3 Implementing filtering

The first step in the development of signal processing systems is the development and selection of the algorithms

- often an initial floating point version of the algorithm is designed
- the numerical properties the algorithm are explored using the floating point reference model
- then the design is mapped into the fixed point solution

Simulation based development of the fixed point solution

- can be based on writing C/C++ code
- alternatively the Fixed Point Toolbox can be used in Matlab
- the Simulink environment contains features for implementing fixed point simulations

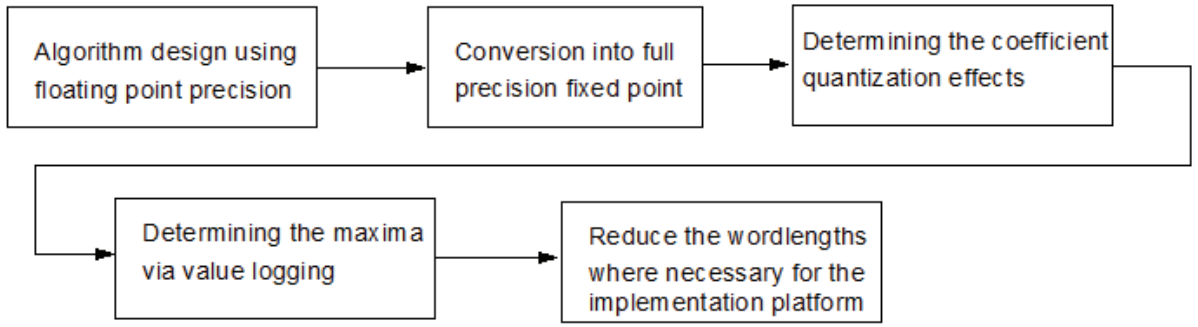

Figure 1: Outline of the tool based design process.

3.1 Developing algorithms using Matlab

The algorithms are often designed using Matlab

- Matlab is an excellent for algorithm design, but floating to fixed point conversion is a disruption point in conversion from software model to hardware implementation
- Code often needs to be rewritten to replace high-level functions and operators with the low level models of the real hardware macro-architecture
- For example, vector rotation in hardware is usually implemented using CORDIC, just employing shifts and adds, while in Matlab multiplications were used. As a result mathematical errors creep in (at least due to reduced arithmetic precision) and the algorithm is no longer the original one.

Hardware designer are normally not too proficient with Matlab

- C/C++ is commonly used for fixed point modeling
- additional verification task is introduced in the process: fixed point C/C++ against Matlab
- persistent algorithmic and numerical errors creep in, and it can be painful to track down fatal problems from among the accumulating errors

3.2 Matlab Fixed Point Toolbox

A solution for floating to fixed point conversion is using Matlab Fixed Point Toolbox, instead of resorting to C/C++ conversion of the algorithm.

- fixed point Matlab solution can be useful even if the final implementation is in software because it is much easier to debug interpreted versus compiled code

Essential behavioural modeling strengths of the Matlab Fixed Point Toolbox include the following:

1. allows defining the word lengths and the positions of the binary point in an arbitrary manner
2. includes possibility to override fixed point definitions, returning back to floating point
   (easy to check if the base algorithm is still the same and the deviations from the specification just because of the fixed point conversion process)
3. enables data logging to record minimum and maximum values, simplifying the tuning of the word lengths
4. is compatible with Filter Design Toolbox, and Simulink fixed point data types (Simulink fixed point features have been implemented using the Fixed Point Toolbox)

3.3 Matlab Fixed Point Toolbox objects

The key tools with of Matlab Fixed Point Toolbox for the design flow

- fi: defines fixed-point numeric objects: signendness, integer and fraction lengths
- fimath: defines how the overloaded operators +,- and * work with fi objects
- fipref: defines the display and logging of fi objects

Other useful tools in Fixed Point Toolbox

- numerictype: defines the data type and scaling for fi objects
- quantizer: adds bias and slope for the more enterprising individuals

However, the Matlab Fixed Point Toolbox has peculiarities from the point of view of hardware designers way of thinking

1. data abstractions, not functional abstractions the hardware designers love
   = no functional elements defined, such as 16 bit multipliers, instead operator overloading!
   the attributes of the performed operations are defined with the variables
   = object oriented thinking; understanding of object oriented programming helps

2. cant define different rounding modes for multiplications and additions with same data
   * assignments to additional variables are needed if this is desired or the fimath-properties must be changed between the operations

3. variables may adopt a new data type via a careless assignment
   * not possible to define a permanent data type and attributes for a variable!!!

3.3.1 Matlab Fixed Point Toolbox: fi object

Fixed-point number objects can be constructed with the fi command. The basic expression to use it has the form

```
fi(value, signedness, word_length, fraction_length)
```

which can be used to map values to any unsigned or signed binary-point scaled fixed point format (up:n, sp:n). Note that the fraction length n can be any positive or negative integer.

Example:

```
>> h = exp(1.0) % This is in floating point format
h = 2.7183
>> h_fixed = fi(h,1,8,3) % Changed to s8.3 format using the fi constructor
h_fixed = 2.7500
>> bin(h_fixed) % Notice that decimal point is not shown 00010.110
ans = 00010110
```

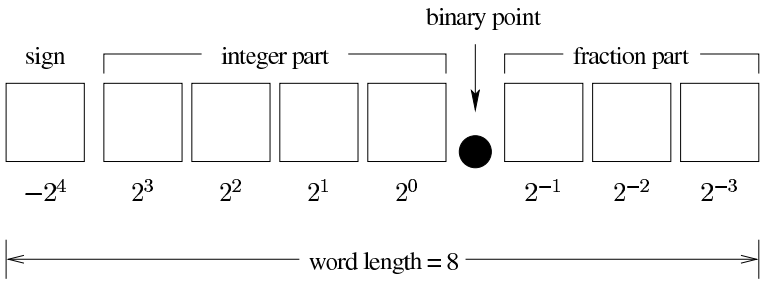

Figure 2: Representing fixed point numbers using s8.3 format.

Matlab Fixed Point Toolbox: fi object

```
>> h_fixed = fi(h,1,8,3)

h_fixed =

    2.7500

            DataTypeMode: Fixed-point: binary point scaling
                  Signed: true
              WordLength: 8
          FractionLength: 3

               RoundMode: nearest
            OverflowMode: saturate
             ProductMode: FullPrecision
   MaxProductWordLength: 128
                 SumMode: FullPrecision
       MaxSumWordLength: 128
           CastBeforeSum: true

>> get(h_fixed) % get all data fields
```

Notice that fi object contains data, numerictype and fimath properties.

Warning:

```
>> h_fixed = 2.75
>> isfi(h_fixed)
ans = 0 % 0 = not fixed point representation!
```

Matlab Fixed Point Toolbox: fi object

Example continued (using default fixed point math settings):

```
>> h = exp(1.0) % This is in floating point format
h = 2.7183
>> h_fixed = fi(h) % Changed to fixed point format using the fi constructor
h_fixed = 2.7183
>> h_fixed_s83 = fi(h_fixed,1,8,3) % format s8.3; precision was lost! 00010.110
h_fixed_s83 = 2.7500
```

We could have specified instead

```
>> h_fixed_8 = fi(hf,1,8) % notice that the fraction is not defined
h_fixed_8 = 2.7188
```

Resulting automatically the best precision that can be achieved at 8-bit wordlength: (format s8.5) 010.10111

From addition

```
>> h_fixed_8 + h_fixed_8 % in principle, addition result may add 1 bit extra
ans = 5.4375 % format s9.5, notice the automatic extension
```

Now from multiplication

```
>> h_fixed_8 * h_fixed_8
ans = 7.3916 % format s16.10: notice the automatic adjustment
```

At this point is good to realize that, for example

```
fi(h,1,8,6) % yields 1.9844 = 01111111 that is the closest representable value
```

3.3.2 Matlab Fixed Point Toolbox: fimath object

The (overloaded) arithmetic operations supported using fixed point objects are +,-, and * (division is supported via function call divide, it has some subtle limitations - better to avoid it...)

By default the results are represented in full precision

- the minimum word and fraction length with no overflow or precision loss
- helps in converting a floating point algorithm into fixed point realization - initially minimal losses of precision
- when departing full precision, overflow and rounding modes can be specified as desired (saturation, truncation, rounding, etc.)

Example on controlling the attributes of the product from multiplication (fimath-properties):

```
>> h_fixed_8.productMode = 'SpecifyPrecision';
>> h_fixed_8.productFractionLength = 3;
>> h_fixed_8.productWordLength = 8;
>> h_fixed_8 * h_fixed_8
ans = 7.3750 % (= 00111.011 s8.3 representation))
```

Matlab Fixed Point Toolbox: fimath object

The attributes of the sum are set as follows to model a 24 bit accumulator

```
>> h_fixed_8.sumMode = 'SpecifyPrecision';
>> h_fixed_8.SumFractionLength = 3;
>> h_fixed_8.sumWordLength = 8;
```

Before going any further, we check the attributes of the number

```
Signed: true, WordLength: 8, FractionLength: 5
ProductWordLength: 8, ProductFractionLength: 3
SumWordLength: 8, SumFractionLength: 3
```

Then

```
>> h_fixed_8+h_fixed_8+h_fixed_8+h_fixed_8+h_fixed_8
ans = 13.7500 % (=00000000000000001101.110)
```

while we lose some precision if the same is calculated via multiplication! (Can you explain why?)

```
>> 5*h_fixed_8
ans = 13.6250  % (=01101.101)
```

At this point it is clear that we could employ the attributes to mimic exact hardware characteristics, but we are still at some distance from finding the specifications for implementation

3.4 Implementing a FIR filter by using fixed point representation

Let us assume that the difference equation of a FIR filter with floating point coefficients is of the form (notice using the indexing scheme of Matlab)

$$y(n) = a(1) * x(n) + a(2) * x(n-1) + ...a(n-m) * x(n-m+1) \quad (1)$$

In order to be able to **exactly** model the computations of a Multiply-Accumulation -unit based design, we need to write our own FIR-filter function. (Using fi-objects and defining the arithmetic properties with fimath-constructor)

Note: our design might differ from the internal operation of dfilt of Matlab

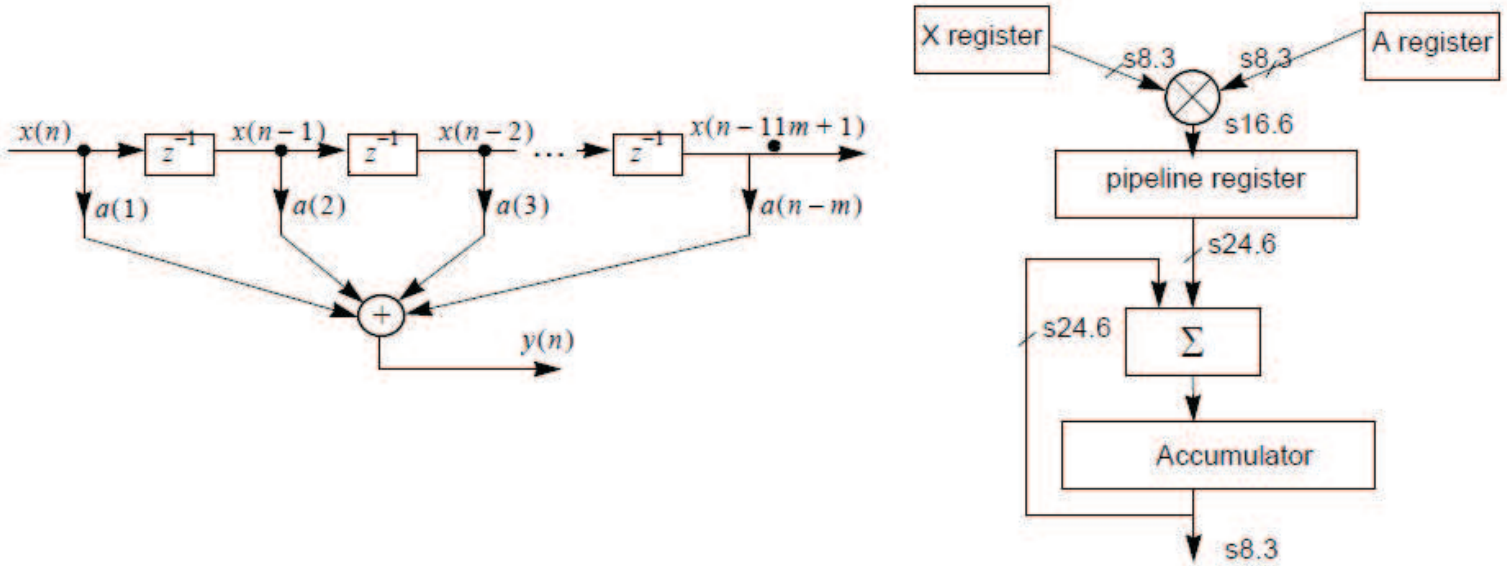

Figure 3: Implementing a FIR filter.

Implementing a FIR filter by using fixed point representation

We decide to mimic the design in the previous picture

- however, to make it reasonably efficient, hopefully without loss of modeling accuracy, we employ row-to-column vector multiplication of Matlab
- we simulate feeding in the samples one by one
- see below (code adapted from an example in Matlab documentation)

```
function y = FIR_System(a,x,y)
% x = vector of input data samples,  a = filter coefficients, y = output data samples
% notice the usual trick of using it is used input variable to set the FIR data type

% First, we create the delay line with the length of the filter
% (actually a column vector or a matrix with dimensions length(a)-by-1)
delay_line = zeros(length(a),1);
if isfi(a) delay_line = fi(delay_line,a.Signed,a.WordLength,a.FractionLength); end;

% We need to loop over the filter length(x) times
for k = 1:length(x)
  % at each iteration we bring in a sample from x and "shift" the delay line
  delay_line = [x(k); delay_line(1:end-1)]; % first [x(1) 0..] then [x(2) x(1) 0..]
  % Multiplying the coefficients and the outputs of each delay line sample
  accumulator_contents = a*delay_line;  % This is fast as it built in Matlab
  y(k) = accumulator_contents;
end;
```

Implementing a FIR filter by using fixed point representation

Initializing the filter and the signal to be filtered in floating point (filter length=17)

```
a=[-0.0136 -0.0139 0.0254 0.0523 -0.0124 -0.0880 0.0252 0.3169...
    0.4807 0.3169 0.0252 -0.0880 -0.0124 0.0523...
    0.0254 -0.0139 -0.0136]; % this is the filter (should be designed properly...)

frequency1=0.1; % lower frequency in the signal (relative to Fs=1)
frequency2=0.4; % higher frequency in the signal

N = 256; % signal length selected to make it easy to show the spectra
t = (0:N-1)'; % preparing for the sample vector of length N
x = sin(2*pi*frequency1*t) + sin(2*pi*frequency2*t); % let's use this stupid signal
y = zeros(size(x)); % we also need to create the output vector
```

Conversions into fixed point and determining the fixed point behavior of the sums and the products, and the final result

```
Fixed_Point_Attributes=fimath('ProductMode','SpecifyPrecision','ProductWordLength',16,
'ProductFractionLength',6,'SumWordLength',24,'SumFractionLength',3);
a_fixed_point = fi(a,1,8,3);
x_fixed_point = fi(x,1,8,3);
a_fixed_point.fimath = Fixed_Point_Attributes;
x_fixed_point.fimath = Fixed_Point_Attributes;
y_fixed_point = fi(zeros(size(x_fixed_point)),1,8,3);
```

Implementing a FIR filter by using fixed point representation

Then we perform the actual filtering. First using floating point

```
>> y = FIR_System(a,x,y);
```

and then use the fixed point variables (notice that the function definition with type inheritance allows to to use the same code for both fixed and floating point versions! This sounds like a great idea — but in the end I am not so convinced...)

```
y_fixed_point  = FIR_System(a_fixed_point,x_fixed_point,y_fixed_point);
```

The original signal and the results are best displayed in seperate pictures (see next page)

```
figure
subplot(411); plot(t,x);
title('Input signal','fontsize',14);set(gca,'fontsize',14);
subplot(412); plot(t,y);
title('Floating-point filtering result','fontsize',14);set(gca,'fontsize',14);
subplot(413); plot(t,y_fixed_point);
title('Fixed-point filtering resultt','fontsize',14);set(gca,'fontsize',14);
subplot(414); plot(t,y- double(y_fixed_point));
title('Difference between floating and fixed point results','fontsize',14); set(gca,'fontsize',14);
```

Implementing a FIR filter by using fixed point representation

The results with the design (we notice that s8.3 under-utilizes the dynamic range)

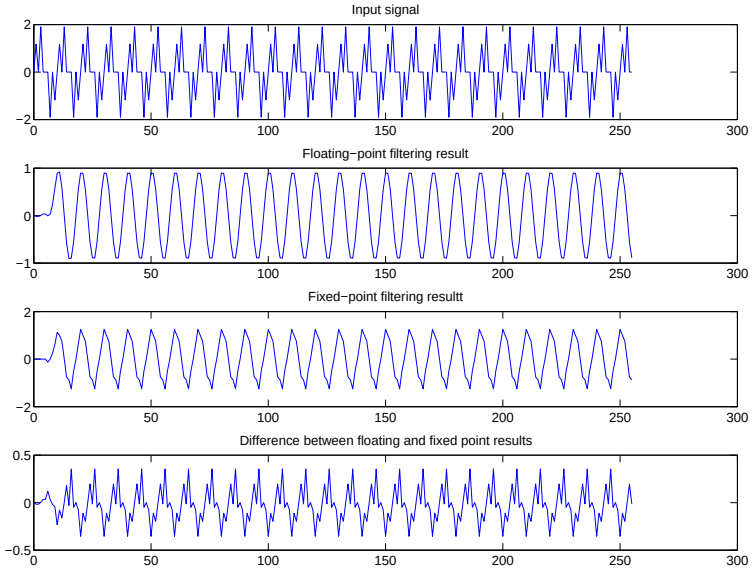

Figure 4: Fixed point filtering example: max quantization error of s8.3 format is 0.0625, but now the error in output is now around 8x higher.

Implementing a FIR filter by using fixed point representation

Plotting results:

```
[H,W] = freqz(a,1);
[H_fixed,W] = freqz(double(a_fixed_point),1);
plot(W/pi,20*log10(abs(H)),'b-',W/pi,20*log10(abs(H_fixed)),'r-');
grid on;
xlabel('Normalized frequency'); ylabel('Magnitude (dB)');
```

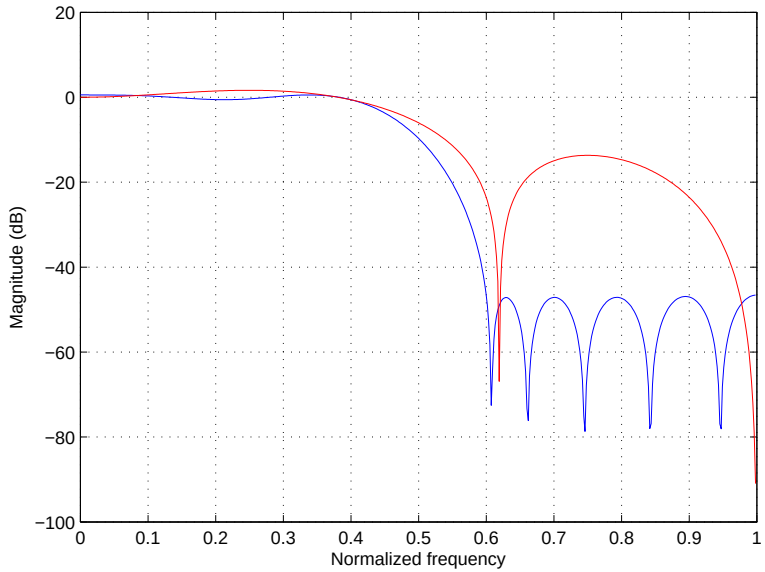

Figure 5: The response of the quantized s8.3 (red) and unquantized filter (blue); notice the impacts of coefficient quantization in the stop band (terrrrrible!).

Implementing a FIR filter by using fixed point representation

To finetune the quantization of the filter coefficients, noticing that they all are below 1, the largest one being around 0.5, we first select s8.7 representation for the coefficients

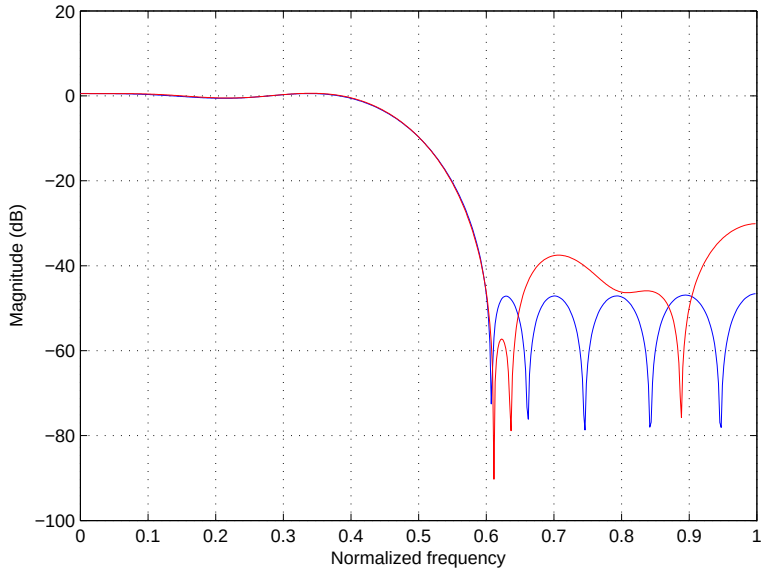

Figure 6: The response of the quantized s8.7 (red) and unquantized filter (blue).

Implementing a FIR filter by using fixed point representation

However, we can do better by selecting format s8.8: notice coefficient interval (-1,1)

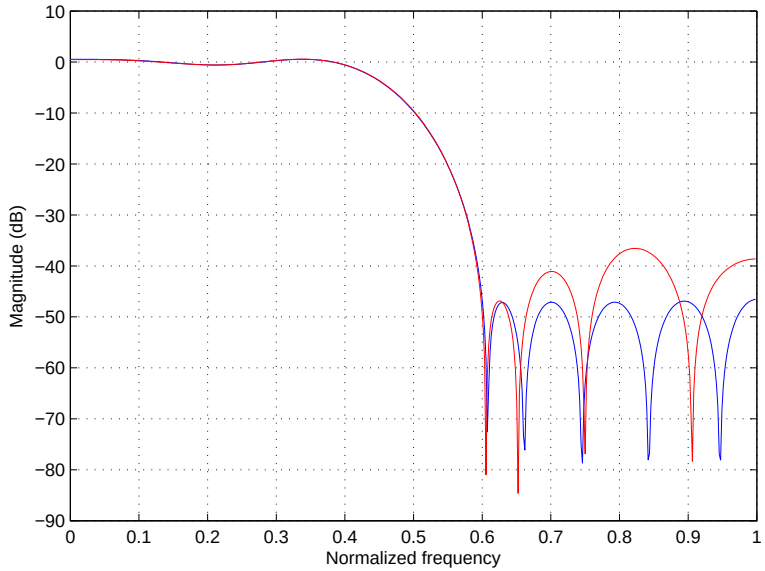

Figure 7: The response of the quantized s8.8 (red) and unquantized filter (blue).

We coud perhaps do even better based on observation that the coefficients are in (-0.5,0.5)...

Implementing a FIR filter by using fixed point representation

Format s8.8 for the coefficients results in a reformulation of our computational solution. Furthermore, as the range for the output appears to be [-1,1], the output could cope with s8.6 that will give us more decimal range - cutting the quantization noise in the output signal.

Notice that for the input we still need s8.5 due to range being [-2,2]. Based on these changes we get the solution below:

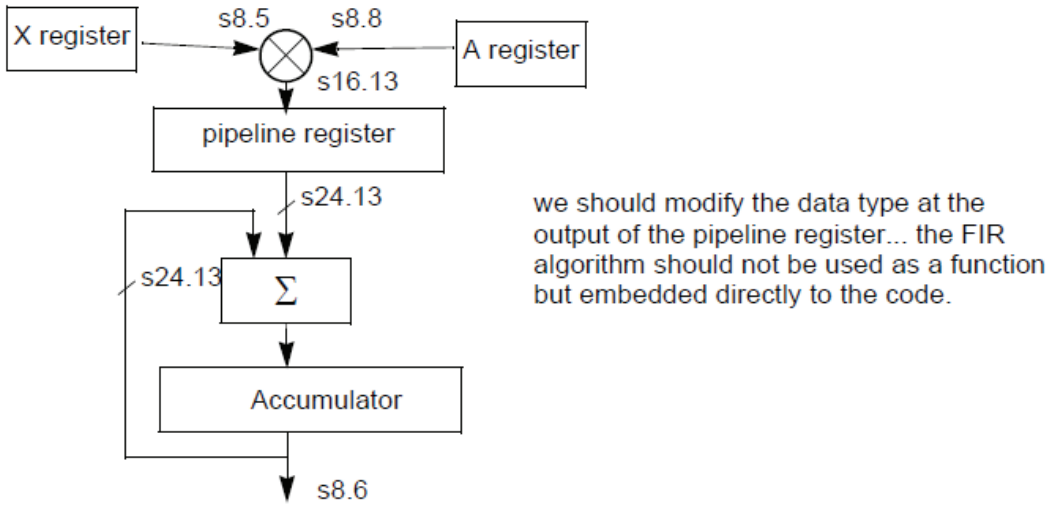

Figure 8: Implementing a FIR filter. For convenience, we keep the pipeline register data type as defined in the beginning.

```
Fixed_Point_Attributes=fimath('ProductMode','SpecifyPrecision','ProductWordLength',16,
'ProductFractionLength',13,'SumWordLength',24,'SumFractionLength',6);
a_fixed_point = fi(a,1,8,8); x_fixed_point = fi(x,1,8,5);
a_fixed_point.fimath = Fixed_Point_Attributes; x_fixed_point.fimath = Fixed_Point_Attributes;
y_fixed_point = fi(zeros(size(x_fixed_point)),1,8,6);
```

Implementing a FIR filter by using fixed point representation

The result obtained that is quite close to the best that can be achieved is below

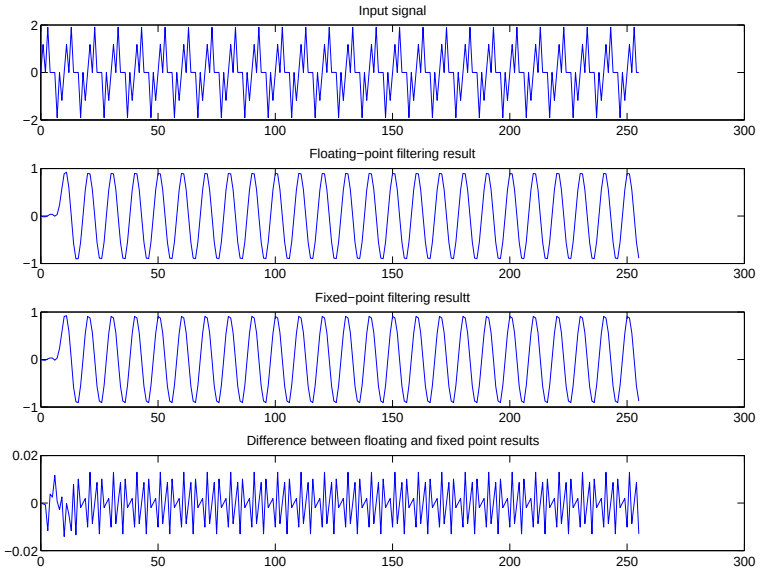

Figure 9: The maximum error in output is [-0.0141, 0.0130]. The maximum quantization error with s8.6 is 0.0078125.

Implementing a FIR filter by using fixed point representation

How good is our result? The length of the filter is 17 taps and the number of bits is 8, so the approximations formulas tell us the following in case of the worst case error

$$|E(\omega)| = 2^{-B} \sum_{m=0}^{N-1} exp(-j\omega m) = 2^{-B} N = 0.0664 \tag{2}$$

that equals to 23dB stop band attenuation, while the uniform quantization error distribution assumption

$$|E(\omega)| = (\frac{N}{3})^{1/2} 2^{-B} = 0.009299 \tag{3}$$

translates into 40.6dB stop band performance. However, these are only for the filter and do not take into account the round-off noise!

At this point we decide not to consult the analytical models, instead, we measure the actual performance from the output of our fixed point model

- from the spectrum of the output signal: 36dB average stop band attenuation
- but the spectral behavior indicates that the round-off errors may correlate, at least for fixed sine inputs
- a new design round my be necessary

Implementing a FIR filter by using fixed point representation

Above we made our analysis in a manual manner, making observations on the value ranges of the signals with our own eyes. Although that is very educational, more automation is desirable.

Automating the dynamic range analysis

1. Find the dynamic ranges at each relevant point [min,max]
2. Set the number of integer bits so that overflows are avoided
3. Set the number of fractional bits so that the SNR specs are achieved

This process can be carried out by using the "fipref" constructor* and perhaps we could reduce the word length of the accumulator to be just enough to hold just the final result that should be correct if 2's complement arithmetics is used?

Creating the fipref object that can be used for logging all fi objects

```
>> Preferences = fipref;
>> Preferences.LoggingMode = 'on';
```

Implementing a FIR filter by using fixed point representation

If we wish, we may use fipref to set, e.g., display properties or more importantly, in debugging

- to override the data type settings to check the correctness of an algorithm being migrated to fixed point precision by temporarily switching it to double precision floating point!
- Notice, that objects that are created while data type override is on, get the overriding data type and keep that type if the override is later turned off
- Remember that if you have used solutions like CORDIC, special care has to be taken by providing replacement code that will work with the floating point override!

```
>> Preferences.DataTypeOverride = 'TrueDoubles';
```

From now on, regardless how fi is used in defining the objects, they all have type 'double' but the objects created before the override, have their originally defined types!

To turn override off:

```
>> Preferences.DataTypeOverride = 'ForceOff';
```

The fipref object Preferences can be returned to default values by

```
>> reset(Preferences); % ensures that data type override and logging are off
```

Implementing a FIR filter by using fixed point representation

We issue the Matlab command

```
>> Preferences.LoggingMode = 'on';
```

and turn the overrides on: we wish to set the range and fractional parts with ease

```
>> Preferences.DataTypeOverride = 'TrueDoubles';
```

After running our algorithm we calculate the ranges (with respect to zero)

```
>> x_fixed_point = fi(x,1,8,5);
>> x_range = max(abs(double(minlog(x_fixed_point))),abs(double(maxlog(x_fixed_point))))
x_range = 1.9021
```

Now we get the integer bits for each fixed point variable of interest, first for x

```
>> x_integer_length = ceil(log2(x_range))
x_integer_length = 1 % notice no sign bit taken into account yet
```

If we have already set the word length (1 below is due to the needed sign bit; we calculated the range with respect to zero).

```
>> x_fraction_length = x_word_length-x_integer_length-1;
% If x_word_length = 8 then x_fraction_length = 6.
```

Finally, we may create a numerictype object that can later be used for setting the attributes

```
>> x_type = numerictype; % automatically signed  fixed point format
>> x_type.WordLength = x_word_length;
>> x_type.FractionLength = x_fraction_length % later we can set the attributes
>> x_fixed_point.numerictrype = x_type; % provided that we havent made a mess in the work space
```

Implementing a FIR filter by using fixed point representation

We find that the proceeding as on the previous page, we get the following numeric type proposals for our FIR filtering

```
a_fixed_point.numerictype: s8.8
x_fixed_point.numerictype: s8.6
y_fixed_point.numerictype: s8.7
```

We realize that the types for x_fixed_point and y_fixed_point differs from the s8.5 and s8.6 used earlier, res

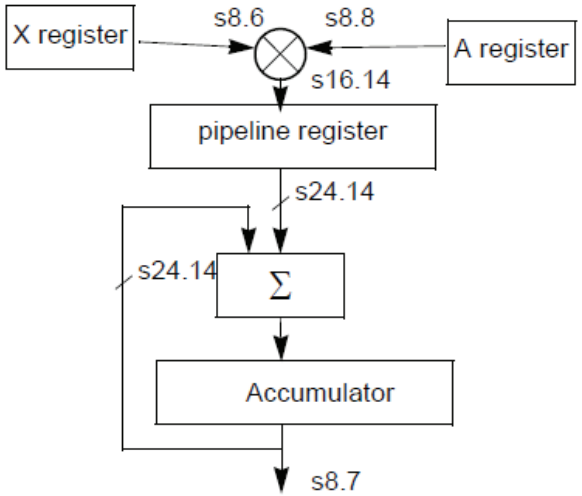

Figure 10: Modified FIR filter implementation.

Implementing a FIR filter by using fixed point representation

Now we should have optimal word lengths, but in reality we could still improve a little bit by introducing slope and bias to coefficients.

At this point we regret our long accumulator register and the respective adder that has too many gates

- we know that the final result fits in s8.7
- we could cut the integer bits to the number needed in the final result of summations provided that we allow for 2s complement overflows
- at the same time we suspect it is good to keep all the fractional bits till the very end

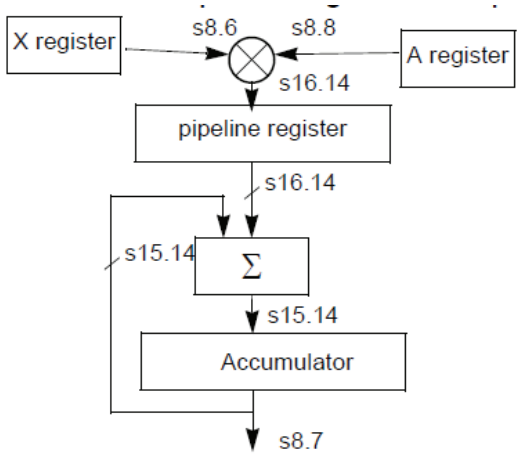

Figure 11: Again modified FIR filter implementation. The error does not change!

Implementing a FIR filter by using fixed point representation

Now we feel tempted to check what happens if we reduce the fraction bits from products

- we know that we have 17 taps, so the sum of fraction bits can travel up to ceil(log2(17)) = 5 bits
- now our products have 14 fraction bits, while the final result should not suffer if the effects of any shorter word lenght are not seen at the 7th bit
- thus we conclude that we need only 7+5=12 fraction bits for the products and sum so our design becomes the one below (this is on the safe side)

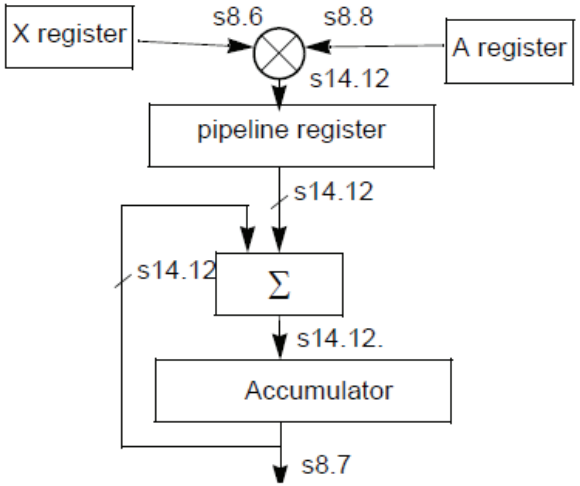

Figure 12: Once again modified FIR filter implementation. The error does not change!

In this particular case (this signal) we can reduce the fraction bits to 7 before change.

Implementing a FIR filter by using fixed point representation finally

When logging is 'on', overflows and underflows in most operations (assignments, additions, subtractions, and multiplications) generate warnings.

To stop at warnings use the usual Matlab means to locate the exact lines

```
>> dbstop in MFILENAME if warning
```

Unfortunately, this appears to be a rather obscure means to find problems unless the debugging person is well aware of the internal philosophy of Matlab objects...