Verifying FMA with Hector

This paper is a tutorial on how to use Hector to verify RTL which implements floating-point fused multiply-add (in 16, 32 or 64 bit formats). In order to explain this process in some detail, we have provided a sample RTL design which implements the multiply-add operation and an actual Hector command script which was used to verify this design. To verify your RTL, you will need the following three things:

1. A reference, coded in C/C++, which is a correct behavioral implementation of the same computation. In order to create a reference model from this code, Hector requires that it be written to comply with specific guidelines. Please refer to section 4.5 of the Hector User's Guide.
2. Your RTL, written in synthesizable Verilog/SystemVerilog or VHDL.
3. A command script, written in the Tcl language, to tell Hector how to build models of your reference design and your RTL, how inputs should be driven into the two designs, and what outputs should be compared.

Please note that this paper is not intended to be a substitute for the Hector User's Guide. Rather, it is intended as a supplement to it. Note that even though Hector can be used to compare two C/C++ implementations or two RTL implementations, the focus of this paper is on verifying RTL by comparing it against a "known-good" behavioral implementation coded in C/C++.

C/C++ Reference Design

You must have a reference design in C/C++. This can be either C/C++ code developed by your company, or a publicly-available reference design such as the Berkeley SoftFloat library, which is available at: http://www.jhauser.us/arithmetic/SoftFloat.html. Once you have determined what you will use as a reference, you will need to write a simple wrapper around it in order to enable Hector to process it. In our example (wherein we verify RTL which implements fused floating-point multiply-add), we will use the SoftFloat function f32_mulAdd(): this takes three inputs a, b and c (multiplier, multiplicand and addend respectively) and returns the result of a * b + c. This module also accepts an integer in [0,4] which indicates the rounding mode to be applied – this is supplied as a global variable. Finally, this module also sets a global variable to indicate which exceptions (if any) were generated as a result of the computation.

The supplied code in maddxx.cc and mulxx.cc (xx is 16, 32 or 64) shows what is involved in writing a wrapper around the reference code. Essentially, the inputs and outputs of the reference code are defined as variables of the right type, and they are declared to Hector as input and output ports using the Hector::registerInput() and Hector::registerOutput() calls – for details, please refer to section 4.5.1 of the Hector User's Guide. You must also map the right input to the global variable softfloat_roundingMode if you are using SoftFloat as a reference. If you are using your own code as a reference, please refer to the appropriate documentation.

Specifically, for multiply-add, you need to define 4 inputs (multiplier, multiplicand, addend and rounding mode) and 2 outputs (the result and a word containing the exception flags) for Hector. maddxx.cc is a working example of such a wrapper file.

RTL

For this demonstration, we have provided working RTL (implemented in SystemVerilog) which implements 16, 32 or 64 bit floating-point fused multiply-add. Please note that this RTL is implemented in a “school-book style” with an emphasis on readability – it is not intended for use in a production design! The RTL can be instantiated to do 16, 32 or 64 bit MADD using the parameter SIZE in the instantiation. Figure 1 shows a block diagram of the RTL.

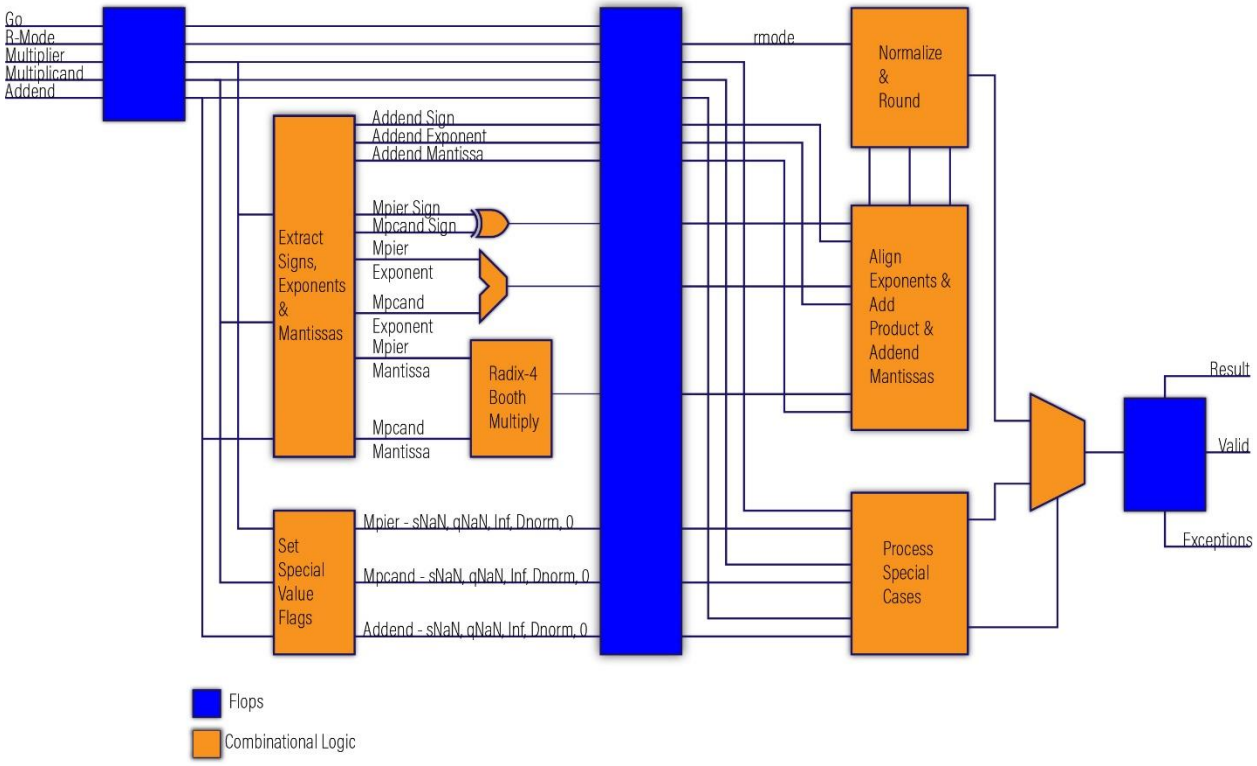

Figure 1: Multiply-add Block Diagram

The RTL consists of 3 files, as follows:

- booth4.sv – this file contains an implementation of radix-4 unsigned multiplication using Booth’s algorithm, and is used to multiply the mantissas of the multiplier and the multiplicand. In sharp contrast with a production-worthy design, this module uses a set of full-adders to add up the Booth-encoded partial products (instead of a more efficient implementation using compressors and a CSA).
- clzmadd.sv – this is a small RTL module which is used to count leading zeroes in the final result as a prerequisite for normalization and rounding.
- muladda.sv – this code implements the actual multiply-add computation. It is implemented as a 2-stage pipeline. The first pipeline stage extracts the signs, exponents and mantissas of the operands, invoked the Booth module to multiply the mantissas of the multiplier and the multiplicand, and also analyses each operand to initialize a set of flags which indicate whether the operand is a special case such as sNaN, qNaN, infinite, denormal or zero. These flags are used in the next pipeline stage.

The second pipeline stage adds the product and addend (after appropriate alignment of the mantissas), performs normalization and rounding, and also processes special cases such as NaNs and infinities.

The RTL accepts the following inputs:

- Multiplier
- Multiplicand
- Addend
- Rounding mode
- Reset (active low)
- Clock
- A “go” signal which starts a computation when it is high in a given clock cycle

The outputs of the RTL are:

- The result, a * b + c
- A 5-bit quantity which contains the exception flags generated by the computation. The mapping of these 5 bits is as follows:
    - Bit 0 – Inexact
    - Bit 1 – Underflow
    - Bit 2 – Overflow
    - Bit 3 – Infinity
    - Bit 4 – Invalid
- A “valid” signal, which when high, indicates that a valid output is available

Of these, the input operands and the rounding mode must be mapped to equivalent inputs in the reference design, and the result and exceptions from the RTL must be compared with the corresponding outputs from the C/C++ reference design to confirm correctness.

Proving Equivalence

Hector is a transaction equivalence checker, which means that it can be used to compare:

- Two unclocked designs (such as two different C/C++ implementations of the same function)
- Two clocked RTL implementations, in which the results from the two RTL designs have different latency
- An unclocked design (implemented in C/C++) and a block of RTL

Because of this flexibility, it is important to understand how you specify what to compare and when to compare it using Hector. The basic unit of time in Hector is the phase, which is half a clock cycle. When you tell Hector to compare outputs, you have to tell it the phase in which the outputs are to be compared. In our example, we are comparing the outputs of a block of C/C++ code with the output of a clocked RTL design.

The C/C++ design produces its output instantaneously (as far as Hector is concerned) whereas the RTL has a 2-cycle lag. If you think of the C/C++ code and the RTL as being clocked, then the C/C++ result is

available in clock cycle 0, but the RTL starts computing at the end of clock 0 (on the rising edge), computes in cycles 1 and 2, and makes the result available in cycle 3.

Hector thinks of clock cycles in terms of phases, where each clock cycle is composed of two phases, those being when the clock signal is high, and then low. This was done to give you the flexibility to verify RTL designs which makes results available on either the rising or the falling edge of the clock cycle.

The sample design works on the rising edge of the clock, so when you set Hector up to do a comparison of outputs, you would tell it to compare the output of the C++ code in phase 1 (the tail-end of clock-cycle 0), with RTL outputs in phase 7 (the tail-end of clock cycle 3). The command script provided as part of the example design shows this.

Hector works by building a model of the RTL and the C++ design in the form of graphs, and then proving that these graphs are equivalent. This is a very hard problem, and there is no known algorithm which can *always* solve this equivalence problem in a reasonable or even unreasonable amount of time. To reconcile this fact with the reality that *you have a schedule to keep*, Hector uses heuristics – rules of thumb, loosely speaking – to effectively solve these equivalence problems much of the time. But the unfortunate flip side of this is that there are problems for which Hector will be unable to converge to a solution. One well-known example of this is multiplication.

When this happens, Hector needs your help in order to proceed. Specifically, it expects you to break down the original (large, intractable) equivalence checking problem into many smaller (and hopefully more tractable) ones: each of these can then be solved by Hector.

There are three techniques available to do this type of decomposition. In increasing order of difficulty and complexity, they are:

- Assume-guarantee
- Case-splitting
- Cascade of proofs

The first two techniques are actually needed to successfully prove multiply-add, but we will also (briefly) discuss the third for the sake of completeness.

Assume-Guarantee

The basic idea of assume-guarantee is that instead of trying to prove an entire proposition all at once, we first prove smaller, simpler partial results. Then we assume that the simpler results are true, and under that assumption, we prove the original result. This makes it easier to prove the original result.

In the specific case of proving multiply-add (or even multiply), this technique is typically applied by first proving that the mantissa multiplication hardware (typically implemented using Booth encoding, Wallace trees and a CSA) performs multiplication, i.e. that it implements the "*" operator.

In the second step, we assume the veracity of that assumption (since we have already proved it), and then prove equivalence of the rest of the design subject to that assumption.

Case-splitting

This just a variation of the case-splitting idea. Consider a situation where you have a block of RTL which takes two 8-bit inputs and produces an 8-bit result. Let us further stipulate that all possible input values

are valid and legal. This means that the number of possible input combinations is equal to $2^8 * 2^8 = 256 * 256 = 65536$. If this task proves intractable for whatever reason, it is reasonable to break the input space into 4 regions (as illustrated in Figure 2), and to verify each region separately. If each such subregion can be verified for correctness, it then follows that the entire design must be correct as well. The key of course is to make sure that the subregions, taken together, actually span the entire original input space. When you do case-splitting with Hector, it automatically generates checks which verify that the case-splits that you have specified actually span the entire input space.

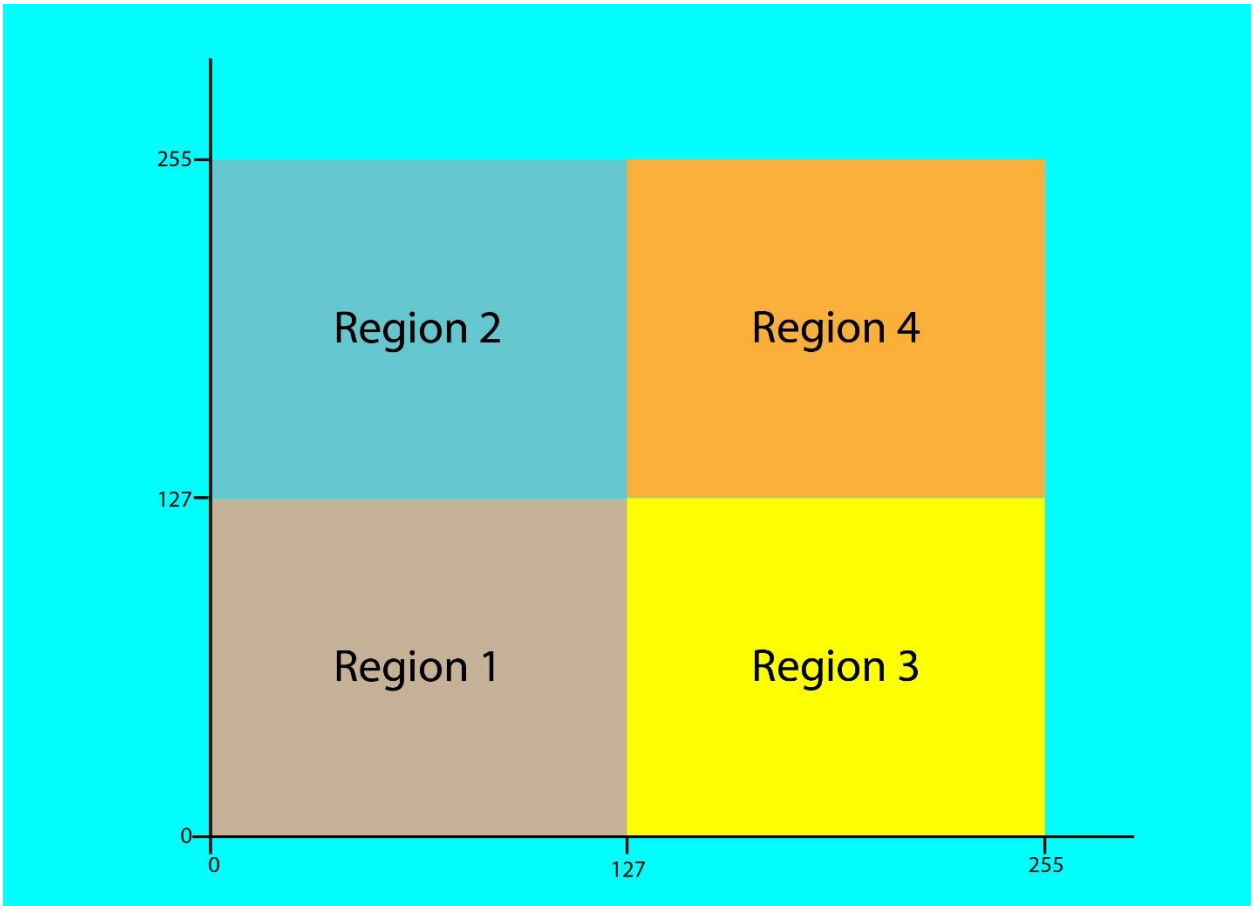

Figure 2: Splitting a large region into subregions

Cascade-of-proofs

Occasionally, the reference and the RTL designs are so different that Hector is unable to find equivalence points between them and is therefore unable to prove equivalence. What can be done in these cases is to build a sequence of RTL implementations. The first of these is very similar to the C++ reference, the second is more different, and so on.

As an example, consider an RTL design "R" which has this issue. Let us say that you write a sequence of RTL designs R1, R2 and R3, such that R1 is similar (in design flow) to the C++ reference, R3 is similar to the production RTL, and R2 is in-between R1 and R3. Now you can set up a sequence of Hector proofs, as follows:

- C++ == R1?

- R1 == R2?
- R2 == R3?
- R3 == Production RTL?

Because these interim RTL designs are not completely dissimilar from each other, it is often possible to successfully prove each of the 4 propositions listed above, and if so, then taken together, they are equivalent to the assertion that C++ == Production RTL.

This technique is not needed to prove multiply-add but you should know about it in case you need it at some point in the future.

Getting a Hector Proof for multiply-add

With this background, let us walk through the strategy used to verify the example multiply-add design. We begin by describing the Tcl procs which are in the Hector command script that we used to verify the example RTL. These procs are as follows (and detailed explanations of the commands in these procs can be found in the Hector User's Guide):

- compile_spec() – this proc compiles the "specification" design (C++) into a Hector model. "cppan" is Hector's C++ analyzer, and the proc contains syntax very similar to what you would use if you were compiling this code with gcc.
- compile_impl() – this is the corresponding proc to compile the RTL into a Hector model. Notice how we are defining a pair of cutpoints (to be activated later) on the inputs to the Booth multiplication module. When you do this with your own design, you will have to find the right place in your RTL at which to insert cutpoints like these.
- make() – this is a convenience proc which invokes the previous procs to compile the two designs, following which it composes the designs into a final model which Hector will use for proofs.
- When you tell Hector to run a proof, you need to specify a "user assumes and lemmas procedure". In this, you define which inputs to the two designs are equivalent to each other, which outputs are to be compared to confirm equivalence, and other pertinent details such as the solvers to use, resource limits, etc. In our example, there are three procs like this (for FP32), the first (ual_result) is used for checking that the output results match, the second (ual_ex) is used for checking that the two designs produce the same exception flags, and the third is used for verifying that the Booth multiplier implements the "*" operator. The proc "hdps_ual" is a working example of how to set Hector up to verify the multiplier.
- Corresponding to the three "user assumes and lemmas procedure" procs, there are 3 procs which actually run the proofs – these are called run_hdps(), run_result() and run_ex().
- case_split_32() – this proc contains a definition of the case splitting that we have done in order to get convergence for the example design.

Hector provides a special solver engine which is optimized for proving multiplication – it is called HDPS. In order to use it, you must insert "cutpoints" into the RTL design at the point where the multiplier and the multiplicand enter the Booth multiplication circuit. Doing this is a two-step process. First, the cutpoint is created – this, in essence, instructs Hector to insert a multiplexer at the location of the cutpoint in the model Hector builds of the RTL. The second step is to actually enable the cutpoint, so

that the point in question, instead of being driven by upstream logic, becomes a “free input” which will be driven by Hector for the purpose of obtaining a proof.

In the Tcl proc hdps_ual, you will see the two “cutpoint” statements: these are creating cutpoints at the inputs to the Booth multiplication module. Note however, that at this point, the cutpoint is merely created, but not enabled. In order to enable it, you must refer to it using the cutpoint name specified in the “cutpoint” command. In this case, the names are “mpier” and “mpcand”. This is what has been done in the lemma specified to check the Booth circuit (in the same Tcl proc).

In the main proof then, the correctness of the Booth mantissa multiplication can now be assumed (since it was proven in the HDPS lemma) – that is why you see the statement:

```
  assume impl.product_mantissa_0(3) == impl.mpier_mantissa_0(3) *
impl.mpcand_mantissa_0(3)
```

. . . in the `ual_result` and `ual_ex` procs. This instructs Hector to abstract away all the logic in the Booth module and replace it with the “*” operator, thereby simplifying the model and significantly increasing the probability that a proof can be obtained in a reasonable period of time.

When checking the main design, it is of course necessary to verify not only that the circuit correctly computes a * b + c correctly, but also that it correctly generates all relevant exceptions defined by the IEEE-754 standard. Note that in order to simplify each individual proof, we have separated the proving of the output result and the proving of exceptions into two separate proofs (the Tcl procs run_result and run_ex).

Each of these procs specifies a case-splitting procedure. For this particular design, we have chosen to split the overall proof into the following sub-proofs:

- Operand A (the multiplier) is NaN, Infinite or 0.
- Operand B (the multiplicand) is NaN, Infinite or 0.
- Operand C (the addend) is NaN, Infinite or 0
- All three operands are normal numbers
- Only operand A is denormal
- Only operand B is denormal
- Only operand C is denormal
- Only A and B are denormal
- Only A and C are denormal
- Only B and C a denormal
- A, B and C are all denormal

There is no particular methodology to this splitting: experience with multiply-add designs has shown this split to be effective.

Note also that within each of these cases we are doing additional case splitting for the cases where the operands are not one of the special values (NaN, Infinity, zero) – again, this particular case splitting is determined experimentally.

Hector has extremely powerful tools to effectively use multiprocessing, and we encourage you to use as much case splitting as necessary to get proofs quickly.