Rapid Prototyping of FPGA based Floating Point DSP Systems

C.H. Ho[1], M.P. Leong[1], P.H.W. Leong[1], J. Becker[2], M. Glesner[3]
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T. Hong Kong.
[2] Institute for Information Processing Technology
EE-Department, Universitaet Karlsruhe (TH), Germany.
[3] Institute of Microelectronic Systems
Darmstadt University of Technology, Germany.
{chho2,mpleong,phwl}@cse.cuhk.edu.hk, becker@itiv.uni-karlsruhe.de, glesner@mes.tu-darmstadt.de

Abstract

A system for the rapid prototyping of floating point hardware designs is presented. This system, called Float*, consists of a floating point class for the simulation of quantization effects associated with low precision floating point operators; an optimizer which can automatically determine the minimal number of exponent and fraction bits required for a specified degree of accuracy; and a parameterized floating point library which can generate floating point operators with arbitrary precision. A digital sine-cosine generator is used as an example.*

1 Introduction

In the standard field programmable gate array (FPGA) based prototyping methodology, algorithms are first developed in standard programming languages such as C on a personal computer or workstation using floating point arithmetic. When the system is later implemented in hardware, a fixed point version of the algorithm is derived from the floating point version and then translated into a hardware design in a hardware description language such as VHDL. Finally, the design is synthesized for a field programmable gate array (FPGA) based prototyping environment where it can be tested.

To date, FPGA systems have almost solely used fixed point arithmetic. Although several groups have implemented floating point adders and multipliers using FPGA devices [6, 3, 2], very few systems employing floating point arithmetic have been reported.

Thus, although most scientific, digital signal processing (DSP) and financial applications are initially developed using floating point computations, its large overhead in a hardware implementation precludes its use in customized hardware. This overhead manifests itself in the form of larger area requirements and longer design time than an equivalent fixed point system.

It is envisaged that FPGA density has improved to a point where area concerns are becoming less significant, and aided by Moore's Law, density will continue to improve at an exponential rate. To address the design time issue, we present a tool called *Float* which automates the translation of a high level algorithmic description to an FPGA implementation. We believe that hardware systems employing floating point computations will become increasingly popular as the density of hardware improves, particularly in applications where variables have a very large dynamic range, or the designer wishes to avoid the complexity of translating the implementation to fixed point.

Float is embedded in the Perl programming language [7]. To use the *Float* system, a user first describes an algorithm using the *Float* class and then simulates it by executing the resulting Perl program. After correctness has been verified, an optimizer together with a set of test vectors are invoked to determine the minimum floating point precision for each variable to reach some user-specified tradeoff between quantization error and circuit size. The algorithm can then be automatically compiled to synthesizable VHDL (currently under development), operators being obtained from a parameterized VHDL *Float* library, which can then be used to make a hardware realization of the design.

A design using the above methodology has the following features:

- The designer need not have expertise in the implementation of floating point arithmetic.

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 IEEE

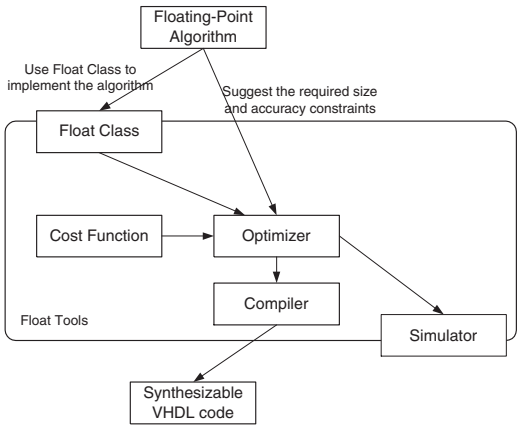

Figure 1. Floating point algorithm design flow.

- The description of the circuit is decoupled from the floating point format used in its implementation.
- The size of the exponent and fraction of the floating point number can be different for each signal in the final implementation. It would be too tedious for a designer to implement this design style manually.
- The optimizer uses a user-specified set of input vectors together with a cost function which takes into account the tradeoff between quantization error and the size of circuit.
- Design time is greatly reduced since simulation is done at a very high level and the resulting hardware implementation is correct by construction.

2 Floating Point Tools

Float consists of the following modules:

- A Perl class called *Float* for the representation of floating-point numbers. Simulation of the effect of low precision floating point operations is performed in this class.
- An optimizer which minimizes a cost function by adjusting the floating point format of the *Float* variables in an algorithm function.
- A VHDL generation module which produces synthesizable VHDL code.

Figure 1 illustrates the *Float* design flow. A designer begins by writing a Perl function, hereafter referred to as the algorithm function, to represent the algorithm to be implemented. All variables used in the algorithm are *Float* objects, where *Float* is a Perl class that is capable of representing a floating-point value under arbitrary precision. The function takes a number of *Float* variables as input and produces a number of *Float* variable as the output.

By varying the precision of the *Float* objects, the optimizer minimizes a cost function which is a weighted sum of the quantization error of the outputs of the algorithm function and the circuit size of the resulting implementation. In order to determine the outputs, a set of test input vectors are required. The algorithm function is executed with the test vectors as inputs, *Float* operators being used to perform computation. The class computes the result using both IEEE double precision and the user-specified precision. These two results are then used to compute the quantization error, with an underlying assumption that the IEEE double precision result is without quantization error, and the *Float* precision is less than double precision. Given the precision of a floating point operator, the cost function also includes a term which is an estimate of the circuit size.

Once the optimizer has determined a suitable precision for each variable in an algorithm function, a compiler can output synthesizable VHDL code for implementing the algorithm on a reconfigurable computing platform. The compiler first parses the algorithm function to produce an expression tree, each node corresponding to either a variable or an operator. The precision of variables are provided by the optimizer, and the compiler simply instantiates components with the required precision from a floating point operator module generator library.

2.1 Representation of Floating-Point Numbers

IEEE 754 double-precision floating-point numbers are 64-bits in length [1]. From left to right (most significant to least significant bit), a double has a sign bit which indicates whether the number is positive or negative, followed by 11-bits for the exponent and 52-bits for the fraction. The exponent is a signed number represented with a bias of 1024 and the fraction represents a number less than 1. The significand of the floating-point number is 1 plus the fraction part. For example, if e is the biased exponent and f is the value of fraction field, the number being represented is $1.f \times 2^{e-1024}$. When the size of the exponent is changed, the bias is changed accordingly as follows:

$$\text{bias} = 2^{\text{ebits}-1} - 1 \qquad (1)$$

where ebits are the number of exponent bits.

Float supports arbitrary precision floating point numbers. The representation used is a variant of the IEEE 754 standard, where the most significant bit is a sign bit, but the exponent and fraction precision of a *Float* number is of lower precision than IEEE double precision since double precision is used as a reference signal for the computation

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 **IEEE**

of the quantization error. *Float* does not currently support denormalized numbers, rounding modes, special values etc.

2.2 Float Class

To describe hardware that utilizes variable precision floating point computations, a class, called the *Float* class, which facilitates the simulation of arbitrary precision floating point arithmetic was developed. Perl is a modern high level programming language which offers improved productivity over traditional languages such as C. The following features of Perl were important to the design of the *Float* system:

- Perl supports objects which are used to abstract the details of variable wordlength operators.
- Perl supports operator overloading so that if x and y are *Float* objects, one can write x + y instead of x.add(y).
- Perl has strong memory management and string manipulation facilities making it easy to construct VHDL module generators.
- Perl is very portable so the *Float* design environment can run on many platforms including Unix, Linux and Windows.
- There are many open source software libraries available for Perl.

The *Float* object provides several methods for interrogation of its parameters and computation. The main ones are:

- add(), multiply(): The add() and multiply() methods will add/multiply two *Float* objects together at their specified precision, creating a new *Float* object. If the two floating point numbers have a different number of exponent bits, the output will have an exponent being the max of the two. Similarly, if the two numbers have different fraction sizes, the output will have fraction bit length equal to the max of the two input bit lengths. Overloading is used so that the $+$ and $*$ operators will invoke the add() and multiply() methods respectively.

  Apart from the arbitrary precision result, another IEEE 754 double precision floating point calculation is also computed. This value is used as a reference value for computing quantization error. Furthermore, the maximum and minimum range of this reference value is stored in the object for computation of the minimum exponent value which is required.
- setExponentSize(), setFractionSize(): The setExponentSize(), setFractionSize() methods will set the precision of a *Float* object. For setFractionSize(), the value of the object will be truncated if the fraction size will be smaller than original.
- setValue(), getValue(): These two methods are used to retrieve and write the value represented by the *Float* object. Two values are stored, the IEEE double precision reference value, and the arbitrary precision value.
- getQERR(): Both the arbitrary size floating point number and reference double precision floating point value are stored in the *Float* object. getQERR() returns their difference.

2.3 Optimization

Although any measure of accuracy could be used, average quantization error, QERR, in decibels is used in this paper. QERR is computed as follows:

$$\text{QERR} = \sum_i 20 \log \left| \frac{\text{out}_i - \text{ref}_i}{\text{ref}_i} \right| \tag{2}$$

where out_i are the outputs and ref_i are the corresponding double precision reference outputs.

The total circuit area is determined by summing the area estimate for each operator. Operator area is estimated from the precision of the *Float* class, assuming a Xilinx Virtex-E series FPGA [8]. Although the area estimation is based on a specific reconfigurable computing platform, optimization using these measures should lead to reasonable area estimates on other platforms.

The area in Virtex slices [8] occupied by floating-point adder is modeled by the following equation:

$$\text{add\_area} = 6 \times \text{ebits} + 12 \times \text{fbits} \tag{3}$$

where ebits is the number of exponent bits in the *Float* representation and fbits is the number of fraction bits.

Similarly, the area occupied by a floating-point multiplier is given by the following equation:

$$\text{mul\_area} = 230 + 8 \times \text{ebits} + \ 106 \times \left(\frac{\text{fbits}}{15} \right)^2 \tag{4}$$

The cost function is computed from the QERR and circuit area measures using the following equation:

$$f_{cost} = a \times \left(\sum_i \text{add\_area}_i + \sum_j \text{mul\_area}_j \right) + b \times \text{QERR} \tag{5}$$

where a and b are non-negative weightings and i and j sum over all the add and multiply operators in the algorithm function respectively.

The optimizer uses the Nelder-Mead [5] method to minimize the cost function (without requiring the computation

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 **IEEE**

of derivatives) by adjusting the precisions of *Float* variables in the algorithm function. The designer can adjust a and b in Equation 5 to weight the relative importance of area and QERR. For example, if the designer needs a very accurate result and circuit area is not critical, a large value of b can be used.

The optimization procedure is outlined as follows:

1. Change the precisions of *Float* variables (using Nelder-Mead).
2. Simulate the algorithm function at the specified precision using user-supplied input data.
3. Compare the result with the reference result and compute the cost function.
4. Repeat until the optimization terminates.

2.4 VHDL Generation

A simple compiler which can generate a datapath from the algorithm function is under development. Since transfer of control statements are not supported, datapath generation is simple, a code generator simply turning *Float* variables into VHDL signals, and *Float* operators into VHDL instantiations of operators from a module library.

The module library was implemented in Perl and currently supports two operators, namely multiply and add. Thus one can use the module library to generate operators with arbitrary precision. Operators are pipelined for high throughput.

3 Digital Sine-Cosine Generator

Digital sine-cosine generators [4] have a number of applications, such as the computation of discrete Fourier transform and in certain digital communication systems, such as in future Hiperlan systems for high performance wireless indoor communication. Let $s1_n$ and $s2_n$ denote the two outputs of a digital sine-cosine generator, the outputs at the next sample can be computed using the following formula:

$$\begin{bmatrix} s1_{n+1} \\ s2_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta)+1 \\ \cos(\theta)-1 & \cos(\theta) \end{bmatrix} \begin{bmatrix} s1_n \\ s2_n \end{bmatrix} \quad (6)$$

Equation 6 will be used as an example of a *Float* application in the rest of this paper, with $\cos\theta = 0.9$. Its algorithm function can be described by the Perl code below:

```
$cos_theta = new Float(8, 23, 0.9);
$cos_theta_p1 = new Float(8, 23, 1.9);
$cos_theta_m1 = new Float(8, 23, -0.1);

$s1[0] = new Float(8, 23, 0);
$s2[0] = new Float(8, 23, 1);

for ($i = 0; $i < 50; $i++)
{
  $s1[$i+1] = $s1[$i] * $cos_theta +
            $s2[$i] * $cos_theta_p1;

  $s2[$i+1] = $s1[$i] * $cos_theta_m1 +
            $s2[$i] * $cos_theta;
}
```

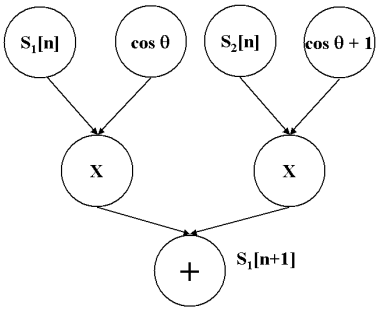

Figure 2. Expression tree for the $s1$ output of the sine-cosine generator.

This algorithm function first declares the variables used via *Float* object instantiations, each object being specified to have an 8 bit exponent and 23 bit fraction in this example. The initial value of the variable is also defined in the *Float* constructor, $s1$ and $s2$ being initialized to 0 and 1 respectively. The update values of $s1$ and $s2$ are derived using the floating-point operators provided by the *Float* class via overloading. Note that the loop is only used for simulation purposes, and is not considered part of the algorithm function.

This algorithm function can be passed to different components for processing. Normally, a set of input vectors is specified for the algorithm function, but since this particular function is an oscillator with no inputs, the time domain response is computed via the loop in the algorithm function.

The simulator can be used to determine the result and the optimizer can determine a suitable precision format for each of the five *Float* objects in the algorithm function. which minimizes the following optimization, the inner part of the algorithm function can be parsed to produce an expression tree (the expression tree for the $s1$ output of the sine-cosine generator is shown in Figure 2) and VHDL output can be generated from this tree. Finally, the VHDL output can be used for simulation and/or implementation on a reconfigurable computing platform.

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 **IEEE**

Table 1. Area and speed of the floating point library.

Fraction Size (bits)	Circuit Size (slices)	Frequency (MHz)	Latency (cycles)
Multiplication			
7	178	103	8
15	375	102	8
23	598	100	8
31	694	100	8
Addition			
7	120	58	7
15	225	46	7
23	336	41	7
31	455	40	7

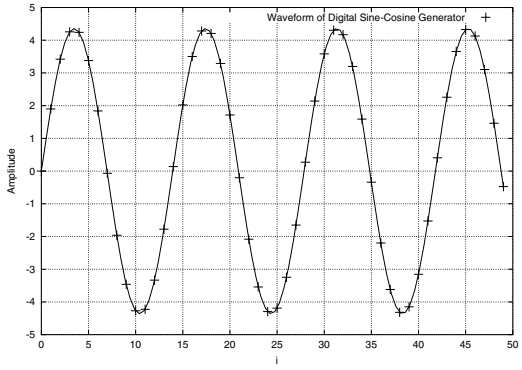

Figure 3. Digital sine-cosine generator reference output.

4 Results

4.1 Module Library

Different configurations of adders and multipliers were extracted from the module library, simulated and synthesised for the Virtex XCV1000E-6 FPGA. Table 1 is a summary of the resource requirements, maximum reported frequency and latency for a fixed exponent length of 8 bits and different fraction sizes. The adder is not yet fully optimized and the maximum frequency was 40MHz with a 7 stage pipeline.

4.2 Simulation of Algorithm Function

The algorithm function of the sine-cosine generator was simulated by directly executing it in Perl. Figure 3 shows the resulting double precision reference output.

Figure 4 shows the quantization error of the *Float* simulation for different fraction sizes, as a function of time. In the simulation, the exponent field was set to be large enough to avoid overflow. As expected, the error is reduced as the number of fractional bits (and hence precision) is increased.

Figure 5 shows the QERR of digital sine-cosine generator with a varying number of fraction bits, assuming that the exponent field is large enough to avoid overflow. For fraction bits varying from 12 to 40 bits, the QERR ranged from -50 to -210 dB.

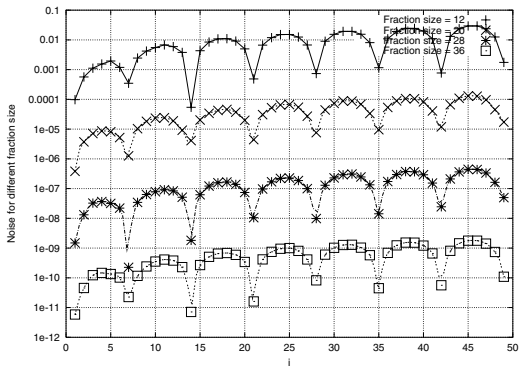

Figure 4. Quantization error of the sine-cosine generator for different fraction sizes.

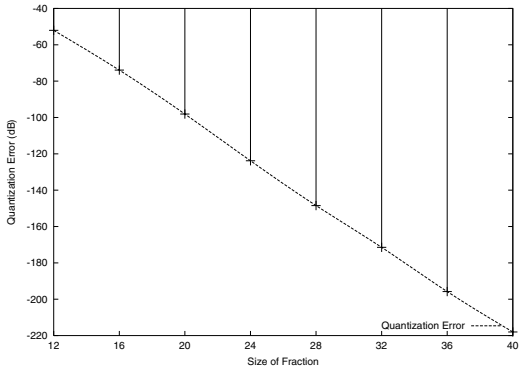

Figure 5. Quantization error for different fraction sizes.

4.3 Optimization

By varying the fraction size of the *Float* objects using the technique described in Section 2.3, the optimizer can minimize the cost function and maintain the quantization error

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 **IEEE**

Table 2. Optimization result using different QERR values where (x,y) are the (exponent size, fraction size) in bits.

QERR	s1	s2	$cos(\theta)$	$cos(\theta)$ $+1$	$cos(\theta)$ -1
-52	(5,10)	(5,12)	(5,11)	(5,11)	(5,12)
-73	(5,15)	(5,14)	(5,15)	(5,15)	(5,16)
-98	(5,19)	(5,18)	(5,19)	(5,19)	(5,20)
-123	(5,23)	(5,21)	(5,23)	(5,24)	(5,24)
-148	(5,26)	(5,28)	(5,27)	(5,27)	(5,28)
-171	(5,31)	(5,30)	(5,31)	(5,31)	(5,32)
-195	(5,35)	(5,33)	(5,35)	(5,36)	(5,36)
-218	(5,38)	(5,39)	(5,38)	(5,39)	(5,40)

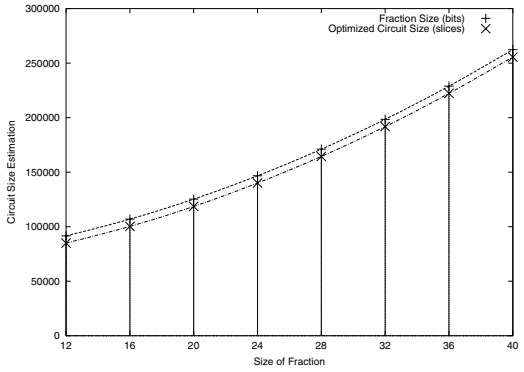

Figure 6. Area estimation of the fixed fraction and optimized circuits.

smaller than a given QERR. This technique was used to determine the minimum area requirements for a given QERR. Table 2 shows the optimized values for number of fraction bits and exponent bits for different minimum QERR. As expected, the trend for all variables is an increase in wordlength as the QERR requirement is increased.

Figure 6 compares the optimized circuit size (which allows variables to have different numbers of fractional bits) to a scheme where all variables have the same number of fraction bits (i.e. the fixed fraction case). The "Fraction Size" curve was made by computing the area of the sine-cosine generator for the case that all variables have the fraction size on the x-axis. The "Optimized Circuit Size" curve was made by using the fraction size of the x-axis as the starting point for an optimization, with the maximum QERR specified to be that of the fixed fraction case. Thus it can be seen from the figure that for the same quantization error, a 2% to 5% reduction in area is achieved by the optimization process.

In the sine-cosine generator, all variables require similar precisions. In applications where variables have widely different precisions, one would expect the scheme allowing different fractional sizes to offer a much larger improvement in area efficiency.

5 Conclusion

The *Float* environment for the rapid prototyping of floating point digital system was described. These tools enable the designers to concentrate on higher level algorithmic issue thus increasing the productivity and being able to explore more of the design space in a give time. A digital sine-cosine generator was implemented using Perl description as an example of using *Float*.

Acknowledgements

The work described in this paper was supported by a direct grant from the Chinese University of Hong Kong (Project code 2050240), the German Academic Exchange Service and the Research Grants Council of Hong Kong Joint Research Scheme (Project no. G_HK010/00).

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach 2nd Edition*. Morgan Kaufmann, 1999.

[2] A. Jaenicke and W. Luk. Parameterised floating-point arithmetic on FPGAs. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 897–900, 2001.

[3] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood. A re-evaluation of the practicality of floating-point operation of FPGAs. In *Proc. FCCM*, pages 206–215, 1998.

[4] S. K. Mitra. *Digital Signal Processing A Computer-Based Approach International Editions 1998*. McGraw-Hill, 1998.

[5] J. Nelder and R. Mead. A simplex method for function minimization. In *Computer Journal*, pages 308–313, 1965.

[6] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proc. FCCM*, pages 155–162, 1995.

[7] L. Wall, T. Christianson, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[8] Xilinx Inc. Detailed functional description. In *Xilinx Virtex-E 1.8V Field Programmable Gate Arrays Databook*, 2001.

Proceedings of the 13 th IEEE International Workshop on Rapid System Prototyping (RSP'02)
1074-6005/02 $17.00 © 2002 **IEEE**