Fused Multiply-Add

Michele Tarocco

Contents

1 Introduction **1**

2 Design implementation **1**
2.1 Inputs 2
2.2 Implementation 2
2.3 Outputs 3

3 Results **4**
3.1 Performances 4

1 Introduction

The objective of this project is to realize a Fused Multiply-Add module.

$$R = \pm(A \times B) \pm C$$

The module has to perform a multiply and a add as one single and indivisible operation. This operation can be useful in the evaluation of a polynomial $a_n x_n + a_{n-1} x_{n-1} + \cdots a_0$ through $[(a_n x + a_{n-1})\, x + a_{n-2}]\, x \cdots$, by using several FMA operations.
Another advantage arises in floating point operations since rounding is performed only once.

2 Design implementation

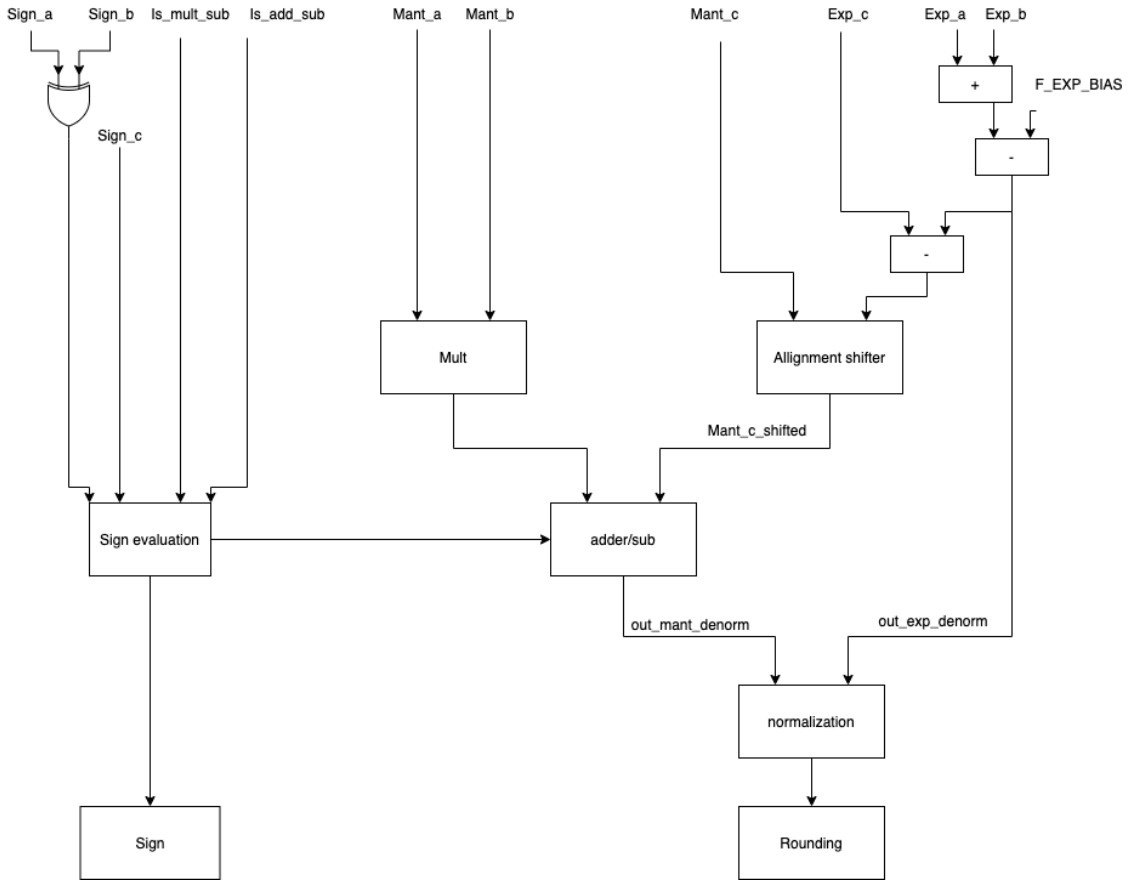

Figure 1: FMA unit

The design of the module is shown in Figure 1. To be precise the design works in two steps: in the first result is evaluated and then stored into a register. Also other useful variables for normalization are stored in apposite registers.
Second step is normalization: the result stored is taken and is normalized.

2.1 Inputs

The module has an input clock(*clk*) and a synchronous active high reset(*rst*) .
Module has several inputs: it takes for each operand his sign, mantissas and exponent: *sign_op(x)_i*, *mant_op(x)_i[(7-1)+1:0]* one extra bit for the hidden bit and *exp_op(x)_i[(8-1)+1:0]* one extra bit for overflow check.
Each operand has also three others input to identify if the operand is not a number , *isSNAN_op(x)_i* and *isQNAN_op(x)_i*, if the input operand is infinite, *isInf_op(x)_i*.
Operand A and B since are involved in the multiplication have two more input to check if operands are zero *isZ_op(x)_i*.
One further input to enable the module *doFMU_i* and two other to define the operation in module, check below the operation table.

is_mult_sub_i	is_add_sub_i	Operation
0	0	FMADD
0	1	FMSUB
1	0	FNMSUB
1	1	FNMADD

2.2 Implementation

The design of the module is not simply a fusion of a multiplication and an addition/subtraction block. Usually to perform an addition/subtraction we evaluate the bigger operand, we increase the smaller' s exponent in order to match the bigger one and then we shift the smaller' s mantissa to the right in order to obtain a proper alignment between operands.
Using this strategy in this module will increase the delay, since we've to wait the result of the multiplication between A and B mantissas, which could require several clock cycles .
Proposed solution is to shift C operand anyway, both if is bigger or smaller than the product and consequently increase or decrease C exponent, allowing both a right shift or a left shift. This solution requires an increment in the size of our operand, as shown in Figure 2.
In this way the alignment of C operand is performed in parallel with the multiplication.

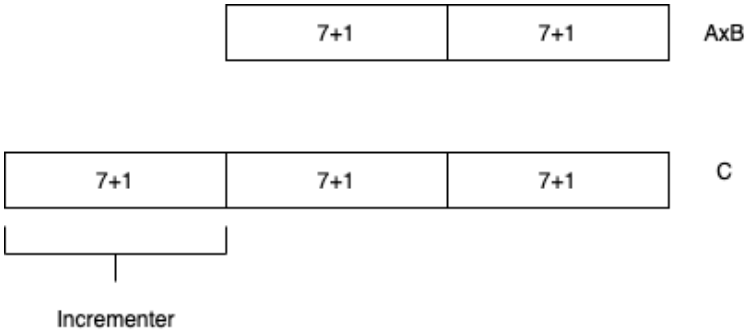

Figure 2: Bit alignment

Exponent resulting from the multiplication is compared to operand C exponent, in order to define if we need to shift C's exponent to the left or to the right. Mantissas are checked only if exponents are equal: this operation is useful to define which operand is the greatest and this becomes useful in the final sign evaluation.
In this way we can save some time, since only in few cases we need to wait multiplication result and only in the case no shift is needed.
A check is done to verify if $\delta = Exp\_a + Exp\_b - Exp\_c - Exp\_Bias \leq -7$, in this case operand C is much bigger than the multiplication result and cannot be shifted to the left otherwise it's value would have been lost. No further operation is required so the result is set equal to C operand.
After alignment result is evaluated with the output sign.
Then normalization begins: first are evaluated left leading zeroes to check if a right shift is required, if not are evaluated right leading zeroes to check if is required a left shift instead.
Several checks are done to verify if an underflow or an overflow has occurred during normalization.

2.3 Outputs

The result of the multiplication between operands A and B requires only $2 \times (7+1) = 16$ bits but, due to the extension of the operand C, the addition/subtraction results requires $3 \times (7+1) = 24$ bits.
Post-normalization output requires the classic 8 bits mantissa and 3 further bits for guard,round and sticky bits. So the 15th bit of the extended normalized result corresponds to the hidden bit, 7th is the guard bit, 6th is the round one and 5th is the sticky bit, as shown in Figure 2.

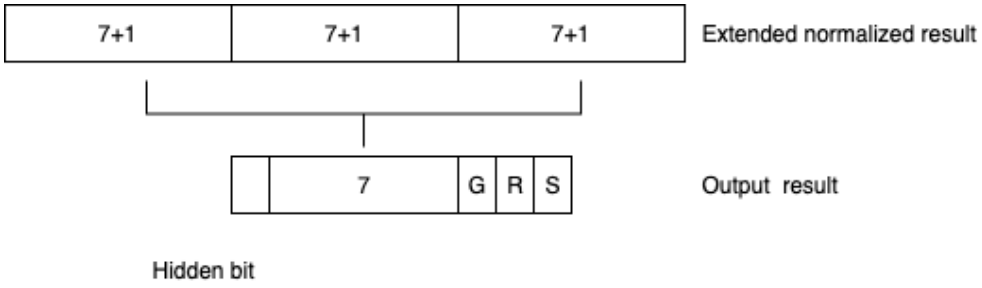

Figure 3: Output result

The module outputs are the resulting sign *res_sign_o*, mantissa *res_mant_o[(7-1)+2+3:0]* and exponent *res_exp_o[8-1:0]*.
There's a further bit to check if an overflow occurred, *is_Overflow_o*, and one for underflow,*is_Underflow_o*.
Also a validity bit is defined as an output,*valid_o*, and last output is *is_to_round_o* which defines if a rounding is needed.

3 Results

Module has been integrated with already existent FPU module.
In order to test module results, DPI library were used.
Results show a match between module and DPI results for almost every case.
This mismatches are presumably due to the fact that testbench uses the already implemented DPI libraries, so , as example, to perform a FMADD is called the multiplication function from the DPI and then the addition function.
A mismatch between results is shown when the multiplication result overflow, in this case the module described is report correctly identifies the overflow, instead the result of the DPI multiplication is then send to the addition causing a wrong evaluation of the result.
A second mismatches occurs in just few cases: results will differ from one bit. This is most reasonably due to the fact that the DPI module performs rounding two times: one after multiplication and one after addition/subtraction.
Performing by hand the rounding will show that the module implemented gives the right result.

3.1 Performances

Overall area occupation is shown in Figure 4.

Name	Slice LUTs (134600)	Slice Registers (269200)	F7 Muxes (67300)	DSPs (740)	Bonded IOB (285)	BUFGCTRL (32)
N lampFPU_top	1415	466	4	2	107	2
lampFPU_addsub (lampFPU_addsub)	185	57	0	0	0	0
lampFPU_cmp0 (lampFPU_cmp)	38	2	0	0	0	0
lampFPU_div0 (lampFPU_div)	225	80	0	2	0	0
lampFPU_f2i0 (lampFPU_f2i)	139	36	0	0	0	0
lampFPU_FMA0 (lampFPU_FMA)	320	67	4	0	0	0
lampFPU_i2f0 (lampFPU_i2f)	182	21	0	0	0	0
lampFPU_mul0 (lampFPU_mul)	159	22	0	0	0	0

Figure 4: Area Occupation

By observing simulation wave-forms we can see that 3,5 clock cycles are required to obtain a valid result, as shown in Figure 5.

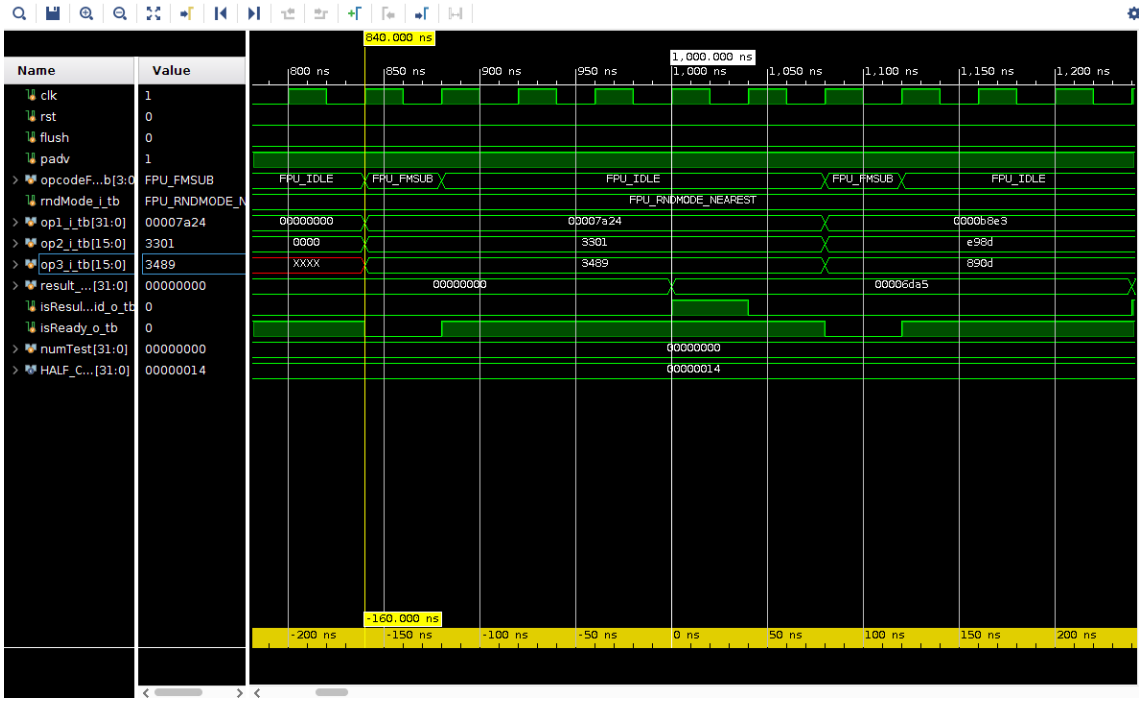

Figure 5: Wave-forms

In the end a timing analysis from the synthesis is shown in Figure 6.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	1137	Total Number of Endpoints:	1137	Total Number of Endpoints:	NA

There are no user specified timing constraints.

Figure 6: Report Timing Summary