Floating point implementation of an FFT block – optimization issues

Radu Suciu[(1)], Peter Reusens[(1)], Thierry Pollet[(1)], Raphael Cendrillon[(2)]

(1) Alcatel Bell
(2) Katholieke Universiteit Leuven- ESAT/SCD
radu.suciu@alcatel.be , peter.reusens@alcatel.be, thierry.pollet@alcatel.be,
raphael.cendrillon@esat.kuleuven.ac.be

Abstract

The IEEE 754 Floating point standard [1] specifies a simple (32 bits) and double precision (64 bits). For the FFT block in a DSL modem (DMT based), even 32 bits proves to be too much, and a custom number of bits can be used. This in turn will lead to savings in area and power on the ASIC implementation. Several techniques were considered for reducing the number of bits.

1. Introduction

The FFT (Fast Fourier Transform, [2]) block is of key importance while designing the ASIC for a DSL modem. The reason is that it takes a lot of area and power on the chip. The hardware implementation of the algorithm can be done in either fixed or floating point. For the floating point implementation, the IEEE standard 754 (IEEE, 1985) was considered. Although the standard specifies only a single precision (32 bits) and a double precision (64 bits), it is possible for some applications that a even 32 bits is far too much for what is needed. In the present contribution a mechanism is described for determining the optimal number of bits.

2. The FFT algorithm

The Discrete Fourier Transform (DFT), given by formula 1 is a tool for performing the Fourier transform on a set of discrete data samples.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{k\,n} \qquad (1).$$

In the above equation x(n) represent the discrete time samples, X(k) the calculated frequency ones. The twiddle factors, W_N are complex numbers, given by the formula:

$$W_N = e^{-j2\pi/N} \qquad (2).$$

The inverse transform is given by (3):

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} X(k) W_N^{-k\,n} \qquad (3).$$

If one would attempt to use the formulas stated above for performing a DFT on a set of discrete data, it would become soon obvious that the computational resources necessary are a great concern.

The Fast Fourier Transform (FFT) are just algorithms for an efficient computation of the DFT formula (1). The FFT is in fact a general name for a variety of algorithms, all of them based on two important properties of the twiddle factors:

- Symmetry: $W_N^{k+N/2} = -W_N^k$
- Periodicity: $W_N^{k+N} = W_N^k$.

The success of the FFT algorithms is due to the fact that the reduction in complex adds and multiplies is huge. For example for 1024 samples, using a radix 2 algorithm one accomplishes a computational reduction of more than 200 to 1, compared with the classical DFT. This gain increases exponentially with the size of the FFT.

Computational algorithms that exploit the symmetry and periodicity of the twiddle factors can be traced back to Gauss, as early as 1805. Nevertheless, the first FFT algorithms are generally considered to be developed by Cooley and Tukey (Cooley and Tukey, 1965). The fast algorithm is based on a structure of simple 'butterflies'. A radix 2 butterfly is presented in Fig. 1.

Such a butterfly performs additions, subtractions and multiplications. The radix 2 FFT is a very simple and straightforward algorithm. In a similar fashion one can have radix 4, radix 8, radix 16 or higher order radix decomposition.

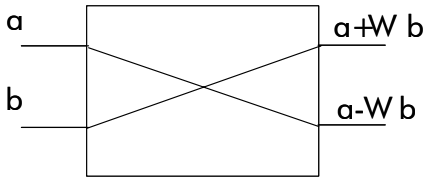

Fig. 1. Radix 2 butterfly

In fact one can have even radix 3, or radix 5 decomposition, but these are rarely used. In fact the most commonly used radices are radix 2 and radix 4. There is a vast variety of fast Fourier algorithms, and not just based on a radix decomposition. It is not in the scope of this contribution to make such an extensive study on FFT algorithms.

As factors to evaluate one or the other algorithms one can mention first the total number of operations. From this point of view, radix 4 performs better than radix 2 by 25 % in multiplications (less), the number of additions remaining constant. Another factor is the number of loads and stores from external memory. Each time a butterfly is called, it takes r values from the memory (where r is the radix type:2, 4, etc..) and after the calculation writes them back in the memory. This in turn consumes power and bandwidth. In this respect, radix 4 is also better than radix 2 by a factor of 2. The only advantage of radix 2 lies in its simplicity, especially in data addressing.

3. Data representation on a fixed number of bits

While representing data on a fixed number of bits there are two general formats that are commonly used: fixed and floating point. The pro's and con's for each one are subject to endless debates.

When talking about an FFT implementation in hardware there are several factors to be considered. At first one should consider the issue of overflow (numbers that are higher than the maximum number that can be represented) and underflow (numbers smaller than the minimum number that can be represented). The issue of overflow is more serious than underflow, since it introduces significantly more noise than underflow. Another factor to be considered is the specific hardware support on which the FFT will be implemented. In some cases as for instance FPGA (Field Programmable Gate Array) implementation, the fixed point representation appears to be the obvious solution, since a floating point would lead to an unacceptable increase in size.

In comparison with fixed point, the floating point representation is more robust to overflow and underflow. While designing an FFT core in fixed point, there are special techniques for avoiding overflow.

Scaling makes an evaluation of the worst case scenario at the output of the butterfly, and makes a scaling accordingly (for example by ½ in the case of the radix 2).

The block floating point is still a fixed point calculation, but after each pass of the algorithm an overflow test is performed. If the result of the test shows that an overflow is about to happen, an increase in 1 bit is perform for all the data in that pass. A special variant of the block floating point solution is the converging block floating point. The same as block floating point, the CBFP method operates on more isolated portions of the algorithm, allowing an efficient allocation of the extra bits.

All these techniques are rather 'messy' when a hardware implementation is sought. When an efficient scheduling is considered, such extra operations required for avoiding overflow are a burden.

On the other hand, a hardware operator (add/subtractor and multiplier) is easier for floating point.

In what follows a floating point implementation is considered for the FFT implementation, based on the IEEE 754 standard.

4. IEEE754 floating point

The standard specifies a double precision (64 bits) and a single precision (32 bits). The representation of a number in single precision is:

$$X=(-1)S * M * 2E-127 \qquad (4).$$

In the above expression, S is the bit sign, 0 for a positive number and 1 for a negative one.

M is a normalised mantissa. This means that it only takes values between 1 and 2, but always smaller than 2. Since it will always start with 1, this bit is not represented, allowing one extra bit of precision.

E is the exponent, and 127 represents the bias, or offset binary. This is a value that is specific to the 32 bits representation, and it is in fact 2(E-1)-1.

For 32 bits, one has the following structure: 8 bits exponent, 23 bits mantissa and 1 sign bit. Using the bias allows the exponent to run from 2-127 to 2+128. In Fig. 2 the 32 bit format is presented.

The standard specifies also the possibility of having small denormalised numbers. This feature allows the representation of small numbers by allowing the mantissa to be smaller than 1, (thus not normalised). This feature comes in hand when underflow must be avoided.

The 32 bit solution gives a huge dynamic range. On the other hand a reduction in number of bits is beneficial in

the particular case of the ASIC implementation of the FFT. The reduction is in size and power consumption.

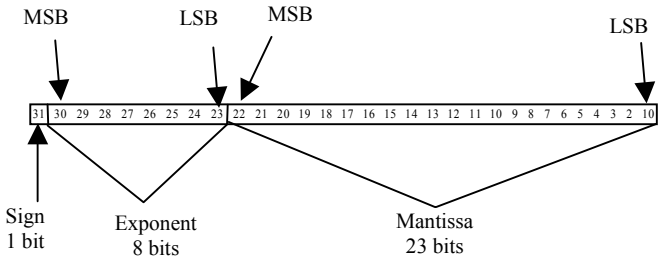

Fig. 2. IEEE 754 representation of simple precision, 32 bits

5. Optimal bit allocation

If 32 bits is most of the time far too much, than a dilemma arises: how low can one drop in number of bits without compromising the FFT. The effects of a 'shortage' in bit allocation are of two types. First, if the number of bits in exponent is too small, than overflow can appear. On the other side, if the number of bits in mantissa is too small than the effects are felt in the SNR (Signal to Noise Ratio).

In order to simulate the hardware implementation of the FFT, a Matlab model was developed. The add/subtractors and multipliers in floating point were implemented such that the model allows a large variety of 'fine tuning' operations, such as modifying the number of bits only in mantissa or in exponent, etc...

The case studied in this contribution is an IFFT size 256 (positive frequency to real time data), which is a typical ADSL transmit FFT. Nevertheless, the mechanism for determining the optimum number of bits described here works for any (I)FFT size. In fig. 3 is described the method for determining the number of bits.

At first, by an 'educated guess' one should fill in the number of bits exponent and mantissa.

At first the number of bits exponent is evaluated. A signal is randomly generated in frequency. The type of signal should be specific for the application. Then, the generated signal enters into the IFFT block in floating point (with the specified number of bits exponent and mantissa). The result then is processed into an FFT block of 'infinite precision' – usually the double precision Matlab FFT. The reason for that is the fact that the analysis is performed in the frequency domain.

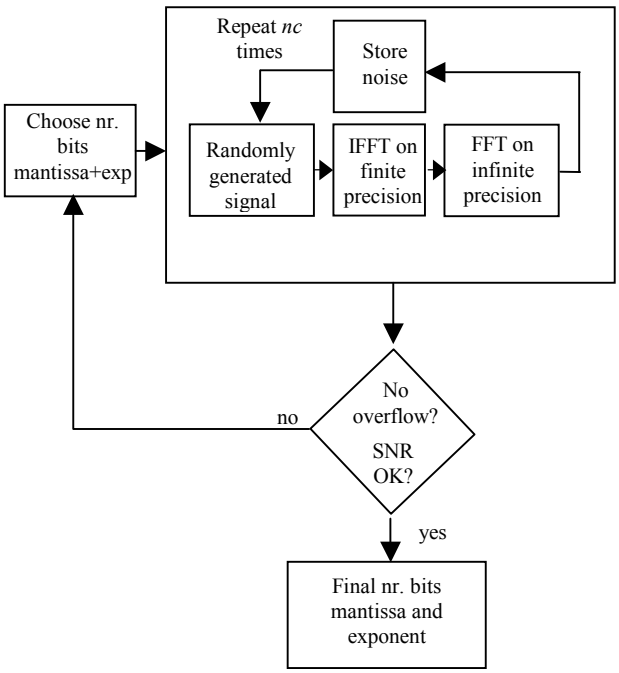

Fig. 3. Determining the number of bits mantissa and exponent

Each time a test is performed to check for overflows. This cycle is performed many times ('*nc*' times), in order to ensure a good probability distribution of the input signal. Typically *nc* is in the order of thousands. If an overflow is encountered, the number of bits in exponent is increased by one, and the cycle begins again. The target is to have a minimum number of bits in exponent, such that no overflow occurs.

Once the number of bits exponent is determined, the number of bits in mantissa is next. As mentioned before, the lack of bits in mantissa is visible in the SNR. As such, one should have clear requirements in SNR in order to be able to determine a sufficient number of bits mantissa. In this respect, it is a known empirically that an increase in mantissa with one bit leads to an increase of 6 dB in SNR.

6. Other techniques for improving the SNR/reducing the number of bits

There are several techniques that allow a further 'refinement' in bit allocation, such as:

- Internal enhanced precision
- Using rounding instead of truncation

- Small denormalised numbers.

The internal enhanced precision consists in keeping inside the memory the same number of bits, and increasing the number of bits (mantissa) inside the butterfly (where the calculations are being done). The fact that the inside the memory the number of bits is not increased leads to a reduction in area. The disadvantage of this method is the fact that one is limited in this internal increase, since a saturation effect appears. Further increase in number of bits will not lead to any significant increase in SNR.

When the result of an operation performed inside the radix cannot be fully represented with the number of bits available, the mantissa has to be adjusted. The simplest way is to use a truncation, either of the type floor (0.1 ->0, 0.9->0), or of the type ceil (0.1 ->1, 0.9->1). Such a truncation of the results may be quite simple in complexity, but leads to an increase of the noise level. A better alternative is to use rounding (0.1 ->0, 0.9->1) to the closest integer.

The small denormalised numbers is in fact a feature of the IEEE 754 standard. Normally the mantissa is normalised, such that it takes values between 1 and 2. Allowing the mantissa to be smaller than one allows the representation of very small numbers.

The last two techniques presented may or may not be implemented, depending on the increase in complexity. Both techniques imply some sort of *if then* decision making, which is not always desirable.

7. Numerical example

The numerical example presented is in fact a feasibility study for an FFT implementation, for an ADSL Alcatel modem. The transmit side has an IFFT size 256 (positive frequencies), which is typical for an ADSL modem.

The input signal is a 14 bit QAM constellation. This means that it consists of complex numbers of the type x+j*z. x and z are discrete integers that can randomly take values from [-127, -125, -123 …+123, +125, +127].

The requirements in SNR for this case are: SNR>=64 dB.

In fig. 4 a histogram is plotted for the case of 13 bits mantissa and 5 bits exponent (a total of 19 bits with the sign bit). For 8000 times (*nc* in fig. 3) a set of 256 random frequencies were generated. Each time when a set of 256 random frequencies is generated, the worst case SNR is stored (out of the 256 ones). As such, a total of 8000 worst cases SNR's are recorded, and plotted in a histogram in fig. 4. Truncation was used in this case, and no other special techniques were employed.

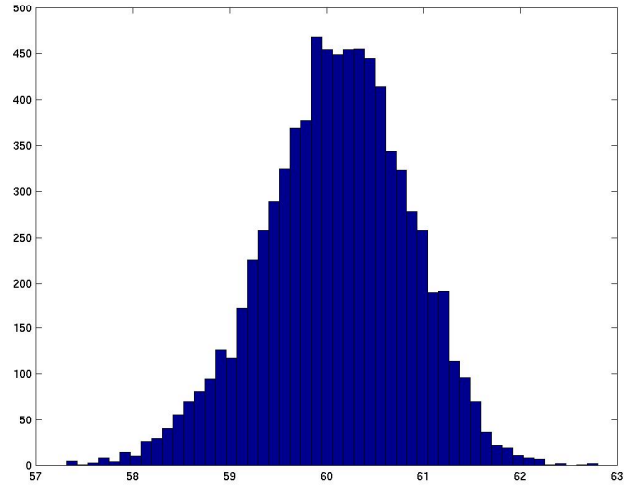

Fig. 4. SNR histogram of "worst cases", for 13+5+1 bits, truncation

The mean value of the values plotted in fig. 4 yields a value of 60.1 dB. It is clear that the requirements of minimum SNR are not met.

In fig. 5, the same plot is presented, this time using internal enhanced precision. In the external memory 13 bits mantissa are used, while inside the add/subtractors and multipliers 16 bits mantissa are used.

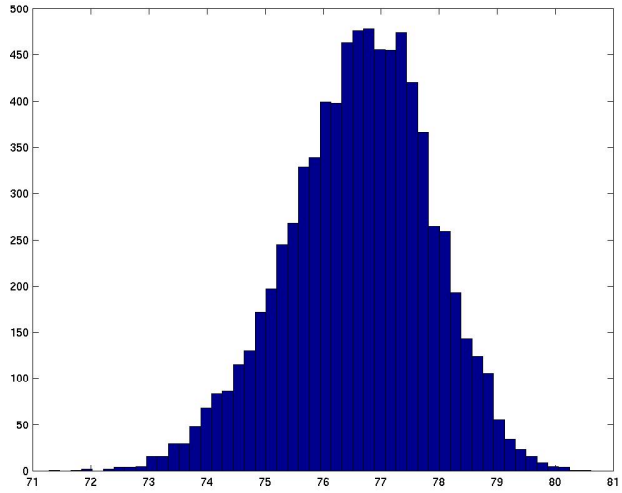

Fig. 5. SNR histogram of "worst cases", internal enhanced precision (13 bits mantissa in external memory, 16 inside the radix), truncation

The mean value of the above plot is 76.6 dB. This time the requirements in SNR are fulfilled. If the number of bits mantissa inside the radix is further increased, a

saturation effect appears, and the SNR does not increases significantly.
As an alternative to internal enhanced precision, rounding will be used instead of truncation. In fig. 6 the same plot is presented, with rounding.

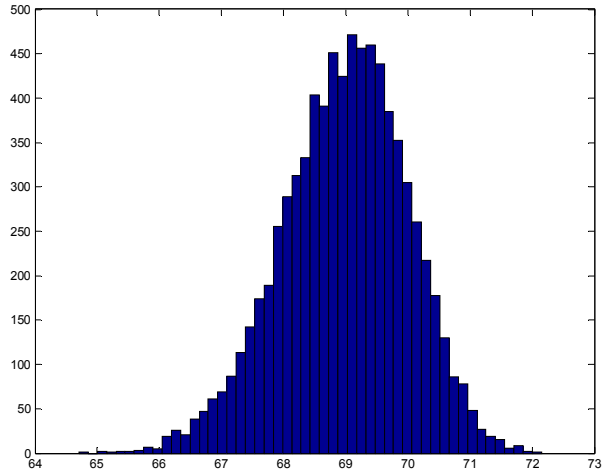

Fig. 6. SNR histogram of "worst cases", for 13+5+1 bits, rounding instead of truncation

In the above plot the mean value is 68.98 dB. Also in this case the SNR requirements are met.
For all the cases presented above the feature of small denormalised numbers was disabled.
As a conclusion to the numerical example presented, 5 bits exponent were enough. 13 bits mantissa can be used, either with 3 bits more internal enhanced precision, or using rounding instead of truncation. The decision on which technique to use (internal enhanced precision or rounding) cannot be taken from a system level. Only by a low level implementation of the FFT core, and by trial and error an answer can be given.

8. Conclusions

Several techniques were proposed for optimizing the number of bits in an floating point FFT implementation. The trade-offs were outlined, and a numerical example was presented. The general conclusion is that by simulations at a high system level some general solutions can be proposed (such as the total number of bits). The decision on what specific techniques to use (internal enhanced precision, rounding instead of truncation, etc.) can be taken only by trial and error, on low level simulations.

9. References

[1] *IEEE Standard for Binary Floating – Point Arithmetic*, 1985

[2] Cooley, J. P. Tukey, J. W., *An Algorithm for the Machine Calculation of Complex Fourier Series,* Math. of Comput., Vol. 19, No. 90, pp. 297-301, 1965.