Integrated Systems Architecture
Lab session 2 Report

Marco Andorno (247222)
Michele Caon
Matteo Perotti (251453)
Giuseppe Sarda

November 14, 2018

1 Comparison between different combinations of adders and multipliers

Starting from the pipelined and retimed architecture developed in Lab 1, we exploited Synopsys DesignWare Library to compare several design metrics using different implementations for the arithmetic operators used in our design.

Among others, DesignWare makes available the following synthesis models of adders and multipliers:

- `DW01_add` [1]:
  - `rpl`: ripple-carry adder
  - `cla`: carry-lookahead adder
  - `pparch`: delay optimized flexible parallel-prefix adder
- `DW02_mult`:
  - `csa`: carry-save array multiplier
  - `pparch`: delay optimized flexible Booth-recoded Wallace tree multiplier

The approach used is totally similar to the one suggested for the first laboratory experience:

- Synthesize design using Design Compiler, setting the clock period do 0 in order to find the maximum frequency f_{max}.
- Synthesize again using a clock period corresponding to $f_{max}/4$ and find out the area of the design.
- Simulate on a sufficient number of samples (1000 was the sweet spot to trade off accuracy and simulation time) in ModelSim to extract the switching activity.
- Run Design Compiler once again to carry out the power estimation.

1.1 Maximum frequency

Table 1 show the maximum frequency achieved for each pair of implementations. Red and green cells highlight respectively the lowest and highest maximum frequencies. Note that the best frequency achieved is the same that Synopsys reached by choosing the implementations on its own in the previous laboratory experience (in fact, the resources report of that base version shows exactly that `pparch` was chosen as the architecture of both the adders and the multipliers).

[1] As the feedback branch of the filter gets subtracted from the input in direct form II, Synopsys actually uses also a subtractor (`DW01_sub`), for which the same implementations as the adder are available. So when we say that we used a certain kind of adder, we mean that we used it for all adders and for that subtractor too.

f_{max} [MHz]		Adder		
		rpl	cla	pparch
Multiplier	csa	285.7	287.4	285.7
	pparch	431.0	684.9	689.7

Table 1: Maximum frequency

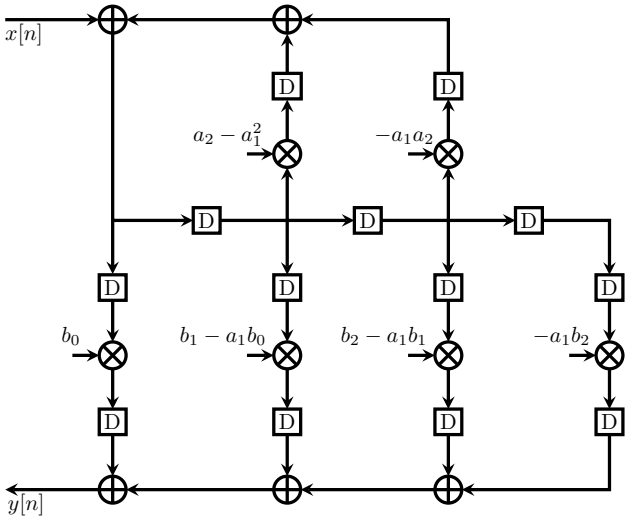

Figure 1: DFG of the filter

Figure 1 shows the DFG of the final architecture achieved at the end of the previous lab, by looking at which, one can quite positively state that the critical path is the delay of a single multiplier T_m, as it is almost certainly larger than the delay of three adders $3T_a$. So one would expect the maximum frequency to remain constant as long as the multiplier used is the same and only the implementation of the adder is changed.

In fact, this is what happens in five cases out of six in table 1: when using the carry-save based multiplier, the clock frequency is always limited by its delay, irrespective of the adder used. But when using the faster multiplier (`pparch`), the slowest ripple-carry adder seems to be the limiting factor, as the maximum frequency it allows is much lower than the one achieved with the same multiplier but faster adders.

So in this case the critical path seems to be the series of three adders, instead of the single multiplier. To verify this hypothesis, the DFG of the filter was slightly modified by introducing an additional pipeline stage to split the chain of 3 adders into $2 + 1$ adders, as there is a feedforward cutset that allows it. By repeating the synthesis flow using the ripple-carry adder and the parallel prefix multiplier, the maximum frequency increased by around 10%, from 431.0 MHz to 473.9 MHz. Even if this is not a 33% improvement (probably due to design optimizations operated by the tool), as conceptually expected, it can nevertheless be concluded that the chain of three 24-bit ripple-carry adders was in the end the bottleneck of the design, being slower than one single 24-bit parallel prefix multiplier.

All in all, choosing the parallel-prefix multiplier combined with either carry-loohahead or parallel-prefix adders is the best option for speed: the maximum clock frequency is 60% higher than the pair `pparch` multiplier/`rpl` adder, and 242% faster than all carry-save multiplier based implementations.

1.2 Cell area

For what concerns area, table 2 shows that this figure seems to be once again mostly a matter of the type of multiplier chosen. Choosing the parallel-prefix multiplier is again the winning choice, as such implementations have a total area 38% smaller than the one of designs based on a carry-save multiplier.

In the case of area, the choice of the type of adder does not seem to influence much the final figure, showing variations of less than 1% among the three implementation, given the multiplier type.

Area [µm^2]		Adder		
		rpl	cla	pparch
Multiplier	csa	20987.4	21113.5	20978.1
	pparch	15282.8	15408.8	15273.5

Table 2: Total area

1.3 Total power dissipation

Power dissipation is clearly where the advantage of the parallel-prefix multiplier falls short. All implementations based on that show a 32-fold increase in dynamic power and a 14-fold increase in leakage power.

As always in digital design, as well as in life, one cannot get the best of both worlds, but has to settle for a trade-off, in this case between speed and power, as usual.

Power [µW]		Adder		
		rpl	cla	pparch
Multiplier	csa	120.1 3.1	120.1 3.1	120.1 3.1
	pparch	3815 43.5	3816 43.5	3816 43.5

Table 3: Dynamic and leakage power

1.4 Summing up

If the main requirement is pure speed, then an architecture based on parallel-prefix Booth-recoded Wallace tree multipliers and carry-lookahead or parallel-prefix adders is the way to go.

If, on the other hand, the power budget is a limiting factor, one has to settle for lower processing speed and use CSA-based multipliers along with whichever adder they like best.

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

Let's suppose to have a multiplication to be done between two numbers $\mathbf{x}$ and $\mathbf{y}$.

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

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

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 $\boldsymbol{y_{(k_y-1)}y_{(k_y-2)}\cdots y_1y_0}$ 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 a Dadda Tree and the number of FA is reduced simplifying the extended sign bits as proposed in [**?**]. 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 "**10**", followed by a sequence of "$\boldsymbol{1\ \bar{p}^i_{k_x+1}}$" and then a triplet composed of "$\boldsymbol{\bar{p}^0_{k_x+1}p^0_{k_x+1}p^0_{k_x+1}}$". On the second row, under the first element $\boldsymbol{\bar{p}^0_{k_x+1}}$ of this triplet, there is $\boldsymbol{\bar{p}^1_{k_x+1}}$.

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 **x** multiplied for the absolute value of the symbol **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).