

# An Algorithm for Converting Floating-Point Computations to Fixed-Point in MATLAB based FPGA design

Sanghamitra Roy  
Northwestern University  
Evanston, IL-60208  
1-847-467-4610

sroy@ece.northwestern.edu

Prith Banerjee  
Northwestern University  
Evanston, IL-60208  
1-847-491-4118

banerjee@ece.northwestern.edu

## ABSTRACT

Most practical FPGA designs of digital signal processing applications are limited to fixed-point arithmetic owing to the cost and complexity of floating-point hardware. While mapping DSP applications onto FPGAs, a DSP algorithm designer, who often develops his applications in MATLAB, must determine the dynamic range and desired precision of input, intermediate and output signals in a design implementation to ensure that the algorithm fidelity criteria are met. The first step in a flow to map MATLAB applications into hardware is the conversion of the floating-point MATLAB algorithm into a fixed-point version. This paper describes an approach to automate this conversion, for mapping to FPGAs by profiling the expected inputs to estimate errors. Our algorithm attempts to minimize the hardware resources while constraining the quantization error within a specified limit.

## Categories and Subject Descriptors

J.6 [Computer-Aided Engineering]: Computer-aided design;  
I.6.m [Simulation and Modeling]: Miscellaneous

## General Terms

Algorithms, Design

## Keywords

Quantization, Quantizer, Fixed point, Floating Point

## 1. INTRODUCTION

FPGA designs of DSP applications are limited to fixed-point arithmetic owing to the cost and complexity of floating-point hardware. The first step in a flow to map MATLAB applications into hardware is the conversion of the floating-point MATLAB algorithm into a fixed-point version using “quantizers” from the Filter Design and Analysis (FDA) Toolbox for MATLAB. This support is provided in the form of a quantizer object and two functions that come with this object, namely, ‘quantizer()’ and ‘quantize()’. The ‘quantizer()’ function is used to define the quantizer object which allocates the bit-widths to be used along with whether the number is signed or unsigned, what kind of rounding is to be used and whether overflows saturate or wrap. The ‘quantize()’ function applies the quantizer object to numbers. For example a quantization model of

type signed fixed-point, with 1 sign bit, 7 integer bits and 8 fractional bits, rounding to the nearest representable number towards -8, and handling overflow with saturation is defined as follows in MATLAB:

```
>> quant = quantizer(“fixed”, “floor”, “saturate”, [16 8]);  
>> Xq = quantize(quant, X);
```

This paper describes an approach to automate the conversion of floating-point MATLAB programs into fixed-point, for mapping to FPGAs by profiling the expected inputs to estimate errors.

## 2. RELATED WORK

The strategies for solving floating-point to fixed-point conversion can be roughly categorized into two groups [3]. The first one is basically an analytical approach coming from algorithm designers who analyze the finite word length effects due to fixed-point arithmetic. The other approach is based on bit-true simulation originating from the hardware designers. There has been some work in recent literature on automated compiler techniques for conversion of floating-point representations to fixed-point. The BITWISE compiler [5] determined the precision of all input, intermediate and output signals in a synthesized hardware design from a C program description. The MATCH compiler [6] developed precision and error analysis techniques for MATLAB programs. Synopsys has a commercial tool called Cocentric, which tries to automatically convert floating-point computations to fixed-point within a C compilation framework. Constantinides et al [7] have used mixed integer linear programming techniques to solve the error constrained area optimization problem. Chang et al [8] have developed a tool called PRECIS for precision analysis in MATLAB. An algorithm for automating the conversion of floating point MATLAB to fixed point was presented in [2] using the AccelFPGA compiler but that approach needed to have various default precisions of variables specified by the user when the compiler could not infer the precisions.

## 3. AUTOQUANTIZATION ALGORITHM

The autoquantization algorithm consists of five passes, explained in detail in the next few sections. We consider the following piece of MATLAB code segment to explain each step of our algorithm:

```
b=load('inputfile1');  
c=load('inputfile2');  
d=load('inputfile3');  
a(1:100)=b(1:100)+c(1:2:200).*d(1:100);
```

The .\* refers to an element by element product

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC'04, June 7–11, 2004, San Diego, California, USA

Copyright 2004 ACM 1-58113-828-8/04/0006...\$5.00.

### 3.1 Levelization

The levelization pass takes a MATLAB assignment statement consisting of complex expressions on the right hand side and converts it into a set of statements each of which is in a single operator form. For example, the 4<sup>th</sup> statement in the example code segment will be converted into two statements

```
temp1(1:100)=c(1:2:200).*d(1:100);
a(1:100)=b(1:100)+temp1(1:100);
```

The reason for this pass is to make sure that the resulting synthesized output will be bit-true with the original quantized MATLAB statement (described later)

### 3.2 Scalarization

The scalarization pass takes a vectorized MATLAB statement and converts it into scalar form using enclosing FOR loops. The levelized code segment shown above will be converted to

```
for i=1:1:100
    temp1(i)=c(2i-1)*d(i);
end
for i=1:1:100
    a(i)=b(i) + temp1(i);
end
```

We perform the above scalarization pass because we want to control the exact order in which the quantization errors can accumulate in case of a vectorized statement with accumulation.

### 3.3 Computation of Ranges of Variables

The third pass takes a levelized and scalarized MATLAB program and computes the value ranges for each variable in the program, based on the actual inputs and propagates the value ranges in the forward direction. The input variables are assigned MAX and MIN values from the maximum and minimum values of the input files. For other variables, it sets the MAX value to INFINITY and the MIN value to -INFINITY. It then executes forward propagation of values. Let us assume that the input files have the following ranges:

inputfile1:Min= 0, Max=100; inputfile2: Min=10, Max=500  
inputfile3: Min= 0 , Max=200

We perform forward propagation of ranges in the two for loops to find the ranges of temp1 and a. Hence the ranges are given below:

b: Min=0, Max=100; c: Min= 10, Max=500  
d: Min= 0, Max=200; temp1: Min=0, Max=100000  
a: Min= 0, Max=100100

### 3.4 Generation of fixed point MATLAB code

We next convert the levelized, scalarized floating-point MATLAB program into a fixed-point MATLAB program by replacing each arithmetic and assignment operation with a quantized computation. Each quantizer is set to a default maximum precision, quantizer('fixed', 'floor', 'wrap', [40 32]); After generating fixed-point code, the following code is obtained:

```
q1=quantizer('fixed', 'floor', 'wrap', [40, 32]); (same for q2-q5)
b=quantize(q1,load('inputfile1'));
c=quantize(q2,load('inputfile2'));
d=quantize(q3,load('inputfile3'));
for i=1:1:100
    temp1(i)=quantize(q4,c(2i-1)*d(i));
end
```

```
for i=1:1:100
    a(i)=quantize(q5,(b(i) + temp1(i)));
end
```

### 3.5 Auto-quantization

For each benchmark and input set, we calculate the error vector  $e$ , the difference between the output vectors for the original floating-point and the fixed-point MATLAB code obtained in Section 3.4, using actual MATLAB based floating-point and fixed-point simulation. We assume that  $outdata$  is the output vector of the MATLAB code (a in our example)

$$e = outdata_{float} - outdata_{fixed}$$

We next define an Error Metric  $EM$  using the following definition

$$EM \% = \text{norm}(e) / \text{norm}(outdata_{float}) * 100$$

We define the following terms that are used in the algorithm.

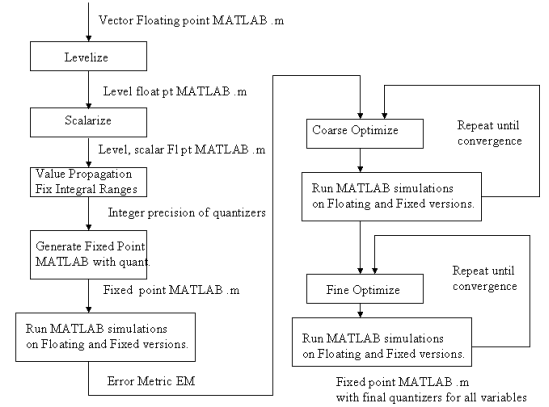
$M_{float}, M_{fixed}$  : Floating and Fixed point Matlab codes

$q_i$  is the  $i_{th}$  quantizer

$q_i$  is characterized by a [wordlength fractionlength] of [ $m_i+p_i, p_i$ ] where

$p_i, m_i$  =precision and integral length of the quantizer

$n$  = total number of quantizers in  $M_{fixed}$  which are generated for assignment/arithmetic operations



**Figure 1 Overview of Autoquantize algorithm**

Autoquantize(  $EM_{max}, M_{float}$  )

Input: Floating point MATLAB program, a set of inputs,  $EM_{max}$

Output : Fine Optimized set of quantizers

```
{  Levelize (  $M_{float}$  )
   Scalarize (  $M_{float}$  )
   Compute_Ranges (  $M_{float}$ , Input files)
   Generate_  $M_{fixed}$  (  $M_{float}$  )
   Fix_Integral_Range (list of quantizers in  $M_{fixed}$ , max range of
                        each quantizer)
   Coarse_Optimize (  $EM_{max}, M_{float}, M_{fixed}$ , list of quantizers  $q$ 
                     with known  $m_i$  )
   Fine_Optimize (  $EM_{max}, M_{float}, M_{fixed}$ , list of coarse optimized
                   quantizers  $q_i, EM_{coarse}$  )
}End Autoquantize
```

The procedure *Fix\_Integral\_Range* determines the  $m_i$  values for each  $q_i$ . The procedure *Coarse\_Optimize* determines the coarse optimized values  $p_i$  for each  $q_i$ . The procedure

*Fine\_Optimize* refines the values of  $p_i$  for each  $q_i$ . The values of  $m_i$  for each  $q_i$  are kept constant during Coarse and Fine Optimization.

**Fix\_Integer\_Range**(list of  $q_i$ , list of  $max_i$ )

```
{
  for each quantizer  $q_i$ 
     $m_i = \text{floor}(\log_2(max_i)) + 2$ 
}End Fix_Integer_Range
```

Using the above algorithm, we have the following integer ranges in our example

$q1=[8+p1,p1]$ ,  $m=8$ ;  $q2=[10+p2,p2]$ ,  $m=10$ ;  $q3=[9+p3,p3]$ ,  $m=9$ ;  
 $q4=[18+p4,p4]$ ,  $m=18$ ;  $q5=[18+p5,p5]$ ,  $m=18$

The fractional length for the quantizers for representing loop indices and flags, are set to zero. We do not apply the coarse and fine algorithm on these quantizers

**Coarse\_Optimize** (  $EM_{max}$ ,  $M_{float}$ ,  $M_{fixed}$ , list of  $q_i$  )

```
{
  Start with lowest precision quantizer(s) ( $p_i = 0/4$  bits depending on integral range, for all  $i$ ), and mark it as L, and mark the highest ( $p_i = 32$  bits for all  $i$ ) precision quantizer(s) as H. Denote M as the quantizer of precision half the difference between H and L (integral lengths kept same)

  Calculate EM values for L, H and M

  If  $EM_M < EM_{max}$  we now search in the interval [L,M], thus we replace M by H. If  $EM_M \geq EM_{max}$  we search in the interval [M,H] thus we replace M by L.

  We repeat the above two steps until we reach the coarse optimal quantizer value(s)  $M_i$  satisfying  $EM(M_i) < EM_{max}$  and  $0 \leq (M_i - L_i) < 1$  &  $0 \leq (H_i - M_i) < 1$ . We call the EM at this point  $EM_{coarse}$ 

}End Coarse_Optimize
```

In our example, if we set an  $EM_{max}$  of 0.1%, after Coarse Optimization we have the following precisions

$q1=[11, 3]$ ;  $q2=[13, 3]$ ;  $q3=[12, 3]$ ;  $q4=[21, 3]$ ;  $q5=[21, 3]$ ;  
 and  $EM_{coarse}=0.0773\%$

The Coarse Optimization algorithm follows a divide-and-conquer approach to quickly move close to the optimal set of quantizers. We call this the coarse optimal point. We vary the precisions to get a set of coarse optimized  $p_i$ s, which are all equal. But the accuracy of the output of an algorithm is less sensitive to some internal variables than to others. Hence starting from the coarse optimal point, we apply our *Fine\_Optimize* algorithm to the Coarse optimized quantizers as discussed in the next section. We don't perform finer optimization directly from the start because it will take a long time to converge. But starting from the coarse optimal point *Fine\_Optimize* moves quickly to the Fine optimal point. We assume that after coarse optimization we get quantizers  $q_1, q_2, \dots, q_n$  with precisions  $p_1, p_2, \dots, p_n$  (which are all equal).

**Fine\_Optimize** ( $EM_{max}$ ,  $M_{float}$ ,  $M_{fixed}$ , coarse optimized  $q_i$ ,  $EM_{coarse}$ )

```
{
  Set  $EM_{fine} = EM_{coarse}$ 

  Let  $EM_i = EM$  for quantizers  $q_1, q_2, \dots, q_i, \dots, q_n$  with precisions  $p_1, p_2, \dots, p_i + 1, \dots, p_n$ 

  Calculate an array DEM where  $DEM(i) = EM_{fine} - EM_i$ , for  $1 \leq i \leq n$  ( The  $i$ th element of DEM records the impact of
```

increasing the precision of the  $i$ th quantizer by 1 bit, on the total error)

- Choose the smallest element (say index  $j$ ) of array DEM for which  $p_j > 0$ ; If  $p_i = 0$  for all  $i$ , we terminate our algorithm: in this condition we cannot further reduce bits for precision
- Calculate  $EM^j = EM$  for quantizers  $q_1, q_2, \dots, q_j, \dots, q_n$  with precisions  $p_1, p_2, \dots, p_j - 1, \dots, p_n$ ; if  $EM^j \leq EM_{max}$  then  $EM_{fine} = EM^j$  and our precisions of quantizers are  $p_1, p_2, \dots, p_j - 1, \dots, p_n$  and we iteratively repeat the above three steps. If  $EM^j > EM_{max}$  we move to the next step
- Calculate DEM using the same definition above
- Find  $j$  for which  $DEM(j)$  is maximum. We also find the smallest element  $DEM(k)$  for which  $p_k > 1$ , if no such element is found we terminate the algorithm
- Calculate  $EM_{(j,k)} = EM$  for quantizers  $q_1, \dots, q_j, \dots, q_k, \dots, q_n$  with precisions  $p_1, \dots, p_j + 1, \dots, p_k - 2, \dots, p_n$
- If  $EM_{(j,k)} < EM_{max}$  then  $EM_{fine} = EM_{(j,k)}$  and  $p_j, p_k$  are changed to  $p_j + 1$  and  $p_k - 2$  and we iteratively repeat the above two steps. If  $EM_{(j,k)} > EM_{max}$  we have found our **fine optimal quantizer values**

}End Fine\_Optimize

Starting with the coarse optimized quantizers and  $EM_{coarse} = 0.0773\%$  as obtained before we get the following precisions by running our *Fine\_Optimize* algorithm

$q1=[8,0]$ ;  $q2=[13, 3]$ ;  $q3=[12, 3]$ ;  $q4=[18, 0]$ ;  $q5=[18, 0]$ ;

and  $EM_{fine} = 0.0793\%$  and this gives our fine optimized set of quantizers with a 10% Factor of safety.

*Fine\_Optimize* basically tries to find out quantizers whose impact on the quantization error is smallest and to reduce precision bits in such quantizers as long as the error is within the EM constraint. Then it tries to find a quantizer whose impact on the error is largest, and increases one bit of precision while simultaneously reducing 2 bits in a quantizer with smallest impact on the error, thereby reducing 1 bit overall. And it performs such bit reductions iteratively until the EM constraint is exceeded. Thus we attain out Fine optimal point for the given  $EM_{max}$ .

### 3.6 Use of Training sets

When using our algorithm for a specific system, we use a small percentage of the actual inputs for our algorithm. The autoquantization algorithm gives the optimal set of quantizers using this sample input. This is known as *training* of the system by sample inputs. Once we develop an optimal hardware by our algorithm, we use a larger set of the actual inputs to test our system, and verify that the actual quantization error is within the acceptable limit. While training the system we can use a factor of safety for the EM constraint. For example if our EM constraint of the actual system is 5%, and we use a  $fos = 10\%$ , the EM constraint fed to the autoquantization algorithm is  $5 * 90 = 4.5\%$ . For a system in which a smaller sample closely resembles the input signal, the factor of safety can be kept very low, or zero. Again the percentage of actual inputs for training the system can vary for different applications. But as we train the system with a very small percentage of the actual inputs, this process is really fast.

## 4. EXPERIMENTAL RESULTS

We now report experimental results on various benchmark MATLAB programs: a 16 tap FIR filter (fir), a Decimation in Time FIR filter (dec), an IIR Filter of type DF1 (iir), and a 64 point FFT

(fft). We ran MATLAB 6.5 to simulate the MATLAB fixed/floating point codes and the AUTOQUANTIZE algorithm. We have used randomly generated normalized inputs of size 2048, in the range (0,1) for our measurements. We took measurements using 5% inputs to Train, and 95% inputs to Test the system. Subsequently, we used Accelchip 2003\_3.1.72 tool to generate RTL VHDL automatically from our MATLAB benchmarks. Finally, we used the Synplify Pro 7.2 logic synthesis tool to map the designs on to Xilinx Virtex II XC2V250 device. The Accelchip 2003\_3.1.72 tool also provides an auto-quantization step. We compared our auto-quantization algorithm with the AccelChip auto-quantizer. Table 1 shows the optimal quantizers selected by the Autoquantize algorithm for EM constraints of 1%, 2% and 5% and also the actual EM values for those quantizers for training and testing inputs. Due to limited space we have shown only two quantizers per benchmark, although more quantizers have been generated in all cases. We have also provided results for maximum precision (32 bit) quantizers and the Accelchip quantizers. We report results of mapping these fixed point MATLAB designs onto FPGAs. Table 2 shows the results in terms of resources used, and performance obtained.

## 5. CONCLUSION

This paper describes how floating-point computations in MATLAB are automatically converted to fixed-point of specific precision for hardware design based on profiling the inputs. Experimental results were reported on a set of four MATLAB benchmarks that are mapped onto the Xilinx Virtex II FPGAs. The results show that it is possible to trade-off the quantization error with the hardware resources very effectively.

## 6. REFERENCES

- [1] W. Sung, and K.I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Trans. Signal Processing*, Dec. 1995
- [2] P. Banerjee, et al, "Automatic conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design," FCCM 2003, Napa Valley, CA
- [3] C. Shi, "Statistical Method for Floating-point to Fixed point Conversion," M.S. Thesis, U.C.Berkeley, EECS, 2002
- [4] P. Banerjee, et al, "AccelFPGA: A DSP Design Tool for Making Area Delay Tradeoffs While Mapping MATLAB Programs onto FPGAs," *Proc. Int. Signal Processing Conference (ISPC)*, 2003, Dallas, TX.
- [5] J. Babb, et al, "Parallelizing Applications into Silicon," *Proc. Of FPGA Based Custom Computing Machines (FCCM)*, Apr. 1999, Monterey, CA
- [6] A. Nayak, et al, "Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs," *Proc. Design Autom. and Test in Europe*, Mar. 2001, Berlin, Germany
- [7] G.A. Constantinides, et al, "Optimum Wordlength Allocation," *Proc FPGA Based Custom Computing Machines (FCCM)*, 2002, Napa, CA
- [8] M.L. Chang, S. Hauck, "Precis: A Design-Time Precision Analysis Tool," *Proc. FCCM*, 2002, Napa, CA

**Table 1 Results of Autoquantize Algorithm for Training set= 5% and Testing set=95%; Factor of safety=10%**

<b>fir16tap</b>	<b>Quantizers</b>	<b>EM% train</b>	<b>EM% test</b>
Max Prec.	qp=[33 32] qr=[34 32]	NA	4.59e-07
EM<=1%	qp=[12 11] qr=[14 12]	0.6697	0.6844
EM<=2%	qp=[11 10] qr=[13 11]	1.4852	1.5380
EM<=5%	qp=[10 9] qr=[11 9]	3.9707	4.1613
Accelchip	qp=[53 52] qr=[53 49]	NA	2.81e-12
<b>dec_fir</b>			
Max Prec.	qp=[33 32] qr=[34 32]	NA	2.47e-06
EM<=1%	qp=[15 14] qr=[16 14]	0.6820	0.6782
EM<=2%	qp=[13 12] qr=[15 13]	1.6041	1.6385
EM<=5%	qp=[11 10] qr=[14 12]	4.2151	4.3190
Accelchip	qp=[53 52] qr=[53 49]	NA	1.34e-11
<b>iirdfl</b>			
Max Prec.	qc=[34 32] qi=[33 32]	NA	4.08e-06
EM<=1%	qc=[15 13] qi=[16 15]	0.8130	0.8610
EM<=2%	qc=[14 12] qi=[15 14]	1.6310	1.7135
EM<=5%	qc=[13 11] qi=[13 12]	4.0729	4.1463
Accelchip	qc=[53 51] qi=[32 31]	NA	0.3789
<b>fft64tap</b>			
Max Prec.	qi=[33 32]qm=[34 32]	NA	3.41e-07
EM<=1%	qi=[12 11]qm=[13 11]	0.7125	0.7180
EM<=2%	qi=[11 10]qm=[12 10]	1.4777	1.4290
EM<=5%	qi=[7 6] qm=[11 9]	4.3214	4.1763
Accelchip	qi=[32 31] qm=[25 8]	NA	3.9014

**Table 2 Results on Xilinx Virtex II XC2V250 device**

<b>fir16tap</b>	<b>LUTs</b>	<b>MUX</b>	<b>MULT</b>	<b>Freq(MHz)</b>
Max Prec.	225	100	2	59.7
EM<=1%	161	75	1	76.9
EM<=2%	159	62	1	77.5
EM<=5%	155	75	1	78.0
Accelchip	809	536	6	43.6
<b>dec_fir</b>				
Max Prec.	426	45	2	69.5
EM<=1%	385	21	1	93.0
EM<=2%	369	21	1	94.5
EM<=5%	353	21	1	96.6
Accelchip	970	324	6	48.9
<b>iirdfl</b>				
Max Prec.	1020	872	8	49.8
EM<=1%	277	133	2	78.9
EM<=2%	275	126	3	68.9
EM<=5%	255	112	3	69.4
Accelchip	2035	1961	118	41.0
<b>fft64tap</b>				
Max Prec.	3164	1815	39	40.6
EM<=1%	2885	1262	27	48.8
EM<=2%	2821	1253	27	47.4
EM<=5%	2608	1230	27	49.4
Accelchip	5266	2723	48	43.9