Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Disciplina Projeto em Engenharia Elétrica

Projeto e Implementação de Filtros Bi-Dimensionais em MatLab e C para Aritmética de Ponto Fixo

Genildo de Moura Vasconcelos
genildo@dee.ufcg.edu.br

Orientador:
Angelo Perkusich
perkusic@dee.ufcg.edu.br

Campina Grande, Fevereiro de 2006

Biblioteca Setorial do CDSA. Fevereiro de 2021.

Sumé - PB

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Disciplina Projeto em Engenharia Elétrica

Projeto e Implementação de Filtros Bi-Dimensionais em MatLab e C para Aritmética de Ponto Fixo

Genildo de Moura Vasconcelos
(Aluno)

Angelo Perkusich
(Orientador)

Sumário

1. Introdução

Os filtros digitais aparecem como uma das mais importantes funções do processamento digital de sinais, sendo usando em diversas aplicações, incluindo o processamento de voz, dados, imagem, vídeo, sonar, radar, e eletrônicos. Onde a sua implementação em sistemas embarcados demanda cada vez mais o processamento em tempo real de grande quantidade de informações. Porém, algoritmos de processamento digital de sinais (como por exemplo, imagens) exigem grande capacidade computacional.

Os sistemas embarcados baseados na aritmética de ponto fixo são geralmente mais baratos e mais rápidos do que arquiteturas baseadas na aritmética de ponto flutuante, porque eles usam menos silício e têm menos pinos externos [1]. Estas limitações estimulam o estudo de arquiteturas que trabalham com aritmética de ponto fixo como o DSP (Digital Signal Processor).

O projeto e implementação de algoritmos para a plataforma de Processadores Digitais de Sinais baseados em aritmética de ponto fixo, pode ser baseado nos seguintes passos:

1) Formulação matemática dos algoritmos a serem implementados;
2) Implementação e simulação dos algoritmos em linguagens como o MatLab [3] e C/C++ usando aritmética de ponto flutuante;
3) Implementação e simulação dos algoritmos em linguagens como o MatLab e C/C++ usando aritmética de ponto fixo;
4) Código em C/C++ usando aritmética de ponto fixo para a plataforma final (DSP);
5) Utilização de métricas de desempenho.

Esse projeto está dividido em três etapas, a citar:

1) Projeto e implementação de filtros bi-dimensionais em MatLab [3] e C para a aritmética de ponto flutuante;
2) Projeto e implementação de filtros bi-dimensionais em MatLab e C para aritmética de ponto fixo;
3) Utilização de métricas de desempenho (objetivas e subjetivas) referentes à qualidade e tempo de processamento dos resultados obtidos.

Na presente atividade, trataremos da segunda etapa do projeto. Iremos implementar os filtros da média, mediana e Wiener adaptativo em ponto fixo.

2. Objetivos

2.1. Objetivos Gerais

Agregar conhecimentos sobre as arquiteturas de ponto flutuante e ponto fixo.

2.2. Objetivos Específicos

Implementação dos filtros da Média, Mediana e Wiener adaptativo em ponto fixo utilizando as linguagens de programação MatLab e C e então realizar uma análise de desempenho com relação aos tempos de execução dos filtros.

3. Processadores de ponto fixo

Os processadores de ponto fixo tal como os DSP's apresentam como principal característica em relação aos processadores de ponto flutuante o formato usado para armazenar e manipular números. A representação numérica dos DSP's de ponto fixo é normalmente de no mínimo 16 bits, possibilitando a representação de inteiros com e sem sinal, bem como, frações com e sem sinal. As operações aritméticas utilizadas com bastante freqüência no processamento digital de sinais tal como a operação MAC *"Multiplica e Acumula"* geralmente estão otimizadas nos DSP's. O acesso à memória nos DSP's receberam significante atenção no desenvolvimento da arquitetura, onde barramentos extras foram inseridos permitindo ao processador uma maior eficiência na transferência dos dados.

3.1 O TMS320C55xx

Para este trabalho escolhemos como arquitetura alvo a família de DSPs TMS320C55xx de 192Mhz da Texas Instruments. O C55xx foi projetado para prover um baixo consumo, desempenho otimizado e uma alta densidade de código (menor número de instruções por ciclo do clock) [1]. Abaixo temos algumas características do C55x:

- Código fonte compatível com todos os dispositivos TMS320C54x;

- Duas unidades MAC de 17bits por 17bits que podem executar duas operações MAC em um mesmo ciclo;
- Uma unidade lógica e aritmética de 40bits que executa operações com alta precisão; e uma unidade lógica e aritmética adicional de 16bits para a execução de operações aritméticas mais simples em paralelo com a unidade lógica e aritmética principal;
- Quatro acumuladores de 40bits para armazenamento computacional dos resultados de forma a diminuir os acessos à memória;

3.2 A arquitetura do TMS320C55xx

A arquitetura do C55x consiste de quatro unidades de processamento: uma unidade de instrução de buffer (IU) que traz as instruções da memória para a CPU, uma unidade de fluxo de programa (PU) que controla a execução do programa, uma unidade de fluxo de endereços-dados (AU) que serve como um gerenciador de acesso aos dados para os barramentos de leitura e escrita, e uma unidade de computação de dados (DU) que controla a maioria das aplicações no C55xx.

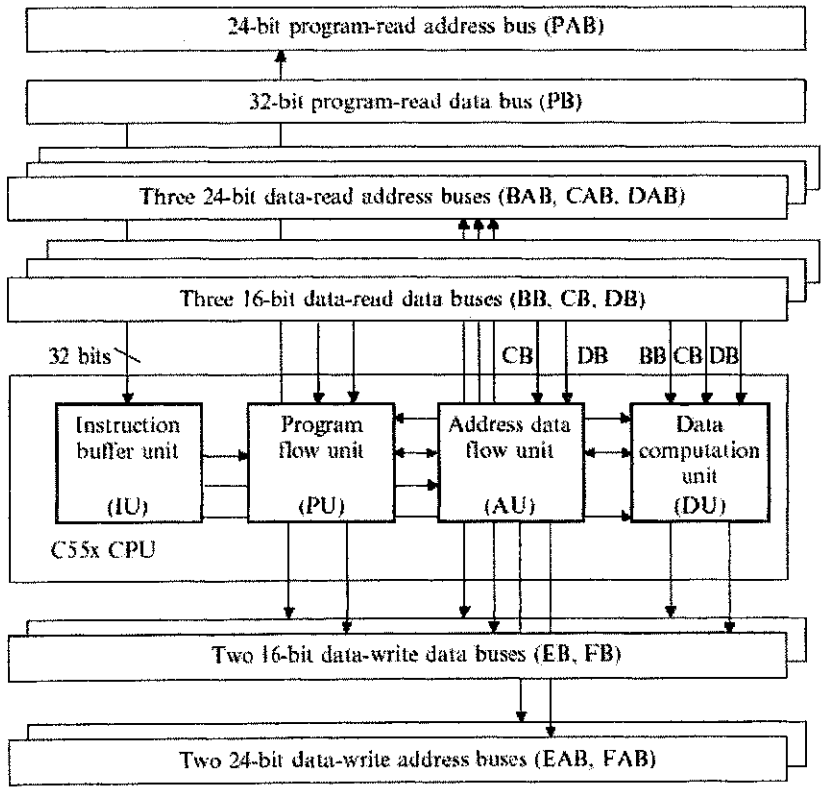

Figura 1 - Diagrama da CPU do TMS320C55x

3.3 O mapa de memória do TMS320C55xx

O C55xx possui uma capacidade total de endereçamento de 16Mbytes disponível para dados e programa. O espaço de programa é usado para as instruções e o espaço de dados é usado para o armazenamento de dados e o mapeamento dos registradores na memória da CPU. O espaço de I/O é separado do espaço de programas e dados, e é usado para comunicação duplex com os periféricos. O mapa de memória dos 16Mbytes é mostrado na figura 2. O mapa de memória é dividido em 128 páginas de dados (0-127). Cada página tem 64Kbytes, o bloco de memória do endereço 0 até 0x5F na página 0 é reservado para os registradores de mapeamento da memória (MMRs).

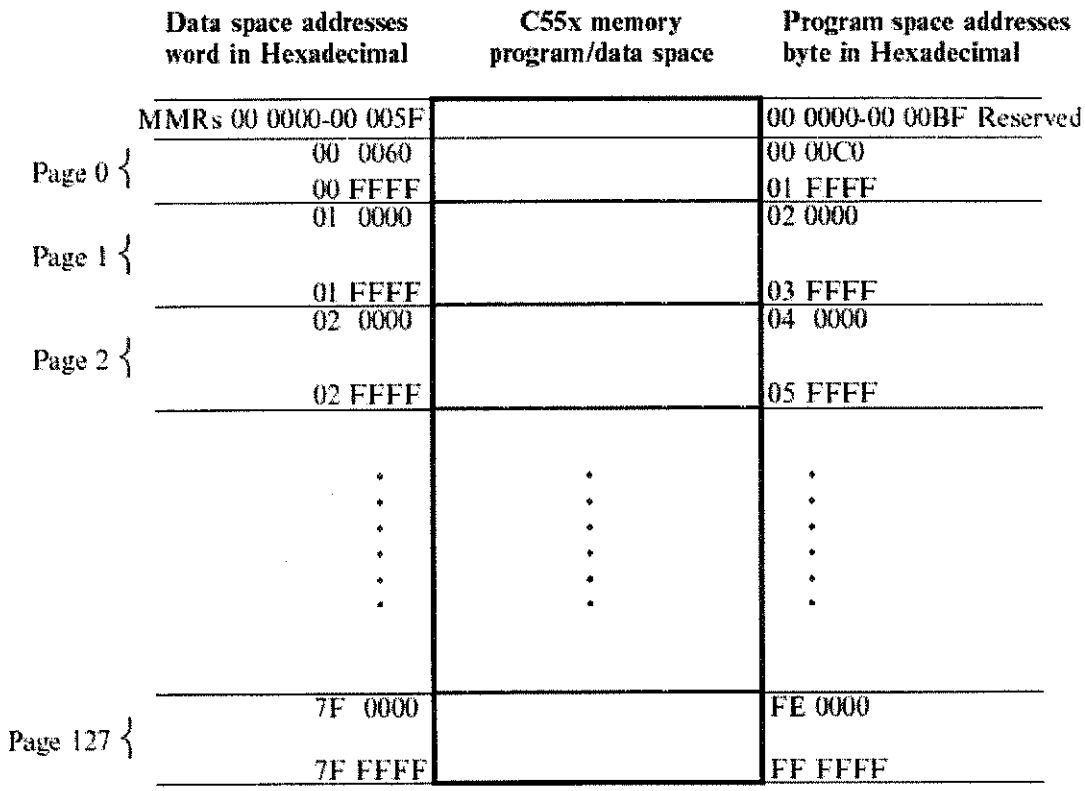

Figura 2 - Mapa de memória do TMS320C55xx

3.4 Ferramenta para desenvolvimento de software

A implementação dos algoritmos para o TMS320C55xx geralmente consiste no desenvolvimento em MatLab para verificação e análise de sua funcionalidade. Após este estágio utiliza-se a ferramenta Code Composer Studio (CCS), que é o software para programação e simulação dos DSP's da Texas Instruments.

4. Representação de Números em Ponto Fixo

Existem várias formas para operar aritmeticamente com números binários. Uma delas é a aritmética de ponto fixo. Na aritmética de ponto fixo, o ponto tem uma localização fixa no registrador. A representação de um número em ponto fixo se dá da seguinte maneira:

$$m2^{-e} \tag{1}$$

Onde,

m : mantissa

e : expoente

Algumas regras devem ser respeitadas na aritmética de ponto fixo, tais como:

- Mudança de expoente

$$m2^{-p} = m2^{r-p} \times m2^{-r} \tag{2}$$

- Adição e Subtração

$$n2^{-r} + m2^{-r} = (n+m)2^{-r} \tag{3}$$

Obs: Se os expoentes são diferentes a conversão para mesmo expoente deve ser feita antes da adição ou subtração.

- Multiplicação

$$ab = n2^{-p} \times m2^{-q} = (nm)2^{-(p+q)} \tag{4}$$

- Divisão

$$\frac{a}{b} = \frac{n2^{-p}}{m2^{-q}} = (\frac{n}{m})2^{(q-p)} = (\frac{n}{m})2^{r+q-p}2^{-r} \tag{5}$$

Obs: Para não perder precisão devemos efetuar a multiplicação por 2^{r+q-p} antes da divisão por m.

4.1 Representação dos números no formato Q

A representação de números no formato Q tem por finalidade aumentar a precisão dos cálculos em ponto fixo.

Os procedimentos para determinação da representação Q serão apresentados através de um exemplo:

Exemplo:

Partindo da necessidade de representar o número 8.65 (decimal) em uma variável int (16bits), a representação no formato Q mais adequada consiste naquela que apresentará uma maior precisão sem causar overflow.

Para uma variáveis de 16bits, a representação complemento de dois possui 15 bits de dados e 1 de sinal, onde os valores estão compreendidos entre -2^{15} até $2^{15}-1$, ou seja, o maior valor inteiro positivo será 32767.

Logo, para um dado número x, temos:

$$N_{max} = 2^q \cdot x \tag{6}$$

Onde,

N_{max} = máximo valor que a variável pode possuir;

q = a representação no formato Q correspondente;

x = o número que deseja-se representar no formato Q;

Aplicando o logaritmo na base 2 na equação (6):

$$\log_2(N_{max}) = q + \log_2(x) \tag{7}$$

$$q = \log_2(N_{max}) - \log_2(x) \tag{8}$$

Logo, para um $x = 8.65$, este número possuirá a seguinte representação no formato Q:

$$q = \log_2(32767) - \log_2(8.65) = 11.88 \tag{9}$$

Escolhendo um valor para Q acima de 11.88 estaríamos ultrapassando a máxima representação de uma variável *int* que seria de -32768 à 32767, logo escolhemos a representação Q igual a 11.

5. Implementação dos Filtros Digitais da Média, Mediana e Wiener Adaptativo em ponto fixo

Esta seção apresentará a implementação dos filtros da média, mediana e Wiener adaptativo em aritmética de ponto fixo, inicialmente no MatLab, para verificação e análise do algoritmo, e determinação adequada da representação das variáveis no formato Q com o intuito de evitar overflow e aumentar a exatidão. A partir dos

resultados obtidos na análise feita no MatLab inicia-se a programação em C utilizando a ferramenta Code Composer Studio (CCS).

5.1. Filtro da Média

Partindo do resultado, na primeira etapa do trabalho, onde foi implementado o filtro da média em aritmética de ponto flutuante, será feito o porte para ponto fixo do algoritmo obtido.

5.1.1 Implementação no MatLab

Abaixo é apresentada a rotina ***meanfilterFixed*** que implementa a função do filtro da média em ponto fixo no MatLab. Esta função tem como parâmetros de entrada ***g*** que corresponde à matriz componente da imagem (RGB) e ***window*** que representa a janela que será utilizada no filtro.

```
**********************************************************************

function mean=meanfilterFixed(g,window)

N = size(g,1);
M = size(g,2);

invsqwindow=1/(window*window);
mean=uint16(zeros(N,M));

lowerWindow=floor((window-1)/2);
upperWindow=ceil((window-1)/2);

%cast para inteiro sem sinal com 16 bits, variável que corresponde no CCS a unsigned char
g=uint16(g);

%invsqwindow em ponto fixo no formato Q19
invsqwindow=uint16(invsqwindow*2^19);
meanAux = 0;
meanAux = uint32(meanAux);

for n=1:N
  for m=1:M
    for i=n-lowerWindow:n+upperWindow
      for j=m-lowerWindow:m+upperWindow
        if(~(i<1 || j<1 || i>N || j>M))
          % Em Q0
          mean(n,m)=mean(n,m)+g(i,j);
        end
      end
    end
    %Q0*Q19 = Q19
    meanAux=uint32(mean(n,m))*uint32(invsqwindow);

    %De Q19 para Q0
    mean(n,m)=uint16(meanAux/2^19);
  end
end
**********************************************************************
```

A primeira alteração que pode-se notar em relação à implementação em ponto flutuante, é que a matriz componente ***g*** deve ser do tipo inteiro sem sinal *(unsigned int)*, o tipo *unsigned* deve-se ao fato da imagem em bmp possuir apenas valores positivos (0 – 255) e 16 bits devido à arquitetura do C55xx, onde 16bits é o tipo de variável de menor tamanho. Esta conversão para *(unsigned int)* é feita através da função I = UINT16(X) que converte o elemento do array X para números inteiros de 16bits sem sinal.

Como o C55xx possui registradores de 32 bits, em alguns casos, serão utilizados valores auxiliares de 32 bits para salvar os resultados das operações diminuindo a perda de precisão.

Será considerado que a matriz componente **g** está no formato Q0, ou seja, sua representação no formato Q é igual ao seu valor real. Não foi realizada uma normalização na matriz componente, visto que uma boa exatidão será alcançada devido a faixa de valores de ***g*** estarem limitadas entre 0 – 255.

Em relação a variável ***mean***, para um valor máximo de ***window*** igual a 5, seu valor máximo será 255*(5*5), como está ilustrado abaixo:

255	255	255	255	255	255	255
255	255	255	255	255	255	255
255	255	255	255	255	255	255
255	255	255	255	255	255	255
255	255	255	255	255	255	255
255	255	255	255	255	255	255
255	255	255	255	255	255	255

Figura 3 - Valor máximo da variável mean 255*(5x5)

Como o resultado de ***mean*** será a média aritmética de todos os elementos dentro da janela 5x5, esta operação corresponderá inicialmente da soma de todos estes elementos, que estão no formato Q0, logo o somatório também estará em Q0.

Em ponto flutuante, a média é calculada dividindo o somatório dos elementos por (5*5) *(número total de elementos na janela)*. Já em ponto fixo, está operação é feita encontrando o valor de 1/(5*5) ***(invsqwindow)*** e transformando-se este valor para ponto fixo através da representação no formato Q e multiplicando posteriormente por ***mean***.

Deve-se notar que representar o valor de ***invsqwindow*** com 16 bits implicará, no pior caso, que a multiplicação de ***mean*** por ***invsqwindow*** resultará em uma variável

auxiliar de 32 bits. Sendo assim a representação de ***invsqwindow*** no formato Q deve ser determinada de tal sorte que o resultado da multiplicação por ***mean*** não ultrapasse os 32bits da variável auxiliar ***(meanAux)***. Além disso, como ***invsqwindow*** será armazenado em uma variável de 16bits sua representação em formato Q também deve ser adequada para este tipo de variável.

O maior valor que ***invsqwindow*** pode ter ocorre quando ***window*** é igual a 3 e não quando **window** igual a 5 como ocorre em ***mean***, onde 3 será o menor valor que será tratado para ***window*** no algoritmo de ponto fixo. Logo o valor máximo de ***invsqwindow*** será 1/(3*3), a partir destas considerações podemos determinar qual o formato Q mais adequado:

Para uma variável de 16bits inteira sem sinal, o máximo valor que pode ser representado corresponde a:

$$N_{max} = 2^{16} - 1 = 65535 \tag{10}$$

A determinação do Q adequado se dá a partir da equação 8:

$$q = \log_2(65535) - \log_2\left(\frac{1}{9}\right) = 19 \tag{11}$$

Desta forma,

$$\boldsymbol{invsqwindow}_{max} = 2^{19} * \frac{1}{9} = 58254 \tag{12}$$

Será verificado ainda, se ao multiplicar-se ***invsqwindow*** por ***mean***, o resultado não ultrapassará o limite da variável auxiliar ***(meanAux)*** de 32bits. Para isto, deve-se observar que partindo do valor máximo de ***mean*** "255*(5x5) = 6375", pode-se montar a seguinte equação para o valor máximo de ***invsqwindow***:

$$\boldsymbol{mean}_{max}^{*}\ \boldsymbol{invsqwindow}_{max} \leq \left(2^{32} - 1\right) \tag{13}$$

$$6375 * 58254 = 371369250 \leq \left(2^{32} - 1\right) = 4294967295 \tag{14}$$

Desta forma, observa-se que a representação em formato Q não causará overflow, já que os extremos de ***invsqwindow*** e ***mean*** foram testados.

Após esta multiplicação, meanAux estará também no formato Q19, para retornar o valor de ***mean*** para Q0, deve-se dividir o valor de ***meanAux*** por 2^{19} que poderá ser armazenado em ***mean*** de 16bits, sem grande perda de precisão.

Desta forma, temos o algoritmo em ponto fixo, que poderá agora ser portado para o TMS320C55xx utilizando a linguagem C no Code Composer Studio.

5.1.2 Implementação no Code Composer Studio

Partindo do algoritmo que foi desenvolvido no MatLab, será feito agora o porte do filtro da média para C, no ambiente Code Composer Studio, que permitirá ter o código simulado em uma arquitetura de ponto fixo, neste caso, o DSP TMS320C55xx.

Abaixo temos o código da função do filtro da média, implementada em C.

```
*****************************************************************************************
void meanfilterFixed(unsigned char *inpArray,int window, int rows, int columns)
{
        int i,j,n,m,tmp,tmp2,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;
        unsigned short mean[144];

        //Tabela com os valores de 1/window*window em Q19
        unsigned short invsqwindowTable[] = {58254, 20971};

        unsigned short invsqwindow;

        unsigned long meanAux;

        //Seleção do 1/window*window de acordo com a entrada
        invsqwindow = invsqwindowTable[(window-3)/2];

        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        for (n=0;n<N;n++){
                for (m=0; m<M; m++){
                        tmp=n*columns+m;
                        mean[tmp]=0;
                        for(i=n-lowerWindow; i<=n+upperWindow;i++){
                                for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                        if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                                tmp2=i*columns + j;
                                                mean[tmp]=mean[tmp]+inpArray[tmp2];
                                        }
                                }
                        }

                        meanAux=(unsigned long)mean[tmp] * (unsigned long)invsqwindow;

                        mean[tmp]=(meanAux) >> 19;

                }
        }

        //Cópia dos dados de mean para inpArray
        for(i = 0; i < rows; i++){
                for(j=0; j<columns; j++){
                        tmp2=i*columns+j;
                        inpArray[tmp2]=mean[tmp2];
```

```
            }
        }
}
*****************************************************************************
```

A função meanFixed receberá como parâmetros o array ***inpArray*** que é do tipo unsigned char, nesta arquitetura o unsigned char é uma variável tipo inteiro sem sinal de 16 bits. Esta variável apontará para as componentes da imagem RGB, e como cada componente tem seu valor limitado de 0 – 255 o tamanho da variável unsigned char é mais do que suficiente para armazenar todos estes valores, o parâmetro ***inpArray*** corresponde ao parâmetro ***g*** no código em MatLab.

O parâmetro ***window***, assim como o código em MatLab, corresponde ao tamanho da janela, e os parâmetros ***rows*** e ***columns*** que representam respectivamente o número de linhas e de colunas da componente imagem a ser processada.

Aqui também deve-se trabalhar com a representação no formato Q, sendo que já foi calculado que o formato Q19 é a máxima representação que podemos utilizar na variável ***invsqwindow*** para que não ocorra overflow. Utiliza-se uma tabela com os valores de ***invsqwindow***, em Q19 para os casos de ***window*** igual a 3 e 5, com a finalidade de evitar cálculos em ponto flutuante. Estes valores são encontrados da seguinte forma:

Para ***window*** igual a 3, ***invsqwindow*** em ponto flutuante será:

$$\boldsymbol{Invsqwindow}_{floating} = \frac{1}{3*3} = 0.11111111 \tag{15}$$

Em ponto fixo teremos:

$$\boldsymbol{Invsqwindow}_{fixed} = \left(2^{19}\right) * \boldsymbol{Invsqwindow}_{floating} = 58254 \tag{16}$$

Para ***window*** igual a 5, ***invsqwindow*** em ponto flutuante será:

$$\boldsymbol{Invsqwindow}_{floating} = \frac{1}{5*5} = 0.04 \tag{17}$$

Em ponto fixo teremos:

$$\boldsymbol{Invsqwindow}_{fixed} = \left(2^{19}\right) * \boldsymbol{Invsqwindow}_{floating} = 20971 \tag{18}$$

Para armazenar o valor da multiplicação de ***mean*** *(Q0)* por ***invsqwindow*** *(Q19)* utiliza-se a variável auxiliar ***meanAux*** que é do tipo *unsigned long* de 32 bits, seu valor estará em Q19, para representar ***meanAux*** em Q0, desloca-se para a direita 19 bits *(operador >>)*, operação equivalente a dividir seu valor por 2^19, este valor será armazenado em ***mean***.

Após a varredura de todas as posições tem-se então concluída a filtragem da matriz componente, restando por último atualizar os valores de ****inpArray*** com os valores filtrados da variável ***mean***.

5.2. Filtro da Mediana

O filtro da mediana possui uma característica particular, que consiste em não necessitar de cálculos aritméticos para a implementação do seu algoritmo. Isto decorre do fato de que para encontrarmos a mediana de um array de dados, basta ordenarmos o array em ordem crescente ou decrescente e selecionar o elemento central do array. Desta forma observa-se que a implementação em ponto fixo não demanda a representação das variáveis no formato Q. Seguindo a mesma metodologia utilizada para implementação do filtro da média, inicia-se com o ambiente MatLab e então segue-se para o desenvolvimento no Code Composer Studio.

5.2.1 Implementação no MatLab

Abaixo é apresentada a rotina **medianfilterFixed** que implementa a função de filtro da mediana no MatLab. Esta função tem como parâmetros de entrada ***g*** que corresponde a matriz componente da imagem (RGB) e ***window*** que representa a janela que será utilizada no filtro, assim como no filtro da média.

Como a implementação não demanda nenhuma conversão para ponto fixo, sua implementação no MatLab em ponto fixo apresenta-se da mesma forma que em ponto flutuante.

```
**************************************************************************************
function median=medianfilter(g,window)

N = size(g,1);
M = size(g,2);

invsqwindow=1/(window*window);
median=uint16(zeros(N,M));

tempArray=uint16(zeros(window*window,1));
tmpArrayIndex=1;
```

```
g = uint16(g);

lowerWindow=floor((window-1)/2)
upperWindow=ceil((window-1)/2)

for n=1:N
  for m=1:M
    tmpArrayIndex=1;
    for i=n-lowerWindow:n+upperWindow
      for j=m-lowerWindow:m+upperWindow
        if(~(i<1 || j<1 || i>N || j>M))
          tempArray(tmpArrayIndex)=g(i,j);
          tmpArrayIndex=tmpArrayIndex+1;
        end
      end
    end

    %ordenar tempArray com o uso de dois indexadores (i e j)
    i=1;
    j=2;
    while i<=tmpArrayIndex-2
        while j<=tmpArrayIndex-1
                if(tempArray(i)>tempArray(j))
                        tmp=tempArray(i);
                        tempArray(i)=tempArray(j);
                        tempArray(j)=tmp;
                end
        j=j+1;
      end
      i=i+1;
      j=i+1;
    end
    median(n,m)=tempArray(ceil((tmpArrayIndex-1)/2));
  end
end
```

****

5.2.2. Implementação no Code Composer Studio

A implementação no Code Composer Studio como não demanda de transformações para o formato Q, o porte consistiu na alteração dos tipos das variáveis para os tipos permitidos pela arquitetura do DSP, onde utilizou-se *unsigned char*.

****

```
void filtroMediana(unsigned char *inpArray, int window, int rows, int columns){

        //Definição de variáveis
        int i,j,n,m,tmp,tmp2,tmpArrayIndex,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;

        unsigned char median[144], tempArray[144];
```

```
        //Definição dos limites da janela
        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        //Laços que varrem as linhas e colunas
        for (n=0;n<N;n++){
          for (m=0; m<M; m++){
                        //Laços para o armazenamento de valores no tempArray em uma vizinhança
                        //local window x window
                        tmp=n*columns+m;
                        tmpArrayIndex=0;
            for(i=n-lowerWindow; i<=n+upperWindow;i++){
                    for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                    //Teste para verificar se o pixel excede as bordas
                                    if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                            tmp2=i*columns + j;
                                            tempArray[tmpArrayIndex]=inpArray[tmp2];
                                            tmpArrayIndex++;
                                    }
                    }
            }

                        //Ordenameno de tempArray com o uso de dois indexadores (i e j)
                        i=0;
                        j=1;
                        while(i<=tmpArrayIndex-2){
                                while(j<=tmpArrayIndex-1){
                                        if(tempArray[i]>tempArray[j]){
                                                tmp2=tempArray[i];
                                                tempArray[i]=tempArray[j];
                                                tempArray[j]=tmp2;
                                        }
                                        j++;
                                }
                                i++;
                                j=i+1;
                        }
                }
        }

        //Cópia dos dados de mean para inpArray
        for(i = 0; i < rows; i++){
                for(j=0; j<columns; j++){
                tmp2=i*columns+j;
                inpArray[tmp2]=(unsigned char)(median[tmp2]);
                }
        }

}
```

****

5.3. Filtro de Wiener Adaptativo

O filtro de Wiener, como será apresentado, possui um número maior de operações, o que irá demandar um maior cuidado com os valores limites evitando-se o aparecimento de overflow. Como em toda representação em ponto fixo existe o compromisso entre precisão e tamanho da palavra que será usada, utilizando-se palavras

muito pequenas para representar o valor em ponto fixo, tem-se perda de precisão, entretanto, evita-se o desperdício de memória e o acréscimo de processamento que será necessário para o gerenciamento das variáveis de tipo maiores.

5.3.1. Implementação no MatLab

A seguir temos a implementação do filtro de Wiener adaptativo no MatLab, assim como foi desenvolvido nos filtros passados utilizou-se os tipos de variáveis compatíveis com as do CCS.

```
**************************************************************************************
function output=wienerfilterFixed(g,window)

N = size(g,1);
M = size(g,2);

invsqwindow=1/(window*window);
mean=uint16(zeros(N,M));
var=uint16(zeros(N,M));
output=int32(zeros(N,M));
noise=0;
noise=int32(noise);
noise1 = 0;

%g em Q0
g = uint16(g);

%Definição dos limites da janela
lowerWindow=floor((window-1)/2);
upperWindow=ceil((window-1)/2);

% invsqwindow em Q14
invsqwindow = uint16(invsqwindow*2^14);
meanAux = 0;
meanAux = uint32(meanAux);

varAux = 0;
varAux = uint32(varAux);

gAux = 0;
gAux = int32(gAux);

%Laços que varrem as linhas e colunas
for n=1:N
  for m=1:M

    mean(n,m) = 0;
    var(n,m) = 0;
    varAux = 0;
    meanAux = 0;

    %Laços para o cálculo da média e variância locais
    %em uma vizinhança local window x window
    for i=n-lowerWindow:n+upperWindow
      for j=m-lowerWindow:m+upperWindow
        %Teste para verificar se o pixel excede as bordas
```

```
            if(~(i<1 || j<1 || i>N || j>M))
              %Q0+Q0 = Q0
              mean(n,m)=mean(n,m)+g(i,j);
              %Q0+Q0*Q0 = Q0
              varAux = varAux + uint32(g(i,j)*g(i,j));
            end
          end
        end
        %Q0*Q14 = Q14
        meanAux= uint32(mean(n,m))*uint32(invsqwindow);

        %De Q14 para Q0
        mean(n,m) = uint16(meanAux/2^14);

        %Q0*Q0 = Q0
        meanAux = mean(n,m)*mean(n,m);

        %Q0*Q14 = Q14
        varAux = uint32(varAux)*uint32(invsqwindow);

        %De Q14 para Q0
        var(n,m) = uint16(varAux/2^14);

        %Q0 - Q0 = Q0
        varAux=uint32(var(n,m))-uint32(meanAux);

        %De Q14 para Q0
        var(n,m) = uint16(varAux);


        %Estimativa do ruído
        noise=noise+int32(var(n,m));
      end
    end

    noise1= double(noise)/(N*M);
    noise = int32(noise1);

    %Laço para o cálculo da expressão:
    %g(n,m)= mean(n,m) + max(0, var(n,m - noise))/max(var, noise) * [g(n,m) - mean(n,m)];
    for n=1:N
      for m=1:M
        output(n,m)=g(n,m)-mean(n,m);

        gAux=int32(var(n,m))-int32(noise);

        if(gAux < 0)
          gAux = 0;
        end
        %Caso ocorra divisao por 0, a saída será igual à média
        if(var(n,m)<=0)
          output(n,m)=mean(n,m);
        else
          %De Q0 para Q15
          output(n,m)=output(n,m)*2^15;
          %(Q15/Q0)*Q0 = Q15
          output(n,m)=output(n,m)/int32(var(n,m))*gAux;
          %De Q15 para Q0
          output(n,m) = output(n,m)/2^15;
          %Q0 +  Q0 = Q0
          output(n,m)= output(n,m) + int32(mean(n,m));
          end
      end
```

```
end

output = double(output);

*****************************************************************************
```

Como já foi descrito na primeira etapa do projeto, o filtro de Wiener é obtido através da resolução da equação de Wiener, dada por:

$$\hat{f}(n,m) = \mu(n,m) + \frac{\max\left(0, \sigma^2(n,m) - v^2\right)}{\sigma^2(n,m)}\left(a(n,m) - \mu(n,m)\right) \tag{19}$$

Onde μ e σ^2 representam respectivamente a média e a variância dos valores contidos dentro da janela escolhida, seus valores são determinados a partir das equações abaixo.

$$\mu(n,m) = \frac{1}{NM}\sum_{i,j\in\eta} a(i,j) \tag{20}$$

$$\sigma^2(n,m) = \frac{1}{NM}\sum_{i,j\in\eta} a^2(i,j) - \left[\mu(n,m)\right]^2 \tag{21}$$

Logo, podemos observar que deveremos durante o porte para ponto fixo preocuparmo-nos especialmente com estas variáveis, além de ***invsqwindow***.

Assim como foi demonstrado na implementação do filtro da média, o pior caso de ***invsqwindow*** ocorre quando ***window*** é igual a 3. Isto implicará em uma representação de ***invsqwindow*** no formato Q19. Deve-se testar todas as operações para verificar se a representação de ***invsqwindow*** no formato Q19 não causará overflow.

Iremos inicialmente analisar a operação de ***invsqwindow*** com ***mean***, onde teremos que a multiplicação de ***invsqwindow*** com ***mean*** será armazenado em ***meanAux*** de 32bits.

$$\boldsymbol{mean}_{max} * \boldsymbol{invsqwindow}_{max} = 2^{32} - 1 \tag{22}$$

Como já sabemos o maior valor que a variável ***mean*** pode ter ocorre quando ***window*** = 5, neste caso teremos ***mean*** = 6375.

Logo,

$$\boldsymbol{invsqwindow}_{max} = (2^{32} - 1) / \ 6375 = 673720 \tag{23}$$

Desta forma podemos determinar que a representação no formato Q será:

$$q = \log_2(673720) - \log_2\left(\frac{1}{3*3}\right) = 22 \tag{24}$$

Logo, a representação de ***invsqwindow*** no formato Q19 não causará problemas de overflow na operação de ***invsqwindow*** com ***mean***.

Iremos agora analisar a operação de ***invsqwindow*** com ***var***, onde o máximo valor de ***var*** será $255^2*(5*5) = 1625625$, que ocorrerá no caso de ***window*** = 5 com uma imagem toda em branco. A operação de ***invsqwindow*** com ***var*** irá ser armazenado em ***varAux*** de 32 bits.

$$var_{max} * invsqwindow_{max} = 2^{32} - 1 \tag{25}$$

Logo,

$$invsqwindow_{max} = \frac{(2^{32} - 1)}{1625625} = 2642 \tag{26}$$

Desta forma podemos determinar a representação no formato Q, onde:

$$q = \log_2(2642) - \log_2\left(\frac{1}{3*3}\right) = 14 \tag{27}$$

Desta forma a representação de ***invsqwindow*** com Q19 iria causar overflow na multiplicação de ***invsqwindow*** com ***var***. Desta forma utilizaremos a menor representação de ***invsqwindow*** no formato Q, que como foi determinado será Q14.

Após a determinação da média e da variância iremos encontrar a saída filtrada a partir do algoritmo de Wiener adaptativo através da equação 19.

Como a saída possui valores entre 0 – 255, e a variável ***output*** que armazenará os valores de saída é do tipo *long (32bits)*, teremos 31 bits para magnitude e 1 para sinal.

Desta forma, temos:

$$q = \log_2(2^{31} - 1) - \log_2(255) = 15 \tag{28}$$

Desta forma, o algoritmo apresentará excelente precisão nos cálculos em ponto fixo, visto que calculamos o pior caso para a determinação das representações no formato Q.

5.3.2. Implementação no Code Composer Studio

Partindo do algoritmo que foi desenvolvido no MatLab, iremos agora implementar o filtro de Wiener adaptativo em C no ambiente Code Composer Studio que permitirá termos o código simulado em uma arquitetura de ponto fixo, neste caso, o DSP TMS320C55xx.

Abaixo temos o código da função do filtro de Wiener adaptativo, implementada em C.

```
*****************************************************************************************

void filterWienerFixed(unsigned char *inpArray, int window, int rows, int columns)
{
        int i,j,n,m,tmp,tmp2,lowerWindow,upperWindow;
        int N = rows;
        int M = columns;
        float noise1=0;
        long noise=0;

        long output;
        unsigned short mean[arrayN];
        unsigned short var[arrayN];



        //Tabela com os valores de 1/window*window em Q14
        unsigned short invsqwindowTable[] = {1820, 655};

        unsigned short invsqwindow;

        unsigned long meanAux;
        unsigned long varAux;
        long gAux;

        varAux = 0;
        //Seleção do 1/window*window de acordo com a entrada
        invsqwindow = invsqwindowTable[(window-3)/2];


        lowerWindow=(int)(window-1)/2;
        upperWindow=(int)((window-1)/2.0+0.5);

        for (n=0;n<N;n++){
                for (m=0; m<M; m++){
                        tmp=n*columns+m;
                        mean[tmp]=0;
                        var[tmp]=0;

                        varAux = 0;

                        for(i=n-lowerWindow; i<=n+upperWindow;i++){
                                for(j=m-lowerWindow; j<=m+upperWindow;j++){
                                        if(!(i<0 || j<0 || i>N-1 || j>M-1)){
                                                tmp2=i*columns + j;
                                                //Q0 + Q0 = Q0
                                                mean[tmp]=mean[tmp]+inpArray[tmp2];
                                                //Q0 + Q0*Q0 = Q0
```

```
                                                varAux=varAux+(unsigned long)(inpArray[tmp2] *
inpArray[tmp2]);
                                        }
                                }
                        }
                        //Q0*Q14 = Q14
                        meanAux = (unsigned long)mean[tmp] * (unsigned long)invsqwindow;
                        //De Q14 para Q0
                        mean[tmp] = meanAux >> 14;
                        //Q0*Q0 = Q0
                        meanAux =  mean[tmp]*mean[tmp];
                        //Q0*Q14 = Q14
                        varAux = varAux * (unsigned long)invsqwindow;
                        //De Q14 para Q0
                        var[tmp] = varAux >> 14;
                        //Q0 - Q0 = Q0
                        varAux = (unsigned long)var[tmp]-(unsigned long)meanAux;
                        //Em Q0
                        var[tmp] = varAux;
                        noise=noise+var[tmp];


                }
        }

        noise1=(float)noise/(float)(N*M);
        noise=(long)noise1;

        for(n=0; n<N; n++){
                for (m=0; m<M; m++){
                        tmp=n*columns+m;
                        output=inpArray[tmp]-mean[tmp];
                        gAux=var[tmp]-noise;
                        if(gAux<0){
                                gAux=0;
                        }
                        if(var[tmp] <= 0){
                                inpArray[tmp]=mean[tmp];
                        }
                        else{
                                //De Q0 para Q15
                                output=output << 15;
                                //(Q15/Q0)*Q0 = Q15
                                output=(output/(long)var[tmp])*gAux;
                                //De Q15 para Q0
                                output=output >> 14;
                                //Q0 + Q0 = Q0
                                inpArray[tmp]=output + mean[tmp];

                        }
                }
        }
        //Cópia dos dados de output para inpArray
}

*****************************************************************************
```

6. Métricas de Desempenho

Com base no número de instruções que são executadas na simulação do C55xx foi analisado o desempenho dos filtros em ponto flutuante e em ponto fixo. A determinação do número de instruções que cada filtro executa ocorreu a partir da subtração do número de instruções necessárias para a execução do código completo (*funções de leitura, escrita, configuração e a função do filtro*) por o número de instruções do código com a chamada da função do filtro comentada. Este procedimento foi realizado para os códigos em ponto flutuante e ponto fixo. Os procedimentos para a aquisição do número de ciclos que um determinado código executa estão descritos em [2]. A seguir, descreve-se os resultados obtidos para os filtros estudados.

6.1. Filtro da Média

Seguindo os procedimentos supracitados para determinação do número de instruções necessárias para execução do filtro da média, encontram-se na tabela 2 os resultados obtidos para o código em ponto flutuante.

Tabela 1 - Número de instruções em ponto flutuante

Operação	Número de Ciclos
Nº de instruções com a função filtro da média	2 953 755
Nº de instruções sem a função filtro da média	2 167 917
Nº de instruções da função filtro da média	785 838

Dado que o número de instruções necessárias para executar o filtro da média são 785 838, o tempo de execução será dado pela divisão do número de instruções pelo clock do DSP, que neste caso é 192Mhz.

Tempo de execução do filtro da média em ponto flutuante:

$$t_{float} = \frac{785838}{192Mhz} = 4.092ms \tag{29}$$

Em seguida, na tabela 3, encontram-se os resultados para o código em ponto fixo:

Tabela 2 - Número de instruções em ponto fixo

Operações	Número de Ciclos
N° de instruções com a função filtro da média	2 245 176
N° de instruções sem a função filtro da média	2 167 917
N° de instruções da função filtro da média	77 259

Tempo de execução do filtro da média em ponto fixo:

$$t_{float} = \frac{77259}{192Mhz} = .402ms \tag{30}$$

Com base nos tempos encontrados das implementações em ponto flutuante e ponto fixo, pode-se comprovar que o código implementado em ponto fixo apresenta um tempo de execução bem menor do que a execução em ponto flutuante. Para esta implementação em especial o código em ponto fixo foi 10.18 vezes mais rápido.

A imagem apresentada na figura 4 foi utilizada como imagem teste. A imagem, para este objetivo de determinação de desempenho, não necessita ser grande. Então, para reduzirmos os problemas com alocação de memória utilizou-se uma imagem 12x12, para o processamento de imagens maiores seria necessário o processamento por blocos ou alocação de memória externa que poderá ser feito em trabalhos futuros.

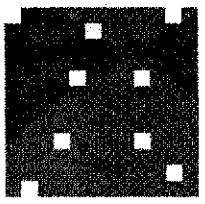

Figura 4 – Imagem 12x12 utilizada para determinação de desempenho

As figuras 5 e 6 são respectivamente o resultado do filtro da média em ponto flutuante e em ponto fixo, como pode-se notar não há diferença entre as imagens o que caracteriza a boa precisão utilizada na representação das variáveis em ponto fixo no formato Q.

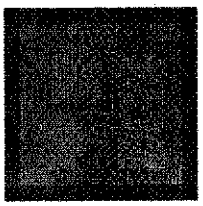

Figura 5 - Filtragem em ponto flutuante

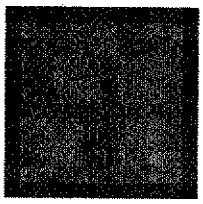

Figura 6 - Filtragem em ponto fixo

6.2. Filtro da Mediana

O algoritmo do filtro da mediana é o mesmo em ponto fixo e em ponto flutuante como já foi explanado devido à inexistência de cálculos aritméticos, consequentemente não há alteração no desempenho do algoritmo.

6.3. Filtro de Wiener

De forma semelhante aos procedimentos utilizados na determinação do número de instruções da função do filtro da média, determinou-se o número de instruções do filtro de Wiener nas aritméticas de ponto fixo e ponto flutuante. Os resultados estão apresentados nas tabelas 7 e 8.

Tabela 3 - Número de instruções em ponto flutuante

Operações	Número de Ciclos
Nº de instruções com a função filtro de Wiener	4 298 931
Nº de instruções sem a função filtro de Wiener (I/O)	2 168 040
Nº de instruções da função filtro de Wiener	2 130 891

Tempo de execução do filtro de Wiener em ponto flutuante:

$$t_{float} = \frac{2130891}{192Mhz} = 11.098ms \tag{31}$$

Tabela 4 - Número de instruções em ponto fixo

Operações	Número de Ciclos
Nº de instruções com a função filtro de Wiener	2 394 062
Nº de instruções sem a função filtro de Wiener (I/O)	2 167 917
Nº de instruções da função filtro de Wiener	226 022

Tempo de execução do filtro de Wiener em ponto fixo:

$$t_{float} = \frac{226\,022}{192Mhz} = 1.177ms \tag{32}$$

Para esta implementação o código em ponto fixo foi 9.43 vezes mais rápido. O que confirma mais uma vez as vantagens da implementação em ponto fixo frente a implementação em ponto flutuante.

Assim como no teste do filtro da média, a figura 4 foi utilizada como imagem teste. As figuras 7 e 8 são respectivamente o resultado do filtro da Wiener em ponto flutuante e em ponto fixo.

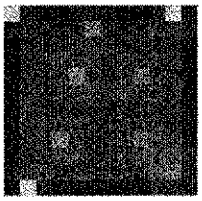

Figura 7 - Filtragem em ponto flutuante

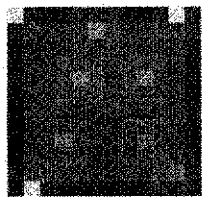

Figura 8 - Filtragem em ponto fixo

7. Conclusões

Aplicando as técnicas para o porte de algoritmos em ponto flutuante para ponto fixo nos filtros digitais, obtivemos êxito tanto do ponto de vista funcional, onde pôde ser observado que a figura filtrada em ponto flutuante e em ponto fixo se equipararam, quanto do ponto de vista de desempenho, onde através da determinação do número de ciclos observou-se que os filtros da média e Wiener foram em média 9 vezes mais rápidos do que os respectivos em ponto flutuante.

Podemos ainda destacar que o ganho no desempenho dependerá significativamente da aplicação, como no caso do filtro da mediana não haveria melhoria no desempenho visto que o mesmo não possui cálculos aritméticos que poderiam ser otimizados para uma arquitetura de ponto fixo.

Vale ressaltar ainda o compromisso que há entre precisão e memória que envolve as operações em ponto fixo. Quando aumentamos a precisão das variáveis tínhamos consequentemente um incremento significativo na memória necessária para a execução do código, que para dispositivos como DSP's é crítica devido à pequena quantidade de memória interna disponível.

8. Referências Bibliográficas

[1] Kuo, Sen M., 2001. *Real Time Digital Signal Processing:* 1st Edition. John Wiley & Sons, Ltd

[2] VASEGHI, Saeed V., 2000. *Advanced Digital Signal Processing and Noise Reduction.* 2nd Edition. John Willey & Sons.

[3] MATHWORKS, The, 2005. *MATLAB: Image Processing Toolbox User's Guide.* The MathWorks, Inc.

[4] NETWORK, The C++ Resources, 2005. *C++ Reference*. Site: http://www.cplusplus.com/ref/ (outubro de 2005)

[5] SMITH III, Julius O., 2004. *Introduction to Digital Filters with Audio Applications.* Deparment of Music, Stanford University. Site: http://ccrma.stanford.edu/~jos/filters/Definition_Filter.html (outubro de 2005)

[6] PRATT, William K., 2001. *Digital Image Processing: PIKS Inside.* 3rd Edition. John Willey & Sons.