Integrated Systems Architecture
Lab session 2 Report

Marco Andorno (247222)
Michele Caon
Matteo Perotti (251453)
Giuseppe Sarda (255648)

March 19, 2019

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

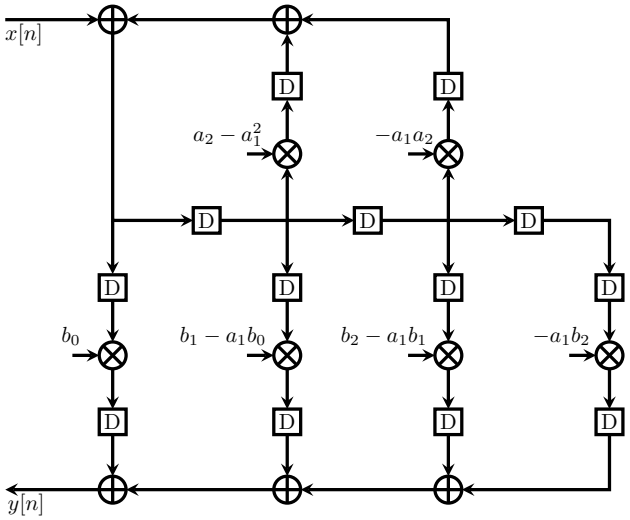

Figure 1: DFG of the filter

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 $\mathbf{y}$

We want to perform the multiplication with the **MBE-radix4** encoded version of the multiplier **y**. This shrewdness allows us to reduce the number of partial products by half: indeed **y** is encoded with $\boldsymbol{k_y'}$ symbols in $\{\pm 2, \pm 1, 0\}$. If $r = 4$ is the radix:

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

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 $\boldsymbol{y_{(k_y-1)} y_{(k_y-2)} \cdots y_1 y_0}$ 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 table 4.

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

Table 4: MBE conversion 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**. It follows a figure (2) representing the DP of the multiplier.

Since the entire operation has to last one clock cycle, all the partial products are obtained in parallel by k_y' 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 FA/HA. Since with the multipliers in the filter we are not working with full dynamics, we can do not consider the first $\boldsymbol{k_y^I}$ MSBs, because they do not impact on the others. A saving in power and area is achieved this way. The number of integer bits of the lines of the filter was chosen not to let overflow occur anywhere in the filter; thus the number of integer bits is high enough and doesn't need to be increased. On the contrary the $\boldsymbol{k_y^F}$ LSBs are fundamental because of the carry of the sums in which they are involved, which can impact over the bits on their left. (More on this later)

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 noting that

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

and

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

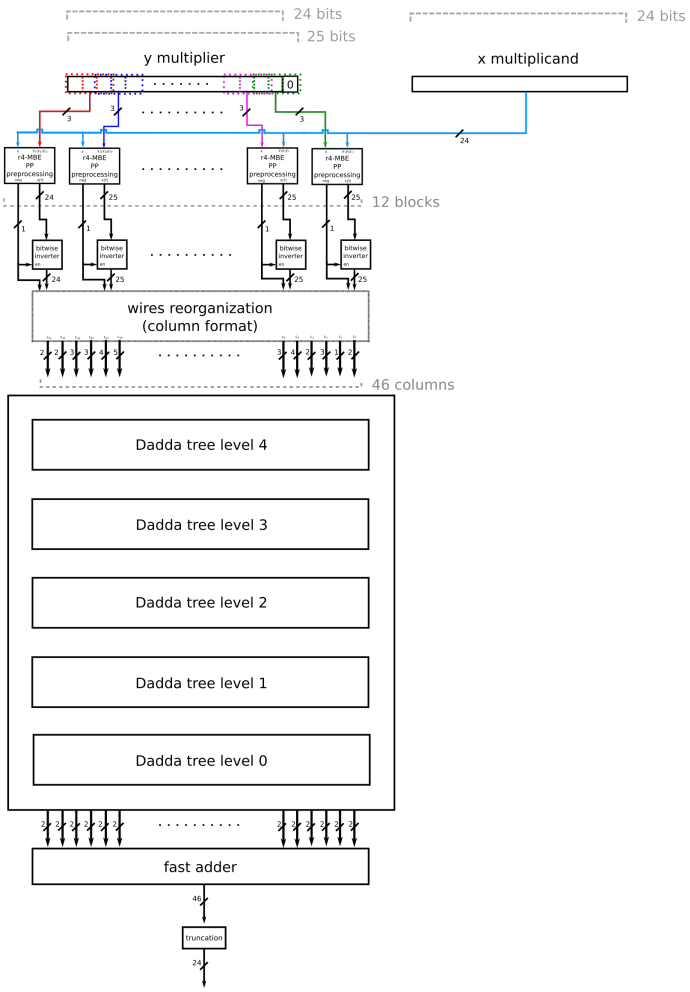

Figure 2: DP of the filter multiplier based on a DADDA tree, with Roorda's approach on MBE-r4 encoding

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

Design of the multiplier In our design we have

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

In a single column we can count up to 13 elements, because the MUX let only pass **x** multiplied for the absolute value of the symbol **Y**. Since this partial product passes only through a conditional negation block, to obtain a real C2 negation for a final partial product there is the need for a LSB to be summed. This "negative" bit (one of the three which encode a single symbol) is added to the LSB of its partial product: therefore we have 12+1 elements at most in a column (this can be seen in figure 3). This is not so bad, because with the Dadda Tree we are still in the case of having only 5 levels of adders (13 elements in a column at most).

As said before, the sign extension bits can be reduced to chains of '1', and these chains can be "read" in column to be further simplified; the two MSB columns can be cut away because it's known that those columns will be only sign extension at the end of the multiplication. This is a reflection of what it was said before: in the filter architecture in which the multiplier is used the parallelism is $k = 24$ (Q2.22) for each words, but the coefficients are known and less than one[2], so the integer part of the product is always representable with 2 bits. the parallelism of 24

The effect of this entire process is summarized in 3. The red dots are the "negative" bits to complete the C2 negation, white dots are the extended sign bits, each blue dot represent the negated sign bit of a partial product. The light blue dots are actually white ones, because they represent the extended sign bits of a partial product.

The dashed red vertical line extrudes the two MSBs from the bindings of the adders.

For the first design the binding has been done by hand for each level of the tree. The following schematics show where FAs and HAs have been put.

The last operation in an addition, and can be implemented by a fast adder. In our design the adder has been kept behavioural, because the main aim of this work is to study a multiplier and how the uncertainty of the results changes with the adoption of some tricks to improve the figures of merit without keeping full precision.

VHDL description A new method to write VHDL has been adopted to speed up the description of the circuit. The first part (common to all design) in which the partial product are computed was described by hand, but the part of the Dadda trees and the bindings are written by a parametric python script. This allows a great re-usability and a fast method to compare multiple instances with different styles of binding. Furthermore this way of working is not prone to errors, because the number of lines of hand written code is drastically reduced. Another advantage of the script is that the VHDL description can scale up well with the number of bits.

The strategy to write the VHDL is the following: for each DADDA level a matrix of *std_logic_vector* is created. From one level to the next only the useful bits are assigned or processed (all the assignments are bitwise assignments from one matrix to the next).

Adders are instantiated where they are needed, and the port map follows the same "one matrix to the next" strategy used for the assignments.

To implement this design the first hand written part of the file prepares the first matrix (in the following it will be used the term "grid") called **grid5**

At first the partial product are assigned to the matrix cells with standard logic vector assignments, to reproduce the first scheme of 3. Then all the dots are flattened to the top of the grid

[2]Be aware: this design is only valid under this assumption. Otherwise all the MSBs have to be taken into account.

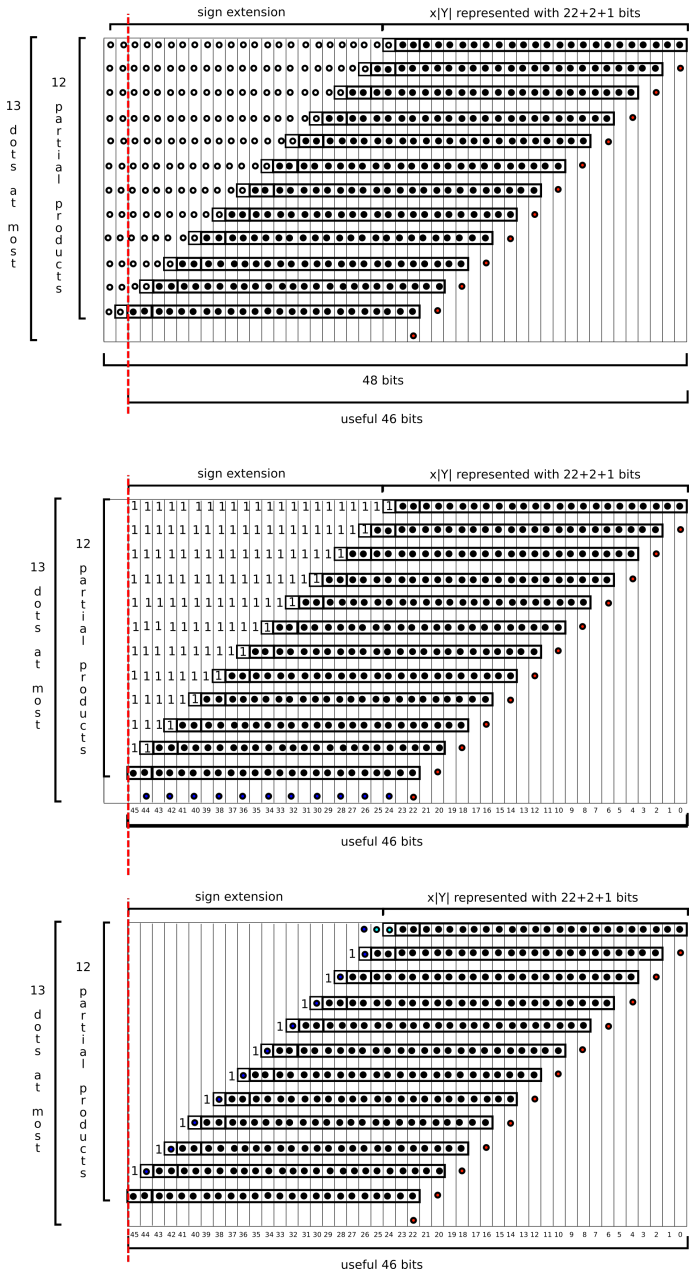

Figure 3: Applying the Roorda's method to reduce the numbers of adders

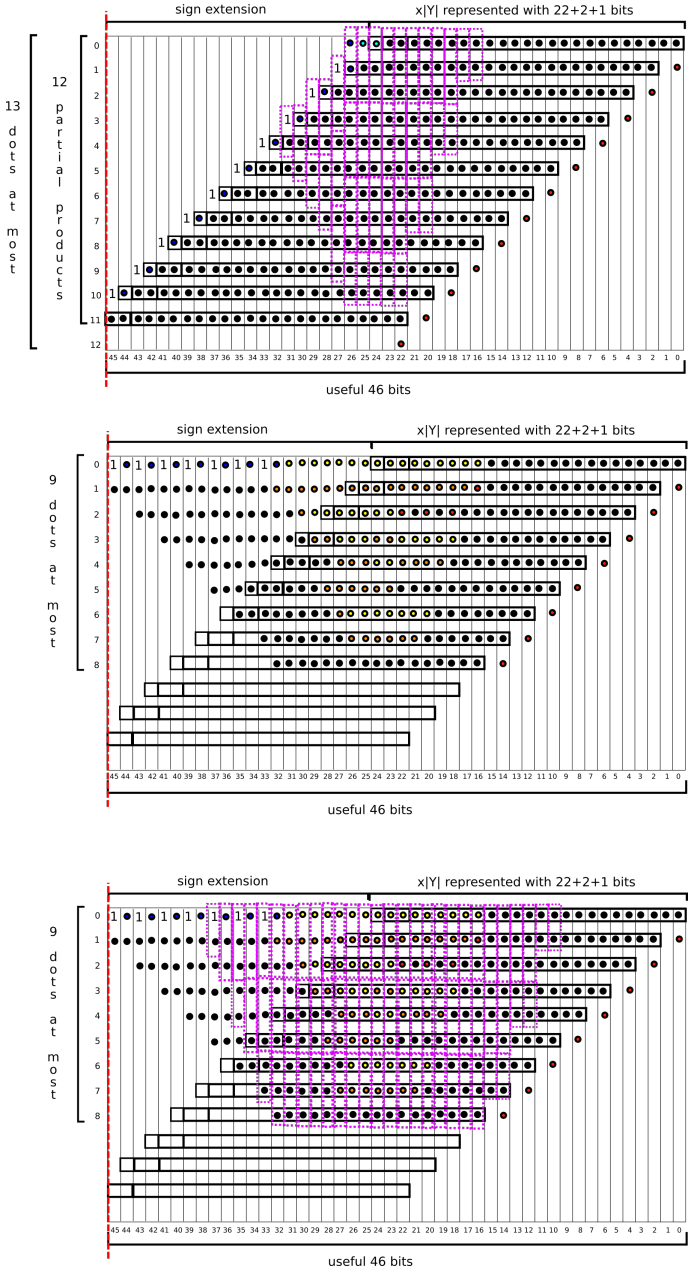

Figure 4: Applying the Roorda's method to reduce the numbers of adders

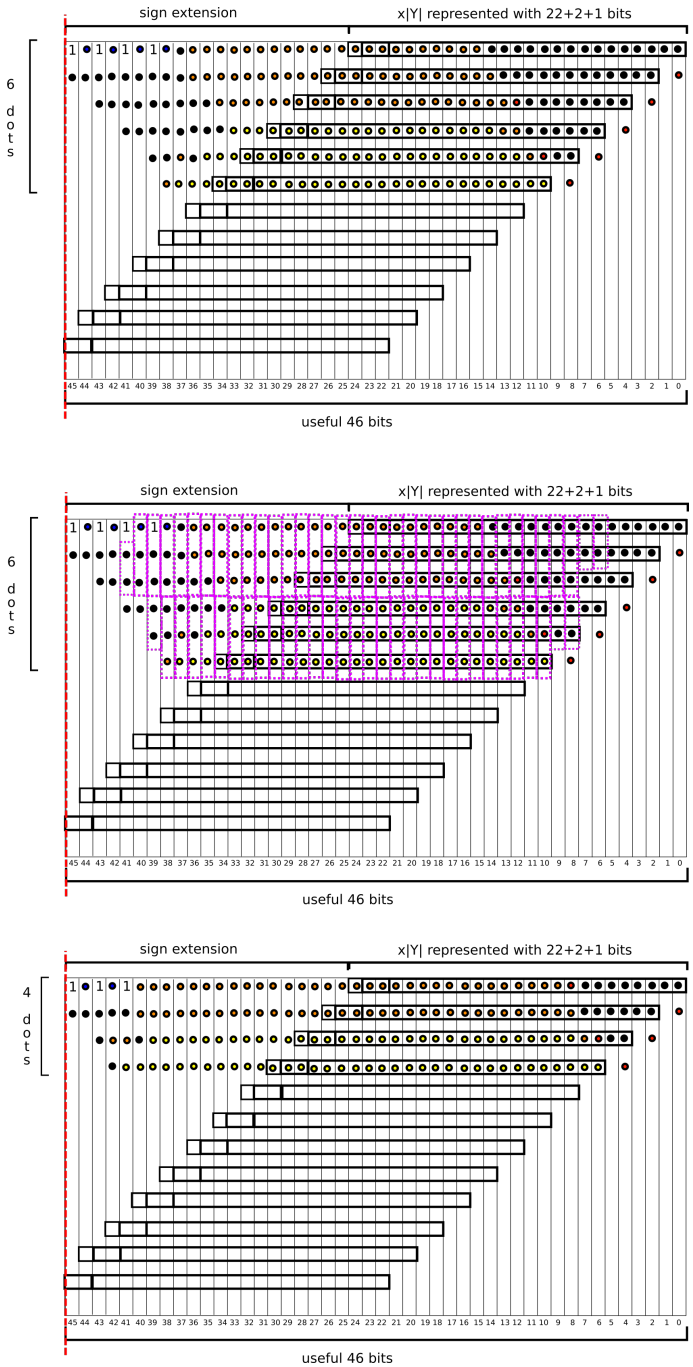

Figure 5: Applying the Roorda's method to reduce the numbers of adders

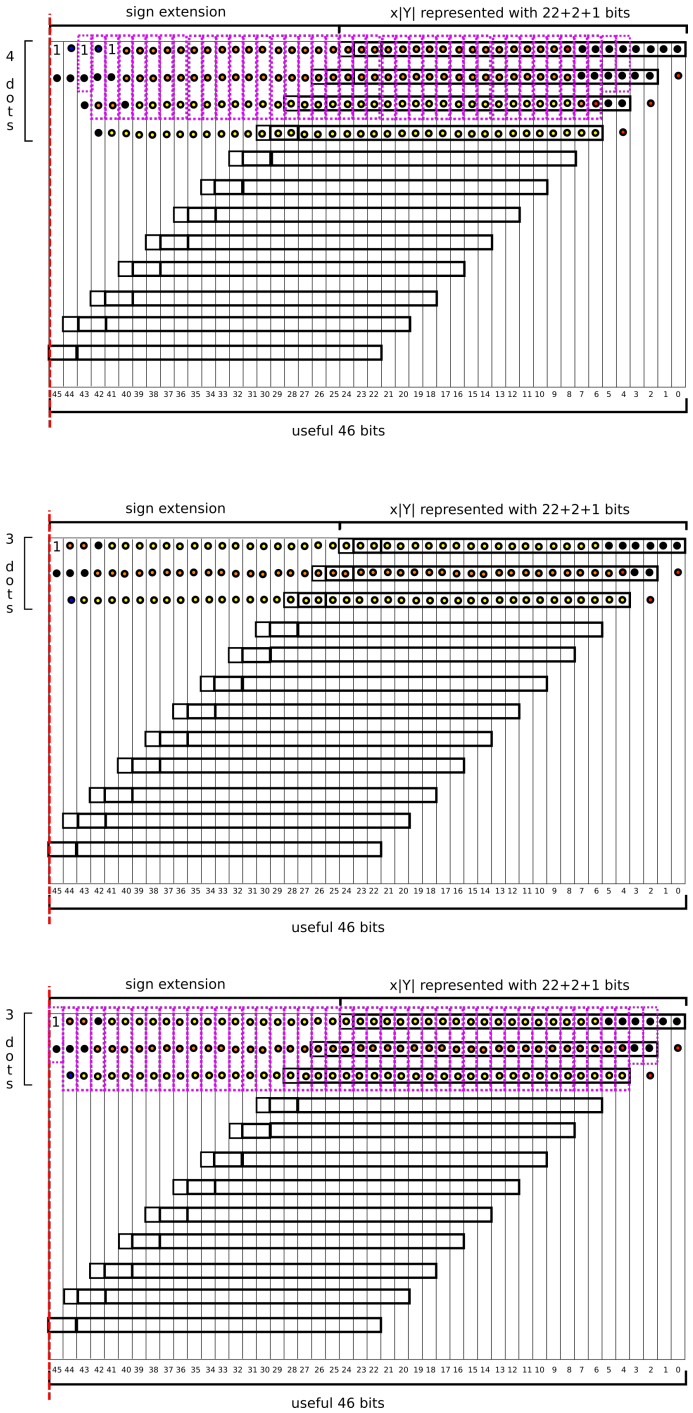

Figure 6: Applying the Roorda's method to reduce the numbers of adders

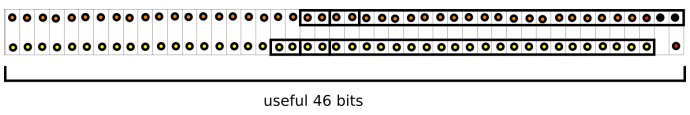

Figure 7: Applying the Roorda's method to reduce the numbers of adders

before the first binding. This is in contrast with the scheme of 4, where the first bininding is done before the flattening. All the other figures show exactly what is described by the VHDL.

The process of bringing all the bits to the ceiling of the grid is done by the hand written part of the file and it is word length dependent. It's not difficult to implement it in the python script, and this improvement would make the entire multiplier parametric.

The python script As already said only the first part of the VHDL has been kept hand written. The bindings of the tree, the instantiation of the adders and the assignments from one grid to the next are fully automatic.

In its basic version it needs only a list with a number of cells equal to the number of columns of the grid, with each cell containing the number of elements of the corresponding column. With this information the script, starting from the rightmost column (i.e. from the column of the LSB of the result), analyzes if there's the need for a compressor (a FA, HA or an approximated one) and writes into another list the number of adders needed.

Then the main routine writes on the file all the component instantiations and the assignments to end up with a new grid with all the elements flattened to its top, respecting the DADDA approach to have the minimum number of levels with a ALAP resources binding approach.

Both for bindings and assignments the script has to deal with the carries of the compressors, to work well. The details of the routine can be found looking at the code.

The script works also for versions 2 and 3 of the project: it receives parameters to design a multiplier in which an arbitrary number of LSB columns does not appear in the tree (version2 of the project) and to use a variable number of approximated compressors when the binding is performed. These two design choices can be important to save area and power (and maybe to increase the clock frequency, but it depends on the used ports and on the technology). A more detailed description of the working principles of the script in designing v2 and v3 multipliers can be found in the corresponding sections.

The script produces also a drawing of each level of the tree: this output can be redirected on the prompt or to a file and used to check the correctness of the result of the bindings.

3 Approximated architectures analysis

3.1 Approximated multiplier architecture

The main idea for *version 2*, as suggested in the lab paper, is to approximate the partial products in order to gain mainly in terms of power and area. This can be achieved by simply truncating **k** leftmost bits of every partial product before allocating any half or full adder in those positions, as shown in the following picture. Obviously we pay in precision since we loose the contribution on final resut for those bits we've set to '0'. In the following sections we will explain the analysis on how multiplier and filter characteristics changes by varying the k paramiter.

3.2 Approximations with 4-2 compressors

A further approach, to improve the three figures of merit of the design, is to substitute the FAs and the HAs with approximated compressors which take in input four bits with the same weight and give in output one bit of the same weight of the inputs and one bit with double this weight. In **??** this technique with a cutting of the LSBs was adopted, and it was shown good results wrt the main figures of merit.

The approximation is variable and it is function of how many 4-2 compressors are used in the multiplier tree and where. The proposed compressor implementation can only miss the sum bit, while the carry one is always correct. This implies that the maximum error is related to the weight of the column in which the compressor has been allocated: it's easy to understand that an approximated compressor allocated on the a *left-most* bit creates an error higher than the one produced by a compressor put more on the right.

The script we used for the standard version allows also to generate many multipliers with a different number of approximated compressors, allocated in different positions.

script The basic version of the script performs the binding only with FAs and HAs. It starts from the LSB and assigns as many FAs as possible where needed to compress the bits' number of a certain column following DADDA's approach. If the reduction from one level to the next is possible using an HA instead of the last FA, then an HA is assigned.

To create an approximated multiplier the binding is performed giving precedence to the 4-2 compressor instead of the FA. Ideally a complete covering with 4-2 compressors can be done, but in order to analyze and fully understand how system-level characteristics change with the number of approximated compressors in the architecture, was introduced in the script the possibility to give as an argument a percentage of approximated adders to be inserted in the binding. Indeed it is possible to ask for a multiplier with a compression level of 40 %, and this means that only the 40 % of the compressors will be put, starting from the MSB or LSB. This last choice is determined by the other passed parameter, the "startingDirection" one.

Where a compression is needed but no compressors can be put, FAs and HAs are used instead. The latters are used also to respect the DADDA "maximum rows" allowed in the tree levels : the algorithm looks always for the minimum compression which allows the minimum number of levels.

3.3 Data collection

In order to understand better how we solved the data collection part, for the explanation we will consider the following flow chart.

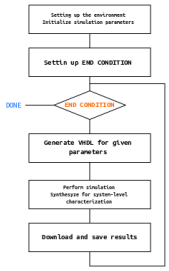

Figure 8: Data collection algorithm

A Python script performs all these actions calling subfuntions already used for first lab and others done ad-hoc for this step.

Setting the environment First passage consists of connecting to the server and cleaning working directories locally and remotely. Then input sample vectors are generated and uploaded to the server with the specific scripts and all the common files needed for simulation. In this first step, model of the IIR filter and of a multiplier are run to produce correct results, usefull for the final analysis.

End condition The *END CONDITION* is set according on which type of data we want to collect. Examples can be:

- for Number_of_LSB_to_drop in renge(0,10,1)
- for Persantege_of_approx_compressor in range(0,50,5)

In the first case we are cycling from 0 to 9, incrementing the iterator by one each cycle. Similar is the second case.
In this way we can obtain all the data we need launching just once the script.

Generate VHDL First step of the loop is the automatic generation of the VHDL description of the Dadda multiplier. To execute the generator we have to specify three parameters:

- Compression Level, percentage of approximated compressor
- Starting Direction, left or right
- Aproximation Bits, number of leftmost bits that have to be truncated

Notice that it is possible to generate a hybrid multiplier whith k bits truncated and a certain $h\%$ of 4-to-2 compressor.

Simulation and Synthesis The simulation and synthesis part is performed twice: once just for the multiplier, and the second time for the whole filter. This is done to understand how different "types" of multiplier impact on the multiplication itself and to the final result of the filter. From the input parameters we gave at the previous step to the final output we can keep track of all the dataflow through our aproximated block.

We have to notice that it is possible to set a flag to enable the Compile Ultra option of Synopsis. This will couse the elaboration time to increase but a more precise synthesis of the circuit.

Result downlaod Last step consist of saving locally results and report from previous elaboration.

3.4 Data Analysis

Starting from all the files downloaded after simulation and synthesis a third script computes error statistics from approximated architecture simulation results and collects significant values from synthesis reports. Actual list of action performed is the following one:

- Test the results from truncated architecture multiplier with Python model.
- Compute error statistics from truncated multiplier.
- Compute error statistics from filter with truncated multiplier.
- Collect synthesis results from truncated multiplier
- Collect synthesis results from filter with truncated multiplier.
- Compute error statistics from multiplier with approx. compressors.
- Compute error statistics from filter with multiplier with approx. compressors.
- Collect synthesis results from multiplier with approx. compressors.
- Collect synthesis results from filter with multiplier with approx. compressors.

Error statistics computation A fast but important review has to be done in order to understand how the error analysis is handled. We decided to use relative distance between exact result and approximate one, since is the only way to have an idea of how much approximation parameters influence final result.

$$e_i = \frac{Se_i - Sa_i}{Se_i} \tag{5}$$

All the results processed by a given architecture are analysed keeping track of the **maximum and the average relative error**. All these values are then stored into log files ready to be used to create graphs or do any kind of analysis.

3.5 Limits and possible improvements

No power analysis Maximum bit we can drop is ???