International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:1, No:2, 2007

Simulation Tools for Fixed Point DSP Algorithms and Architectures

K. B. Cullen, G.C.M. Silvestre and N.J. Hurley
Department of Computer Science
University College Dublin – Ireland

***Abstract*— This paper presents software tools that convert the C/C++ floating point source code for a DSP algorithm into a fixed point simulation model that can be used to evaluate the numerical performance of the algorithm on several different fixed point platforms including microprocessors, DSPs and FPGAs. The tools use a novel system for maintaining binary point information so that the conversion from floating point to fixed point is automated and the resulting fixed point algorithm achieves maximum possible precision. A configurable architecture is used during the simulation phase so that the algorithm can produce a bit-exact output for several different target devices.**

I. Introduction

Fixed point DSP devices are preferred over floating point devices in systems that are constrained by complexity, cost and power consumption such as mobile phones, personal digital assistants and wearable computing devices. In general a fixed point algorithm starts life as a high level floating point simulation model. Converting the simulation model to fixed point arithmetic and then porting it to a target device is a time consuming and difficult process. DSP devices have very different instruction sets so an implementation on one device cannot be ported easily to another device if it fails to achieve sufficient quality. Choosing a target device with an abundance of resources will exceed the constraints of any low power, low cost system. For these reasons it is necessary to evaluate a fixed point DSP algorithm in terms of signal quality before work on the final implementation begins. There are tools available to convert floating point algorithms to fixed point however the conversion process usually requires a great deal of interaction from the user. Also the conversion process usually involves a certain amount of approximation so that the resulting fixed point algorithm does not achieve the maximum possible numerical accuracy. Existing tools to simulate DSP algorithms use a generic form of fixed point arithmetic which does not take architectural details of the target device into consideration such as data-bus and accumulator word lengths and the presence of Multiply Accumulate (MAC), rounding and limiting modules. As a result the simulation cannot produce a bit-exact output and gives an approximate evaluation of signal quality.

The software tools presented in this paper automatically convert floating point DSP algorithms implemented in C/C++ to fixed point algorithms that achieve maximum accuracy. The tools then simulate the algorithm on a generic architecture that can be configured to represent the datapaths in various microprocessor, DSP and FPGA devices so that one simulation model can produce a bit-exact output for several different platforms.

The rest of this paper is organized as follows: section II gives a brief overview of fixed point arithmetic, section III presents the tools and section IV gives an evaluation.

II. Fixed Point Arithmetic

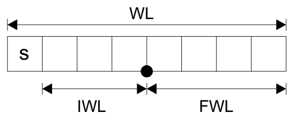

Fig. 1. Generalized fixed point format showing Word Length (WL), Integer Word Length (IWL), Fraction Word Length (FWL) and sign bit (s).

A fixed point variable consists of a binary pattern, usually in 2's compliment encoding, and a binary point. The size of the binary pattern and the location of the binary point are specified using three parameters which are indicated in Fig. 1. Word Length (WL) is the total number of bits in the binary pattern, Integer Word Length (IWL) is the number of bits to the left of the binary point not including the sign bit and Fraction Word Length (FWL) is the number of bits to the right of the binary point. This format can represent numbers in the range $[-2^{IWL}, 2^{IWL})$ with a step size of 2^{-FWL}. The parameter values determine the scaling operations required before and after fixed point operations. For example to add two variables with different FWLs the one with the larger FWL must be right shifted. Implementing an algorithm in fixed point arithmetic involves finding the binary point location for every variable and determining the scaling operations required before and after each operator.

III. Fixed Point Simulation Tools

The software tools are implemented as a C++ class hierarchy. The lowest level of the hierarchy, the `Integer` class, deals with 2's compliment binary pattern operations. Arbitrary precision arithmetic is used to overcome the limitations of the host machine running the simulation so that objects of any size can be created, added, multiplied ...etc. Each overloaded operator determines the correct word length to use for the output so that no data is lost. This means that in an arithmetic expression the word lengths of the intermediate results get larger and larger as more operations are performed until an assignment operator is encountered. The next layer is the fixed point class, `Fixed`, which is a composite of the integer class. It stores binary point information and implements the scaling operations required in fixed point arithmetic due to differences in binary point locations. These two layers conform to the SystemC standard [1]. The third layer in the hierarchy, `ArchFixed`, deals with the architectural details of the target device. Instead of allowing the word lengths of intermediate results to get larger and larger the data is processed by a configurable architecture that constrains the word lengths of intermediate results according to the size of the data-bus, accumulator, multiplier input ...etc and applies rounding and limiting at the appropriate stages. This architecture can be configured to simulate various existing or conceptual devices simply by changing parameter values.

A. Floating Point to Fixed Point Conversion

Starting with a C/C++ floating point algorithm the first step in creating a simulation model is to run the source code formatter. This tool has two functions: to replace variable definitions with fixed point types and to give each variable a unique identifier. For example the following C++ source code:

```
float f(float u1, float v1,
        float u2, float v2) {
  float y;

  y = u1 * v1 + u2 * v2;
  y = y * y;

  return y;
}
```

is replaced with:

```
ArchFixed f(ArchFixed u1, ArchFixed v1,
            ArchFixed u2, ArchFixed v2) {
  ArchFixed y;

  y.label("f:y_i1");
  y = u1 * v1 + u2 * v2;

  y.label("f:y_i2");
  y = y * y;

  return y;
}
```

Each `ArchFixed` object uses the identifier given to it (“`f:y_i1`”) to access a centrally stored table of binary point locations. This method of maintaining binary point information has been developed to overcome the limitations of the standard approach whereby IWL or FWL values are added to the variable definitions in the source code. The problem with this approach is that one source code identifier may represent several different fixed point variables. This problem manifests itself in two different ways.

The first problem is due to the structure of the original floating point algorithm. A fixed point variable is given a binary point location that depends on the range of values assigned to it. Therefore each fixed point variable should be written to once and then read one or more times. In a floating point algorithm it is not uncommon to use a single identifier more than once. In the previous example the identifier `y` is written to twice. It effectively serves as two different fixed point variables with two different binary point locations determined by the range of values in the two assignments. Because of the trade-off between range and precision it is essential to choose binary point locations that give the minimum required range and therefore the maximum precision. If one binary point location is found for `y` then one of the two fixed point variables that `y` represents will not have the correct binary point location. With the new system two different binary point locations are stored and retrieved from the table by the fixed point object `y` using the labels “`f:y_i1`” and “`f:y_i2`” to index the table. A solution to this problem was presented in [2] whereby IWL and FWL values are included in the assignment operations instead of the variable definitions however this method does not solve the second problem.

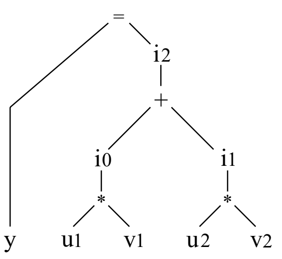

Fig. 2. (a) Example parse tree for the expression `y = u1 * v1 + u2 * v2` showing the implicit variables labelled `i0`, `i1` and `i2`

The second problem with associating binary point information to source code identifiers is that not all variables in a C/C++ program are explicitly defined. An arithmetic expression can be described using a parse tree that shows the sequence of operations and the extra variables needed to store intermediate results. These extra variables are called implicit variables. They are created by the overloaded operators in the `ArchFixed` class and cannot be referred to in the original floating point source code. An example parse tree is given in Fig. 2. In this example the result of the multiplication `u1 * v1` is assigned to an implicit variable `i0`. Because of the statistical correlation between `u1` and `v1` it is possible to left shift the product `i0` without overflow i.e

`i0 = (u1 * v1) << (i0.fwl - (u1.fwl + v1.fwl))`

where the fraction word lengths `i0.fwl`, `u1.fwl` and `v1.fwl` are determined from the range of values assigned to these variables and their word lengths. This left shift does not improve the accuracy of the product but does improve the accuracy of the remaining operations in the expression. In the above example if both `i0` and `i1` are left aligned then the effect of the right shift before addition is minimized. The output variable created by the overloaded multiplication operator in the `ArchFixed` class must represent the results `i0` and `i1` as well as every other multiplication result in the algorithm. This one variable effectively represents several different product variables that would be defined in an assembly language implementation on a target device. It must know which binary point location to use every time a multiplication is performed in order to determine the left shift amount. The standard approach used by other tools such as SystemC is simply to leave out the left shift and to interpolate the fraction word length for `i0` from `u1` and `v1` to get `i0.fwl = u1.fwl+v1.fwl`. This method of interpolation compromises numerical accuracy. The simulation tools presented in this paper solve this problem by maintaining binary point information for implicit variables using labels created by concatenating the names of the input variables. For the example in Fig. 2 the binary point location for `i0` would be stored using the label “`f:(u1*v1)`” and the addition output would be labeled as “`f:((u1*v1)+(u2*v2))`”.

The new system for maintaining binary point information allows every fixed point variable to have its own uniquely determined binary point location including multiple variables that are associated with one identifier in the original source code and those that are implied in arithmetic expressions. The advantage is that the resulting fixed point algorithm achieves maximum accuracy. A second advantage is that because the binary point information is not stored in the source code it can be changed at run time which allows the floating point to fixed point conversion process to be automated.

The standard method used to determine the actual binary point locations is called range estimation. With this approach extra code is added to the floating point version of the algorithm to monitor the range of values assigned to each variable using sample input signals. The range is then used to determine the integer word lengths. The method used by the `ArchFixed` class is based on overflow detection. When the simulation model is first created the binary points are initialized to give each variable extremely low range. The simulation is executed using sample input signals. When a variable overflows due to insufficient range its binary point is moved to the left by the overflow amount. The variables will continue to overflow and adjust until eventually they reach stable values. The table is stored as a human readable file so that binary point locations can also be specified directly by the user. This allows other methods for obtaining binary point locations [3], [4], [5] to be used. At this point the floating point to fixed point conversion is complete and the simulation phase can begin.

B. Configurable Architectural Model

In order for the simulation tools to produce a bit-exact output for different fixed point devices a configurable architecture model, which is shown in Fig. 3, was developed by comparing the major DSP and microprocessor devices to identify the important architectural differences. It consists of mandatory and optional components and is governed by three parameters: w_m, w_a and w_i which are the memory, accumulator and multiplier input word lengths respectively.

The architecture model is only concerned with numerical details. There will be some structural differences between the model and the target devices it can emulate. For example

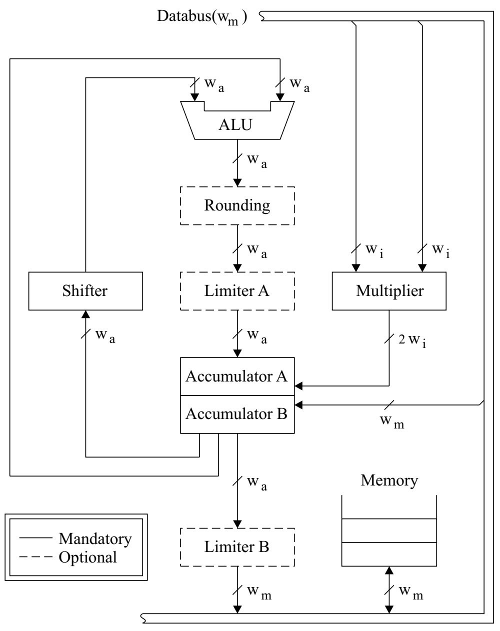

Fig. 3. Configurable Fixed Point Architecture Model. At points where a reduction in word length may occur, such as the connection between the w_m-bit databus and the w_i-bit multiplier input, the least significant bits are discarded.

devices that do not have two accumulators may still be numerically compatible with the model.

The architecture model represents the combined software/hardware operations used to carry out fixed point operations. For instance the multiplier on an integer processor does not calculate the higher order bits in the product. To prevent overflow the inputs are read into the accumulator, right shifted and written back to memory so they can be read back in. In the architecture model this is represented by the connection between the w_m bits of the databus and the w_i bits at each input to the multiplier. The architecture model can also represent software routines on the target device for double precision multiplication, addition ...etc, since these algorithms simply emulate more powerful devices.

The parameter values and optional components for various existing devices are shown in Table I. The ARM9 has only one accumulator but it can use any memory location as a second accumulator since the memory and accumulator word lengths are the same size on integer devices. Similarly the Blackfin devices have data registers that can serve as accumulators.

The configurable architecture is embedded into the overloaded operators of the `ArchFixed` class. Schematic views of the `Fixed` and `ArchFixed` multiplication operator functions are shown in Fig. 4.

IV. Evaluation

A very simple example is described in this section to demonstrate the conversion and simulation features of the tools. Minimax polynomials are often used to approximate

TABLE I
PARAMETER VALUES AND OPTIONAL COMPONENTS FOR SOME EXAMPLE PLATFORMS.

	w_m	w_a	w_i	R	LA	LB
TMS320C5x	16	32	16	n	y	n
ADSP-BF5xx (Blackfin)	16	40	16	y	y	n
DSP56xxx	24	56	24	y	n	y
ARM9	32	32	16	n	n	n

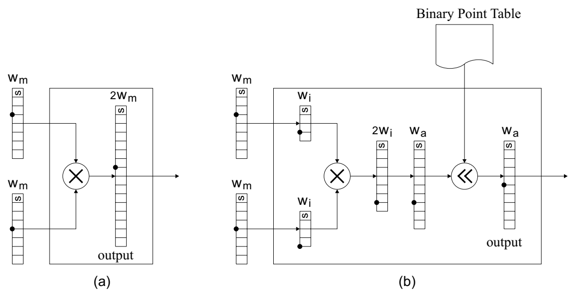

Fig. 4. Schematic view of (a) the `Fixed` and (b) `ArchFixed` multiplication operators. The `Fixed` operators create output objects that are large enough to store the result. The `ArchFixed` operators constrain the word lengths of the inputs and output according to the parameters w_m, w_a and w_i of the configurable architecture model.

functions like cosine, tangent, logarithm ...etc. These polynomials have the property that the maximum value of the approximation error is minimized. For example the 3rd degree minimax polynomial approximation to $\cos(x), 0 \leq x < \pi/2$ is the expression

`y = ((c3 * x + c2) * x + c1) * x + c0`

where the coefficients, `cn`, can be obtained using the Remez-exchange algorithm.

The binary point table produced by the tools for this algorithm will contain entries for 12 variables including "`(c3*x)`", "`((((c3*x)+c2)*x)+c1)`" ...etc.

Fig. 5 shows the Signal-to-Noise Ratio (SNR) for the minimax cosine approximation with polynomial degree on the abscissa and different values assigned to the parameters $< w_m, w_a, w_i >$. Each polynomial degree is implemented as a separate algorithm. The curves shown represent a 24-bit FPGA implementation ($< 24, 24, 24 >$), a 24-bit DSP device implementation ($< 24, 48, 24 >$) and a 24-bit integer microprocessor implementation ($< 24, 24, 12 >$). For the 7th degree polynomial the difference in SNR between the DSP and FPGA implementations is 11.26 dB. The difference between the DSP and integer microprocessor versions is 72.24 dB. All of the curves reach a maximum value of SNR that cannot be exceeded. This is due to the interaction between approximation and fixed point error which is discussed in [6]. This example illustrates the point that even for an algorithm with a small number of operations architectural details have a significant effect on fixed point error. This fact is more significant for real world systems which usually involve thousands of operations.

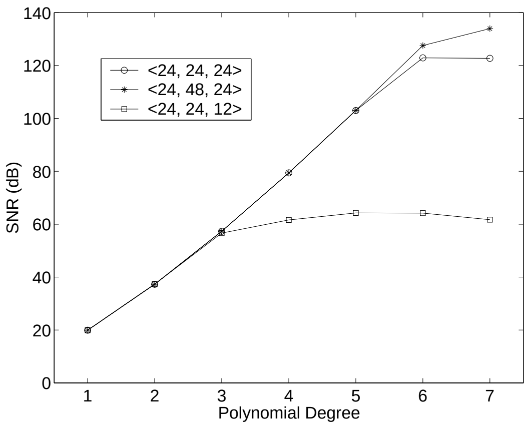

Fig. 5. Signal-to-Noise Ratio (dB) vs polynomial degree for a fixed point minimax approximation of $\cos(x), 0 \leq x < \pi/2$ with different parameter values $< w_m, w_a, w_i >$.

V. CONCLUSION

The simulation tools presented in this paper have applications in the design and rapid prototyping of a wide range of signal processing systems. They have been used successfully to convert an independently developed MPEG 2 AAC encoder [7] from floating point to fixed point. The resulting simulation model was then ported to a 32-bit ARM9 processor and part of the system was ported to an FPGA [8] device. The simulation model was used to determine the ideal architecture for the FPGA version and to establish the need for double precision arithmetic in the ARM version. These implementations have verified that the simulation results are bit-exact.

REFERENCES

[1] The Open SystemC Initiative, "SystemC Version 2.0 User's guide," *http://www.systemc.org*, 2002.

[2] Markus Willems, Volker Bürsgens, Holger Keding, Thorsten Grötker and Heinrich Meyr, "System Level Fixed-Point Design Based on an Interpolative Approach," in *Proc. 34th Design Automation Conference*, Jun. 1997.

[3] Seehyun Kim, Ki-Il Kum and Wonyong Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," in *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Nov. 1998.

[4] Sanmati Kamath, Neeraj Magotra and Ashish Shrivastava, "Quantization Analysis Tool for Fixed-Point Implementation of Real Time Algorithms on the TMS320C5000," *Proc. ICASSP*, May. 2002.

[5] Ki-Il Kum, Jiyang Kang and Wonyong Sung, "AUTOSCALER For C: An Optimizing Floating-Point to Integer C Program Converter For Fixed-Point Digital Signal Processors," in *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing*, Sep. 2000.

[6] K.B. Cullen, A. Guérin, N.J. Hurley and G.C.M Silvestre, "Evaluation of Fixed Point Elementary Functions for FPGA Audio Perceptual Coding," in *Proc. Irish Signals and Systems Conference*, Jul. 2003.

[7] K.B. Cullen, N.J. Hurley and G.C.M Silvestre, "Scalable Architecture for MPEG-2 AAC Encoders," in *Proc. Irish Signals and Systems Conference*, Jun. 2002.

[8] A. Guérin, K.B. Cullen, N.J. Hurley and G.C.M Silvestre, "FPGA Implementation of the MPEG-2 AAC Filter Bank," in *Proc. Irish Signals and systems Conference*, Jul. 2004.