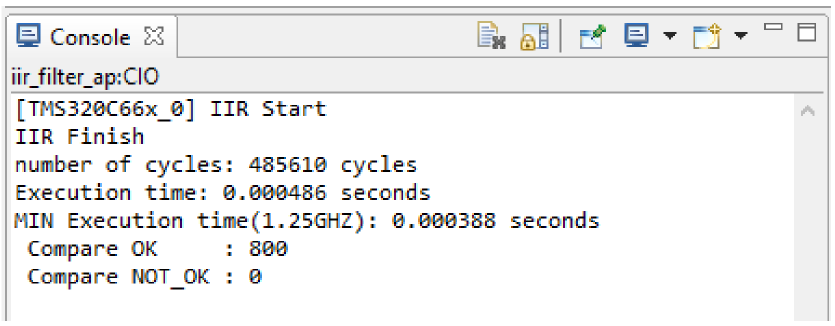
- the file **lnk.cmd** to map section in memory.
- **#pragma DATA_SECTION()** to assign our data to the sections.

4.2 Results

In these tests we used the memories: L2 and DDR3.
We compared them with and without the optimization -O3.

The results are the following:

L2 memory:



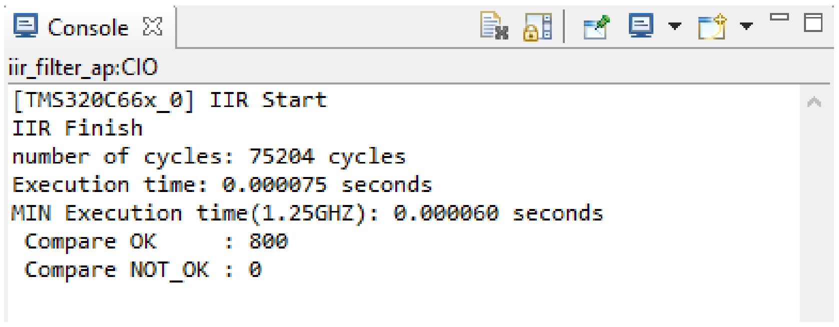
Console

iir_filter_ap:CIO

```
[TMS320C66x_0] IIR Start
IIR Finish
number of cycles: 485610 cycles
Execution time: 0.000486 seconds
MIN Execution time(1.25GHZ): 0.000388 seconds
 Compare OK     : 800
 Compare NOT_OK : 0
```

Figure 4: Memory L2 without optimization.

the execution time with data in L2 and without optimization is $t = 486\mu s$.
the number of cycles is: 485610.

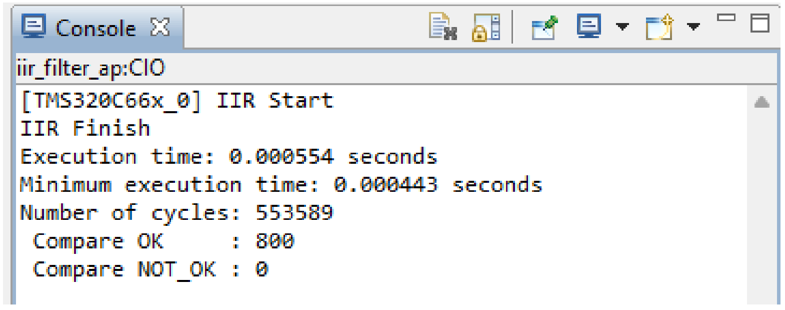


```
[TMS320C66x_0] IIR Start
IIR Finish
number of cycles: 75204 cycles
Execution time: 0.000075 seconds
MIN Execution time(1.25GHZ): 0.000060 seconds
 Compare OK     : 800
 Compare NOT_OK : 0
```

Figure 5: Memory L2 with optimization.

the execution time with data in L2 and with optimization enabled is $t = 75\mu s$. the number of cycles is: 75204.

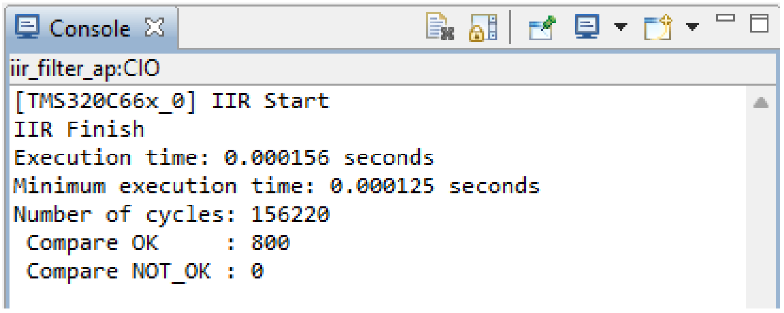
DDR3 memory:



```
[TMS320C66x_0] IIR Start
IIR Finish
Execution time: 0.000554 seconds
Minimum execution time: 0.000443 seconds
Number of cycles: 553589
 Compare OK     : 800
 Compare NOT_OK : 0
```

Figure 6: Memory DDR3 without optimization.

the execution time with data in DDR3 and without optimization is $t = 554\mu s$. the number of cycles is: 553589.



```
[TMS320C66x_0] IIR Start
IIR Finish
Execution time: 0.000156 seconds
Minimum execution time: 0.000125 seconds
Number of cycles: 156220
 Compare OK     : 800
 Compare NOT_OK : 0
```

Figure 7: Memory DDR3 with optimization.

the execution time with data in DDR3 and without optimization is $t = 156\mu s$. the number of cycles is: 156220.

Interpretation

We can see clearly the following differences:

- The execution time with optimization enabled differs greatly from the normal execution, the execution time becomes smaller.
- The DDR3 memory is slower than L2 which makes sense because the L2 cache memory is closer to the core, and it is a static memory on the contrary the memory DDR3 is a dynamic one.

5 Conclusion:

In this practical activity we came to the conclusion that IIR filters can be a good choice when it comes to embedded systems, real time applications and industrial internet of things ”IOT”.

If we have the stable and performing FIR filter that is efficient for frequency separation in one hand and the IIR filter that is less demanding on energy, very fast and less demanding in terms of memory in the another hand. the energy efficiency makes the IIR filter a good option when it comes to embedded applications, we can mention also the speed performance in filtring, since it went from transitory regime to steady state within almost an instant, and it took a reduced number of cycles (basically reduced time) to finish the filtering of the 800 samples right after passing them through the 7 layers fastly, this performance in terms of speed makes IIR good for real time applications, high speed telecommunication, and some biomedical applications too.

Annex:

main.c :

```
#include <stdio.h>
#include <math.h>
#include "IIR.h"
#include "Input.h"
#include "IIR_Output_Ref.h"
#include "csl_tsc.h"
#include<c6x.h>

#pragma DATA_SECTION (Input,".mysect");
#pragma DATA_SECTION (output,".mysect");

#define CYCLE_PERIOD 1e-9
#define CYCLE_PERIOD_min 8e-10
#define N 800

float output[800];
float iir(float xn);
float compare;
unsigned int compare_NOT_OK;
unsigned int compare_OK;

void main (void)
{
    unsigned long long int start,end,nb_cycle,overhead;
    double execution_time;
    double min_execution_time;
    int i;
    compare_OK = 0;
    compare_NOT_OK = 0;
    _CSL_tscEnable();

    start=_CSL_tscRead();

    end=_CSL_tscRead();
    overhead=end-start;

    printf("IIR Start\n");

    start =_CSL_tscRead ();
    for(i=0; i < 800; i++){
        output[i] = iir(Input[i]);

    }
    end=_CSL_tscRead();
    nb_cycle =end-start-overhead;

    printf("IIR Finish\n");

    execution_time = (double)nb_cycle * CYCLE_PERIOD;
    min_execution_time = (double)nb_cycle * CYCLE_PERIOD_min;
    printf("Execution time: %lf seconds\n", execution_time);
    printf("Minimum execution time: %lf seconds\n", min_execution_time);
    printf("Number of cycles: %lld \n", nb_cycle);

    /** Comparaison *********************************/

    for (i=0;i<800;i++)
    {
            compare = abs(output[i] - IIR_Output_Ref[i]);
            if(compare <= 0.01)
            {
                compare_OK ++ ;
            }
            else
            {
                compare_NOT_OK ++;
            }
    }
    printf(" Compare OK     : %d \n",compare_OK);
    printf(" Compare NOT_OK : %d \n",compare_NOT_OK);
}
```

```
/** IIR filter function ************************/

float iir(float xn)
{
    int i;

    X_BUF[0]= xn;
    float y=X_BUF[0]*b_coeff[0];
    for (i=1;i<=7;i++)
        {
        y+=X_BUF[i]*b_coeff[i]-Y_BUF[i]*a_coeff[i];
        }
    Y_BUF[0]=y;
    for(i=7;i>=1;i--)
        {
        X_BUF[i]=X_BUF[i-1];
        Y_BUF[i]=Y_BUF[i-1];
        }
    return y ;
}
```