Integrated Systems Architecture
Lab2 Report

Marco Andorno
Michele Caon
Matteo Perotti 251453
Giuseppe Sarda

November 11, 2018

1 MBE based Multiplier with Roorda's approach and Dadda Tree

Let's suppose to have a multiplication to be done between two numbers **x** and **y**.

- **x** is the multiplicand
- **y** is the multiplier
- $\boldsymbol{k_x}$ is the parallelism of **x**
- $\boldsymbol{k_x^I}$ is the number of bits representing the **I**nteger part of **x**
- $\boldsymbol{k_x^F}$ is the number of bits representing the **F**ractional part of **x**
- $\boldsymbol{k_y}$ is the parallelism of **y**
- $\boldsymbol{k_y^I}$ is the number of bits representing the **I**nteger part of **y**
- $\boldsymbol{k_y^F}$ is the number of bits representing the **F**ractional part of **y**

We want to perform the multiplication with the **MBE-radix4** encoded version of the multiplier **y**. This shrewdness allow us to reduce the number of partial products by half: indeed **y** is encoded with $\boldsymbol{k_y'}$ symbols in $\{\pm 2, \pm 1, 0\}$.

$$\boldsymbol{k_y'} = \lceil \frac{\boldsymbol{k_y}}{\boldsymbol{log_2(r)}} \rceil = \lceil \frac{\boldsymbol{k_y}}{\boldsymbol{2}} \rceil \tag{1}$$

It's possible to MBE-encode a number in radix4 simply taking $\lceil \frac{k_y}{2} \rceil$ 1-bit overlapping triplets of it. If we consider **y** represented as a sequence of bits $\boldsymbol{y_{(k_y-1)} y_{(k_y-2)} \dots y_1 y_0}$ with the **LSB** in position **0**, then for correctly encoding **y** we must add a $\boldsymbol{y_{-1}}$ bit fixed at **0** to complete the first triplet. If $\boldsymbol{k_y}$ is odd then it will be added a bit $\boldsymbol{y_{k_y}}$ to complete also the last one.
The encoded multiplier is then represented by the string of symbols

$$Y_{(\lceil \frac{k_y}{2} \rceil - 1)} Y_{(\lceil \frac{k_y}{2} \rceil - 2)} \dots Y_1 Y_0 \tag{2}$$

chosen from the set $\{\pm 2, \pm 1, 0\}$ wrt the following table The product is now between **x** and **Y** the MBE-radix4 encoded version of **y**. Each partial product between a symbol of **Y** and **x** is performed using a multiplexer: two of the three bits which encode a symbol are used as control lines for the mux which can let pass either **0**, or **x**, or **2x**. The other encoding bit is asserted only if the symbol is negative and it is used to complement the partial product. Moreover it will be added to the LSB of its partial product, to ensure a correct 2's complement negation. This way it's easy to obtain all the possible partial product: **0**, **x**, **-x**, **-2x** and **2x**.

Since the entire operation has to last one clock cycle all the partial products are obtained in parallel by the same number of encoding circuits and multiplexers. The derived tree is thought as

$y_{n+1}y_ny_{n-1}$	Y_n
000	0
001	1
010	1
011	2
100	-2
101	-1
110	-1
111	0

a Dadda Tree and the number of FA is reduced simplifying the extended sign bits as proposed in [1]. We have $\lceil \frac{k_y}{2} \rceil$ partial products to be compressed to only two terms with a Dadda Tree of CSA. The situation is the following: Since we are not working with full precision, we can do not consider the first $\boldsymbol{k_y^I}$ MSBs, because they do not impact on the others. The $\boldsymbol{k_y^F}$ LSBs are on the contrary fundamental because of the carry of the sums in which they are involved. As Roorda highlighted, the bits of sign-extension can be thought as a series of **1** if the complement of the sign bit is added in its original position. This leads to a further optimization, because each column of **1**, starting from the rightmost one, can be simplified in advance knowing that

$$\begin{cases} ? & 1 \\ ? & 1 \end{cases} \implies \begin{cases} ? & 0 \\ ? & 0 \\ 1 & 0 \end{cases} \tag{3}$$

and

$$\begin{cases} ? & 1 \\ ? & a \end{cases} \implies \begin{cases} ? & \bar{a} \\ ? & 0 \\ a & 0 \end{cases} \tag{4}$$

At the end we have the first row in which there is a string of "$\mathbf{10}$", followed by a sequence of "$\mathbf{1}\ \bar{\boldsymbol{p}}^{\boldsymbol{i}}_{\boldsymbol{k_x+1}}$" and then a triplet composed of "$\bar{\boldsymbol{p}}^{\boldsymbol{0}}_{\boldsymbol{k_x+1}}\boldsymbol{p}^{\boldsymbol{0}}_{\boldsymbol{k_x+1}}\boldsymbol{p}^{\boldsymbol{0}}_{\boldsymbol{k_x+1}}$". On the second row, under the first element $\bar{\boldsymbol{p}}^{\boldsymbol{0}}_{\boldsymbol{k_x+1}}$ of this triplet, there is $\bar{\boldsymbol{p}}^{\boldsymbol{1}}_{\boldsymbol{k_x+1}}$.

Design of the multiplier In our design we have

- $\boldsymbol{k_x = k_y = 24}$
- $\boldsymbol{k_x^I = k_y^I = 2}$
- $\boldsymbol{k_x^F = k_y^I = 22}$
- **12** partial products
- at most $\frac{\boldsymbol{k_y}}{\boldsymbol{2}} \boldsymbol{+ 1 = 13}$ elements in a single column

In a single column we can count up to 13 elements, because the MUX let pass only $\mathbf{x}$ multiplied for the absolute value of the symbol $\mathbf{Y}$. The "negative" bit (one of the three which encode a single symbol) it has to be added to the LSB of its partial product: therefore we have 12+1 elements at most in a column. This is not so bad, because with the Dadda Tree we are still in the case of having only 5 levels of FA (13 elements in a column at most).

References

[1] Roorda, "Method to reduce the sign bit extension in a multiplier that uses the modified booth algorithm," 1986.