Design Documentation

This document describes the design of the floating-point multiply-add RTL which has been developed for use as an example of both implementing such hardware as well as proving it using HECTOR.

The design is completely parameterized, and is controlled at the top level by a single parameter SIZE, which when set to 16, 32 or 64 will instantiate a FP16, FP32 or FP64 fused multiply-add unit.

There are 3 RTL modules:

- muladda.sv – this is the top-level module which actually performs the multiply-add operation.
- clzmadd.sv – this module is used to count the number of leading zeros in intermediate results
- booth4.sv – this module implements radix-4 multiplication of unsigned mantissas using Booth's algorithm.

All the rounding modes (round-ties-to-even, round-towards-0, round-towards-positive-infinity and round-towards-negative-infinity) and exceptions (invalid, inexact, overflow, underflow and infinity) required by the IEEE-754 standard have been implemented.

There are 3 HECTOR command scripts which are used for proving the RTL for the FP16, FP32 and FP64 cases: these are in the following files:

- command_script_muladd16.tcl
- command_script_muladd32.tcl
- command_script_muladd64.tcl

There are three C++ wrappers around SoftFloat to implement the "spec" design. These are:

- madd16.cc
- madd32.cc
- madd64.cc

The directory structure used for the code is as follows:

<design-top>: top-level directory

<design-top>/rtl: contains all the RTL

<design-top>/cc: contains all the C code

<design-top>/tcl: contains all the Tcl configuration scripts

<design-top>/run: directory to run HECTOR or DPV. Tcl scripts and filelists are referenced from this directory.

As a precursor to implementing fused multiply-add, simple floating-point multiplication was implemented first. This design consists of the following files:

- fmul.sv – the top-level module which implements floating point multiplication (for FP16, FP32 and FP64). This module, like muladda.sv is parameterized, and can be instantiated to perform

16, 32 or 64 bit multiplication by setting the parameter SIZE to 16, 32 or 64 when instantiating the module.

- clz.sv – this module is used to count the number of leading zeros in intermediate results
- booth4a.sv – this module implements radix-4 multiplication of unsigned mantissas using Booth's algorithm. Note that the same "booth4a.sv" module is used for both multiplication and fused multiply-add.

The corresponding HECTOR command scripts are:

- command_script_mul16.tcl
- command_script_mul32.tcl
- command_script_mul64.tcl

. . . and the C++ wrappers around SoftFloat to implement the "spec" design are:

- mul16.cc
- mul32.cc
- mul64.cc

For initial debug of the floating-point multiplication, a simple testbench was created: this is the file tb.sv.

Finally, the files "files_a" and "files_muladd" contain the RTL files required to build the multiply and multiply-add designs respectively: these are provided to HECTOR (in the "compile_impl" proc) with the "-f" switch. Parameterization is accomplished using the -pvalue+<parameter name>=<value> vcs switch.

Command Scripts

Each command script contains the following Tcl procs:

- compile_spec – to compile the "spec" design
- compile_impl – to compile the "impl" design
- make – invokes "compile_spec, compile_impl" and then invokes "compose" to create the final "all.dfg" file which HECTOR will use for the proof itself.
- ual and hdps_ual – these are the user assumes/lemmas procs for main and HDPS proofs respectively.
- case_split_xx – with "xx" = 16, 32 or 64 is the proc which defines the case-splitting for the proof.
- run_main: starts the main proof
- run_hdps: starts the HDPS proof

In all cases, proving the design is a 2-step process. In the first step, cutpoints are used to isolate the mantissa multiplication, and this is proved using HDPS. Then the main design is proven under the assumption that the Booth multiplication module implements the "*" operator.

Case splitting was necessary to prove all the designs (multiply and multiply-add) in all 3 sizes. For proving 64-bit multiply and 32/64-bit multiply-add, my disk quota was insufficient to launch all the case splits simultaneously: attempting to do so caused a HECTOR crash. Consequently, these designs had to be proved using the following strategy:

- Disable case splits and run the main proof to show that the case-splits are indeed complete (i.e. the automatically generated “completeness” proofs are successful.
- Uncomment segments of the case split procedure and prove the cases in parts and prove each part separately. This is why you will see comment characters in some of the lines in the command scripts. In these partial proof runs, the completeness proof would of course fail, but since it was separately proved earlier, that is OK.

RTL

This section describes the design details of the RTL modules. Please refer to Figure 1 for a block diagram of the design.

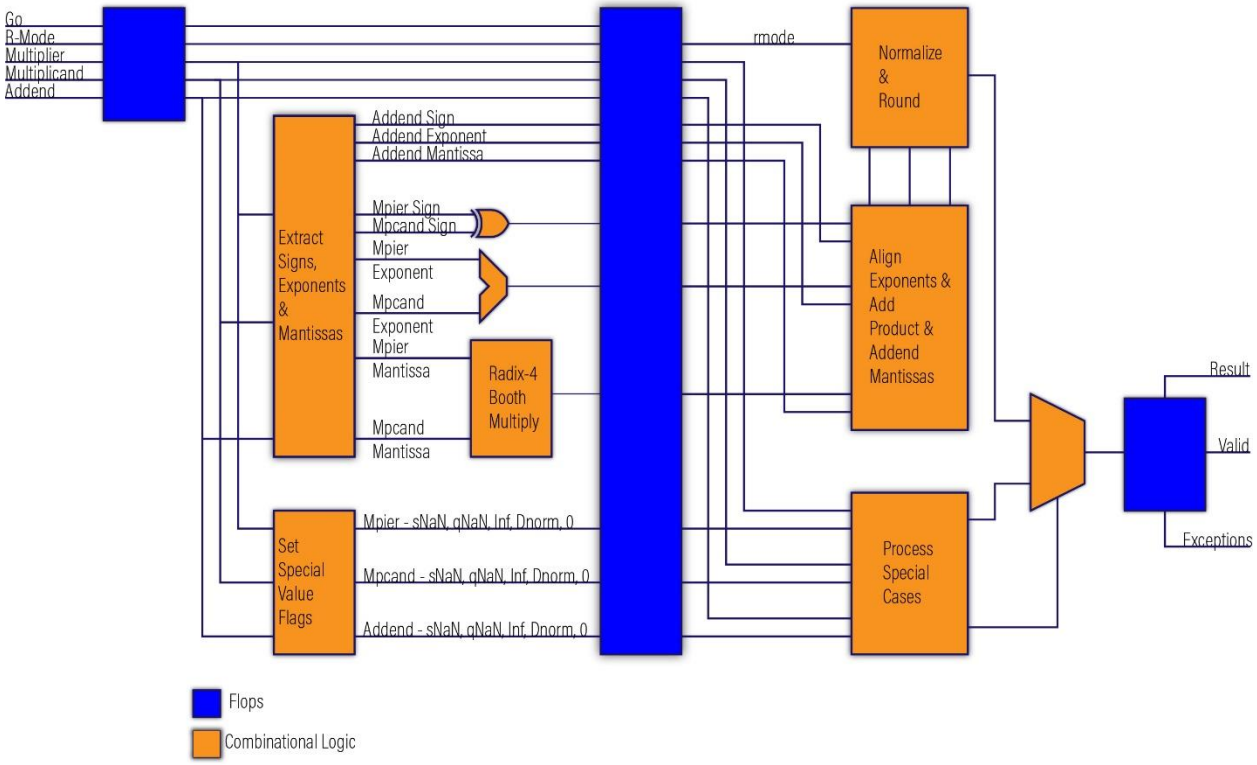

Figure 1: Multiply-Add Block Diagram

clz.sv/clzmadd.sv

These modules count leading zeros. Initially, clz.sv was implemented for multiplication, and subsequently, a more general form of it was implemented for multiply-add – the latter form allows adjusting the leading-zero count via a parameterized offset, whereas this is hard-coded in clz.sv. The reason an offset is required in the first place is that the product mantissa can have some leading zeroes because of multiplying denormal numbers, but there can also be leading zeroes when multiplying normal numbers, and these need to be ignored.

booth4a.sv

This module implements radix-4 unsigned multiplication using Booth’s algorithm. Unlike a “tapeout-worthy” design (which would use a Wallace tree of compressors followed by a CSA to do the final

addition) this implementation uses full adders. This is all that is necessary to demonstrate the use of HDPS to prove multiply and multiply-add in 2 steps, using the "assume-guarantee" technique.

muladda.sv

This is the main module for multiply-add. It implements fused multiply-add in a "school-book" fashion, with an emphasis on readability and ease of implementation. The module is instantiated with a single parameter (which should be set to 16, 32 or 64) to create a 16, 32 or 64-bit fused multiply-add. Based on this parameter, the local function "convert" is used to generate a set of local parameters which are used to index various parts of the N-bit floating point number (e.g. mantissa, sign, exponent, the bit positions for signaling and quiet NaNs, the value of the bias used in the exponent, and the positions of the Guard, Round and Sticky bits used during rounding).

As a general rule, *everything* is parameterized and these values are derived from the single instantiation parameter SIZE.

The RTL takes 7 inputs: clock, resetN, multiplier, multiplicand, addend, rounding_mode and a "go" signal. This last is what triggers the RTL to start a computation.

The design is a "flow-through" 2-stage pipeline.

In the first pipeline stage, the following tasks are performed:

- Disassemble all three input operands into their corresponding signs, exponents and mantissas. During this disassembly, a "1" or a "0" is appended to left-end of the mantissa depending on whether the number is normal or denormal.
- Note that exponents are carried all the way through without removing the bias value – this was found to be simpler to implement because it allowed the use of unsigned arithmetic throughout the computation. Also, multiplication implies adding the exponents of the multiplier and multiplicand, resulting in an extra BIAS value. To match this, an additional BIAS term is added to the addend's exponent when transitioning from stage 0 to stage 1 of the pipeline (this is done in the "always_ff" block at the tail-end of the code.
- Instantiate the clzmadd module to multiply the mantissas of the multiplier and multiplicand.
- Analyze each of the 3 operands and set up a group of 5 bits to indicate whether each operand is SNaN, QNaN, NaN (= SNaN | QNaN, for convenience), INF, DNORM or ZERO. These are used in the next stage of the pipeline.
- Note that the product mantissa can be as large as 11.1111111. . . . . (i.e. < 4) whereas the addend mantissa can only be as large as 1.11111. . . (i.e. <2). Consequently, when passing these mantissas to the next stage, the addend's mantissa is shifted right by 1 bit to ensure that the radix points are aligned.

In the second pipeline stage, the following tasks are performed:

- Deal with all special cases (of which there is a very large number) where one or more of the operands is 0, infinite or not-a-number. If any of these special cases is detected, a result is immediately generated (by quieting an input SNaN or forcing the output to a standard value) and any corresponding exception flags are set.
- If there are no special cases, then we proceed to the actual arithmetic.

- Since the mantissa product is already available from the previous pipe stage, the first step is to compare the two exponents and shift one or the other mantissa (product or addend) to the right to make the exponents match up.
- Then, depending on the signs of the product and the addend, the mantissas are either summed or differenced, and the sign of the result is set appropriately.
- Now, the result mantissa can be as large as 101.111111. . . . so the first task is to check if there is more than 1 bit to the left of the radix point and if not, the mantissa is shifted right by 1 or 2 places to make sure there is only a single "1" to the left of the radix point.
- Then the number of leading zeroes is counted.
- Based on the leading-zero count and the value of the exponent (keeping in mind that at this point, the exponent includes an offset of TWO times the BIAS), there are 4 cases:
    - The exponent is so large that no amount of mantissa shifting can make the exponent small enough to fit into the number of bits available (5, 8 or 11 depending on FP16, FP32 or FP64). In that case, an overflow is generated.
    - The exponent satisfies the Goldilocks criterion (not too big, not too small) so that either there is already a "1" left of the radix point, or can be made to appear there by shifting the mantissa left (and decrementing the exponent accordingly).
    - The exponent is too small, but can be made to fit by either shifting the mantissa left (but not by enough to put a "1" left of the radix point) or further right in order to make sure the exponent will fit into the available number of bits.

  Throughout this process, we keep track of any bits which fall into the "bit bucket" owing to a right shift, and collect these in a "sticky bit" which is used later for rounding.

Following this, rounding as dictated by the rounding_mode input is applied. It is possible that a denormal number will turn into a normal number as a result of rounding. We check for this after rounding and adjust the exponent if necessary.
Finally, we subtract the extra BIAS which we have carried along from the exponent, and assemble the final result.

fmul.sv

This RTL module implements floating-point multiplication and follows the same sequence of steps as the multiply-add module except (of course) the second pipeline stage, which is different because there is no addend term. Rather, the multiplication proceeds as follows:

- Check if there are 2 bits to the left of the radix point, and shift right if so (and adjust the exponent up).
- Count the number of leading zeroes.
- Based on the leading-zero count and the value of the product exponent, shift the mantissa and adjust the exponent to either:
    - Create a normal number
    - An overflow
    - A denormal number which may have a zero to the left of the radix point, possibly more zeroes to the right of the radix point but an exponent which can fit into the available number of bits.
- Perform rounding

- Assemble the result