Kiwi Scientific Acceleration: FPGA HLS of Custom Arithmetic including Gustafson and Yonemoto's Posit Unum

DJ Greaves (and Babar Khan).

Introduction

An appealing aspect of FPGA computation is the ability to use custom bit-widths and custom arithmetic systems. This is well known to save execution energy.

The recent increased use of machine learning algorithms has stimulated interest in custom arithmetic, both for training and deployment: the standard use of single-precision IEEE floating point is generally rekoned to be overkill for nearly all machine learning algorithms. Other application spaces where custom arithmetic is very useful include low-density parity checking and general Bayesian inference.

Using an HLS toolchain, it should be easy to replace one arithmetic system with another or reparameterise a given arithmetic system using alternative field widths. The basic approach will be to overload all of the basic arithmetic operators and allow the HLS compiler to instantiate the appropriate ALUs and custom-width data paths. In this little study we use the KiwiC HLS compiler that accepts dotnet files generated from CSharp.

(In the past I have considered implementing custom fixed and floating point precisions using Kiwi. I am not sure whether any KiwiC extensions are really needed. The operator overload facilties of C# should make all this fairly easy and just work. Generic parametes should allow either a built-in type or a user-defined type to be passed in, provided all the necessary operator overloads exist. Kiwi does not support top-level components with generic type parameters. Instead you have to write a wrapper around any top component that has type generics that fills in concrete values for those parameters.)

And it would be really great to get some evaluation of the floating point within Kiwi - hopefully we can find a sweet spot in terms of custom precision that suits a popular application. The most obvious application, of course, is convolutional neural networks, where having 24 bit mantissas is now widely accepted as being wasteful. Other examples that spring to mind are solving low-density parity codes, other Viterbi/Baysean estimators that use probability and perhaps the Spinnaker Neuon simulator from Manchester. ...

Custom arithmetic formats of interest include

- **cac_pureint** straightforward integer (or float etc, as natively supported).

- **cac_signmag** sign+magnitude integer.
- **cac_fixed** fixed field-width precision integer (using Kiwi.HwWidth() attribute where necessary).
- **cac_rational** fixed point rational.
- **cac_fp** floating point with smaller than usual exponent and mantissa precisions.
- **cac_posit** Gustafson's posit unum system PDF.

So-called Algorithmic Datatypes

There is nothing new to doing arithmetic with custom datatypes in high-level languages. SystemC is a good example of this, where all of the arithmetic operators are overloaded for the custom types. Another example is Mentor's "Algorithmic C (AC) Datatypes Software version (3.7)". In addition, you my find the same ac-datatype code on GitHub. When such a library is used under HLS, we gain a means to extend the arithmetic facilities of the HLS system while preserving perfectly readable source code.

Fortunately C# has a struct datatype that assigns by value rather than be reference (as for a class). Therefore, no nasty overloading of the assignment operator is needed. Also C# allows definition of **implicit** functions that are then interposed by the compiler in many places that would otherwise be a type error. However, there are some important differences between C++ and C# that affected this project:

- C# does not accept numeric values in generic positions (unlike C++ templates).
- C# does not allow overload selection based on generic types without the 'dynamic' keyword being inserted in the source code all over the place.
- ..

Finally, KiwiC supports the **Kiwi.HwWidth** attribute that allows specification of custom-width signed and unsigned integer registers when compiled to hardware and which reports compile-time warnings if wrapping behaviour would be different between the hardware execution and running for development under mono.

Ways of Reparameterising

Our aim is to have identical behaviour in terms of rounding and overflow on custom-precision numebers when running the code as a .NET program under mono or on FPGA via Kiwi. A non-goal is high performance when running under mono: that is for development only.

Since, in C#, we cannot pass numeric values into parameter generics (as we can in C++), we must use a slightly different approach to setting the arithmetic parameters compared with SystemC and Mentor's ac-datatypes.

The methods of parameterisation we might consider are:

- Putting each variant of arithmetic in a different .dll file and linking the client C# files against a selected one. The problem of having multiple different files all with the same name is overcome using the C# **import as** construct, whereby the client code is written using a lightweight type token throughout and the binding to an actual datatype is indirected through the **as** clause.

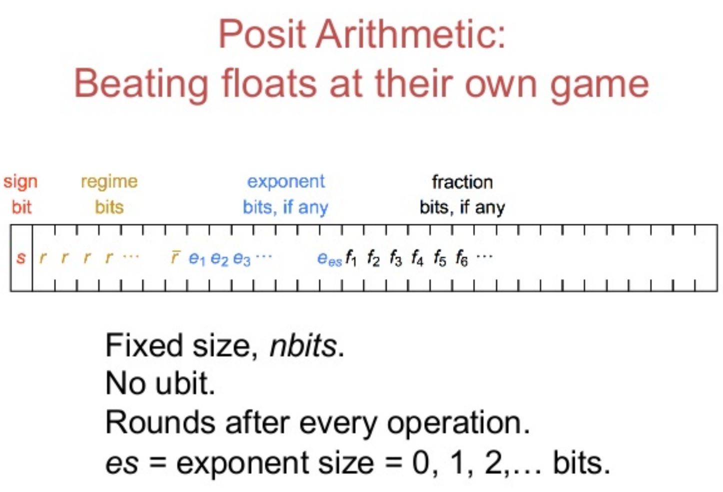
- Instantiating each type with a C# generic parameter that sets its type.

- Creating a heap object that describes the arithmetic's parameters (field widths, encoding etc.) and passing this to the constructor of our primary input data objects and/or hard-coded constant objects and then forwarding it through each overloaded operator.

- ... another ?

- Compiling a large number of forms to RTL and using a **Kiwi.RtlParameter** to select between forms at module instantiation time in its wrapping RTL substrate/shell.

Posit Unums


sign bit
regime bits
exponent bits, if any
fraction bits, if any
s r r r r ⋯ $\bar{r}$ e_1 e_2 e_3 ⋯ e_{es} f_1 f_2 f_3 f_4 f_5 f_6 ⋯

Posit Arithmetic:
Beating floats at their own game

Fixed size, *nbits*.
No ubit.
Rounds after every operation.
es = exponent size = 0, 1, 2,… bits.

The posit unum system is a variation on floating point. It has a number of distinctive features:

- The overall precision/range combination is described with two numbers: the total bit width and the number of bits in the fixed-width exponent field. These must be conveyed out-of-band.

- The exponent and mantissa compete with each other for space inside a fixed-width word: Large or small in magnitude numbers have less mantissa precision. (By 'exponent' in the previous sentence, I mean the combination of the posit's variable-length regime field and the fixed-width part of the exponent.)

- There are some 'standard' recommendations for the total and exponent fields in the cited paper. These span from small to large word sizes, as does the IEEE-754 set of standard encodings. But an HLS implementation could use virtually any combination, including an exponent field of zero. The only restriction is that the sign field always needs one bit and the regime needs at least two bits.

Beating Floating Point at its Own Game: Posit Arithmetic. John L. Gustafson and Isaac Yonemoto. PDF.

First stab C# implementation, ready for testing and performance under HLS analysis: cac_posit.cs

The only expected issue for HLS implementation, especially for small word sizes, is the much greater control-flow variation compared with more straightforward arithmetic systems, such as fixed point, or compared with hard-coded systems, such as IEEE floating point. Regarding the latter, typically the complexity of the floating-point procedures is tucked away inside structurally-instantiated ALU components that may have been hand optimised. Whereas feeding the above posit code straight into a monolithic HLS run, that typically flattens out all subroutine calling by in-lining method bodies, will perhaps give an order of magnitude more control flow complexity than normally encountered.

Fortunately, the Kiwi system supports incremental compilation, so ...

HLS Experiments

REST OF PAGE UNDER CONSTRUCTION.

Refelection API approach.

In this section we report on a minor digression that we did not fully explore ... yet ...

Although an excellent language in most respects, C# does not allow operator overloading to be based on a generic parameter. For instance, if we have a choice of algorithmic datatype and pass one in as argument T to the following fragment, we get a compile-time error.

```
class tinyTest
{
   T code1(T a, T b) { return a+b; }
}
```

The error arises even when the type provides an overload for the addition operator, which it should as a usable arithmetic system.
C# allows constraints on generic type formals using the '`T where ...`' construct. But this is not suffcient: providing a constraint on the type formal that it provides an operator overload for addition does not help. The C# compiler still needs to statically know which method to dispatch (or at least from which OO heirarchy of overloads). It cannot know this in general, under incremental compilation, since the implementation may not be provided until link editing time (which for .NET is normally the same as VM load time).

A solution in the C# language is available however. We can insert the word 'dynamic' at all such overload sites. The code then looks like this:

```
class tinyTest
{
   T code1(T a, T b) { return (T)((dynamic)a+(dynamic)b; }
}
```

The resulting code is ugly and also it has potentially poor performance under software execution owing to the dynamic lookup used at run-time, although the lookup is cached in additional, hidden static variables generated by the C# compiler (mcs anyway). However, under HLS, the HLS compiler can reverse-engineer all of the dynamic lookups assuming that the necessary information is present in the group of .dll files that are read by that HLS run. For instance, KiwiC will not accept top-level generic parameters in any compilation run, and hence all of the information will indeed be present when generating a hardware equivalent. The information is all constant at KiwiC exceution time, and so does not present any run time overhead on the FPGA.

We implemented many of the necessary reflection API mechanisms inside KiwiC, but did not finish off the study yet ... we switched to an alternative coding style instead ... The alternative is not to combine generics and operator overloading, but I think that would defeat your object.

Also, KiwiC must be extended to support much more of the .NET reflection API when 'dynamic' is used. ...

Another way ...via heap fields

Kiwi is now reading in the main parameters of such units from IP-XACT descriptions, so it is no longer necessary to alter KiwiC when the ALU blocks are changed. But I have not emphasised this work owing to even single-precision tending to be overkill for most FPGA-accelerated applications.

So instead of passing the paramaterisation of bit widths in the generic parameter slot we take a different approach.

As in https://www.mentor.com/hls-lp/downloads/ac-datatypes (or https://github.com/andres-takach/ac_types) we define our own structures for representing our parameterised numeric type and use object-oriented overloading of all standard arithmetic operators to make using these values relatively painless from a coding point of view. We also implement setters and getters.

Then we must use a lightly more elaborate syntax for declaring our registers (as is the case for ac-datatypes), such as

```
static cac_fixed v0 = new cac_fixed(10/*bits of precision*/, 4/*Initial value*/);
```

but the body of our programs that use these variables are written without complication owing to overload. Moreover, to change the precision, or indeed the whole system of number representation, only minor edits are needed. These being: 1. To change the precision, change a constant value in the application source code that is used in each register's constructor call (as opposed to the explict 10 in the example above). 2. To change the number representation, e.g. from custom fixed to custom floating point, use a different .dll file in both the mono and KiwiC runs.
If you object to changing the definition of existing code or .dll files when switching to a new basic precision, you might like to indirect through something like a

typedef. A null extension of a class operates like a typedef in C#, but inheritance is not allowed for structs as we need (I think) to make the custom datatype a struct so that assignment has the behaviour we expect (essentially a deep copy rather than an object handle copy, which would be fatal). There are probably various C# tricks for doing this more neatly that I am not aware of. But one way that is neat and works is for the custom datatype to accept a struct (or class) as the precision operand to its constructor that sets not only the precision but also the basic representation via a second field that is an enumeration and where the method bodies of the overloads dispatch to the appropriate implementation based on this enumeration. Hence, to extend the range of representations supported and to switch between them we do not need to upset the previous meaning or behaviour of any identifier.

However, if the aim is to set the width of registers in a hardware design, it does not matter if the computation on these parameters is done inside the C# compiler or inside the Kiwi compiler: either way it will not take place at hardware run time.

So converting the ac_fixed to a C# equivalent has to be done in a way that passes the numbers around in C# data structures that are all 'elaborated away' by the Kiwi compiler.

There' an example coding style that enables this to be done through the generics in this article https://www.codeproject.com/Articles/33617/Arithmetic-in-Generic-Classes-in-C which might be worth exploring, but basically you could just as easily put the parameters into a base heap object that is pointed at by all concrete expression nodes formed in the course of a computation and rely on the HLS synthesiser detecting that the values in this base object are compile-time constants for the HLS synthesis and promulgating their value to all elaboration steps so they do not end up being represented at hardware run time (except in terms of their main use in setting register widths, just like Verilog's parameter values).

Q. Can I pass constant expressions into my attributes, such as \verb+Kiwi.HwWidth(), to make highly-parameterisable code? When do the constant expressions get evaluated? Can values set via \verb+Kiwi.RtlParameter()+ be used within hardware width expressions attributes? Posit Unum Overheads

Kiwi is now reading in the main parameters of such units from IP-XACT descriptions, so it is no longer necessary to alter KiwiC when the ALU blocks are changed. But I have not emphasised this work owing to even single-precision tending to be overkill for most FPGA-accelerated applications.

Conclusions

TBC ...

References

Beating Floating Point at its Own Game: Posit Arithmetic. John L. Gustafson and Isaac Yonemoto. PDF.

June-Sept 2017. UP.