Integrated Systems Architecture

Politecnico di Torino

Department of Electronics and Telecommunications

Laboratory 2: Digital arithmetic

Authors:
Marco Andorno (247222)
Michele Caon (253027)
Matteo Perotti (251453)
Giuseppe Sarda (255648)

March 21, 2019

This report along with all the source files, scripts, reports and diagrams for the project can be found on GitHub at https://github.com/mksoc/ISA-digital-arithmetic.

Contents

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).

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 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

[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.

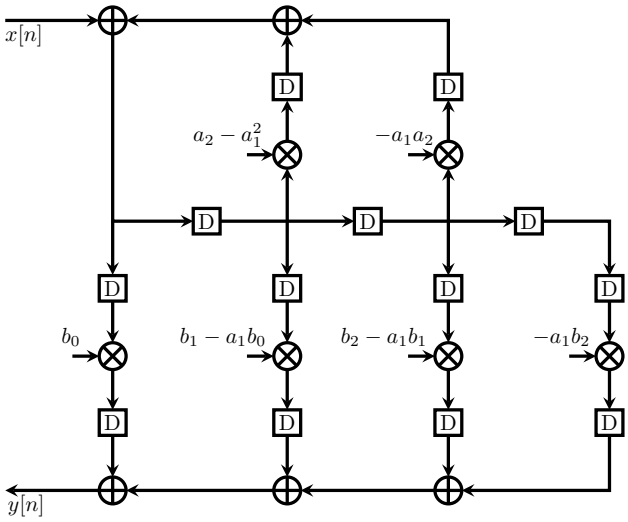


Figure 1: DFG of the filter

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 Fine grain retiming

Synopsys Design Compiler is able to perform fine grain pipelining, moving the registers at the input and output of the arithmetic components inside them. In other words, the internal architecture of each operator is split two or more smaller parts, separated by an internal pipeline register. This allows to better balance the critical paths inside the design. In our case, as discussed in the previous section, the critical path is represented by the delay of a single multiplier T_m. However, the loop bound in the recursive part of the filter that states the best possible scenario for the delay is given by:

$$T_\infty = \frac{2T_a + T_m}{2} = T_a + \frac{T_m}{2}$$

Since T_a is likely to be lower than $T_m/2$, there still room for improvement exploiting fine grain retiming inside the loops. This is done enabling the *ultra mode* in Design Compiler with the command:

```
set_ultra_Optimization true
```

and using the command `compile_ultra` in place of the former `compile` used before. Notice that in a FIR filter this technique is much more effective than in IIR filters. In a FIR, an arbitrary amount of pipeline stages can be introduced. This means that each operator can be split in an arbitrary number of smaller components, since there's no loop bound setting an upper limit to the performance improvement because there are no loops at all in the FIR, feed-forward architecture. If the arithmetic operators are feed-forward too, as the Brent-Kung, Kogge-Stone or Ladner Fisher adder, the theoretical upper bound to the performance is the delay of a single logic level between two pipeline register. Of course, the overhead due to the clock-to-output delay and setup time of the registers. Moreover, a heavily pipelined architecture might increase the latency.

2.1 Synthesis results

Performing the synthesis at the maximum possible frequency with the architecture reported in figure 1, we obtained the timing, area and power results shown in table 4. Notice that this time Design Compiler is not told which implementation to use for adders and multipliers.

f_max [MHz]	847.46
Area [µm^2]	20566.3
Dynamic Power [µW]	4852
Leakage Power [µW]	454.8

Table 4: Compile ultra synthesis results

The maximum frequency increased by 30% with respect to the best solution found using the standard optimization mode. The area increased by 54% with respect to the `pparch` version synthesized in the in the previous section. Regarding power, the dynamic power has increased by 25%, while the increase in leakage power is ten times higher than the one extracted using the `compile` command. These results mark the employment of the *ultra mode* suitable only if speed is the only figure of merit the designer is interested in.

2.1.1 Further improvements

As already pointed out in the first section, the critical path can be reduced from $3T_a$ to $2T_a$ inserting an additional pipeline stage in the feed-forward part of the filter. Performing this change here too, we obtained the results shown in table 5. The maximum frequency increased by an additional 5%, and the area by an additional 14%. This way, the critical path is already in the loops of the recursive part. This is proven true by the fact that adding an additional pipeline level to the feedforward part doesn't lead to better performance.

f_max [MHz]	892.86
Area [µm^2]	23489.4
Dynamic Power [µW]	-
Leakage Power [µW]	433.2

Table 5: Compile ultra synthesis results, optimized critical path

3 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 $\mathbf{y}$. This shrewdness allows us to reduce the number of partial products by half: indeed $\mathbf{y}$ is encoded with $\boldsymbol{k_y^{'}}$ symbols in $\{\pm 2, \pm 1, 0\}$. If $r = 4$ is the radix:

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

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 $\mathbf{y}$ 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 $\mathbf{0}$, then for correctly encoding $\mathbf{y}$ we must add a $\boldsymbol{y_{-1}}$ bit fixed at $\mathbf{0}$ 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 6.

The product is now between $\mathbf{x}$ and $\mathbf{Y}$, the MBE-radix4 encoded version of $\mathbf{y}$. Each partial product between a symbol of $\mathbf{Y}$ and $\mathbf{x}$ 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 $\mathbf{0}$, or $\mathbf{x}$, or $\mathbf{2x}$. 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: $\mathbf{0}$, $\mathbf{x}$, **-x**, **-2x** and $\mathbf{2x}$. 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

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

Table 6: MBE conversion table

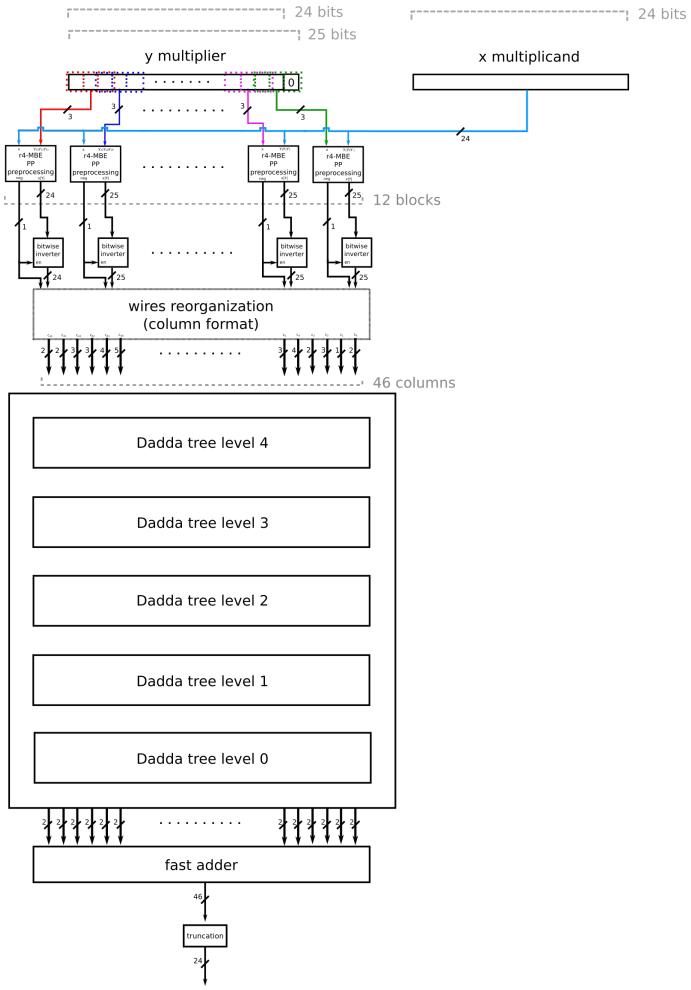


Figure 2: DP of the filter multiplier based on a DADDA tree, with Roorda's approach on MBE-r4 encoding

thought as 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 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

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

At the end we end up with 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

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

In a single column we can count up to 13 elements, because the MUX let only pass **x** multiplied by 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 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.

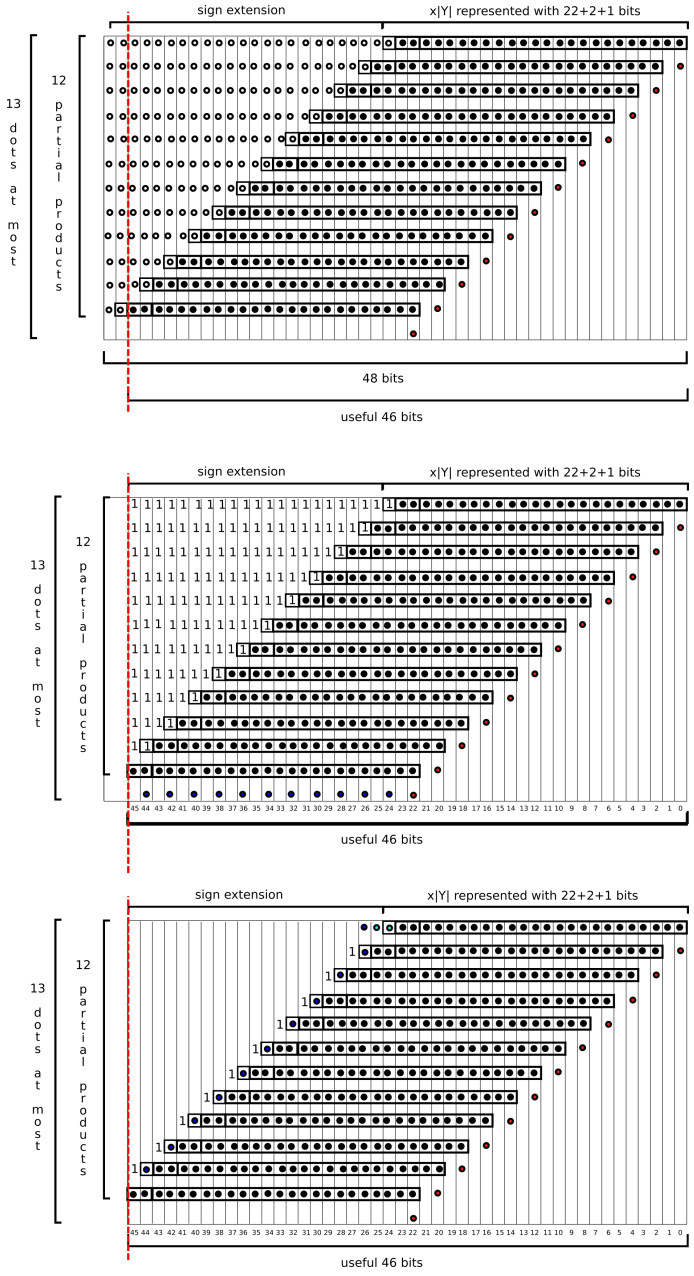


Figure 3: Applying the Roorda's method to reduce the numbers of adders

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.

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

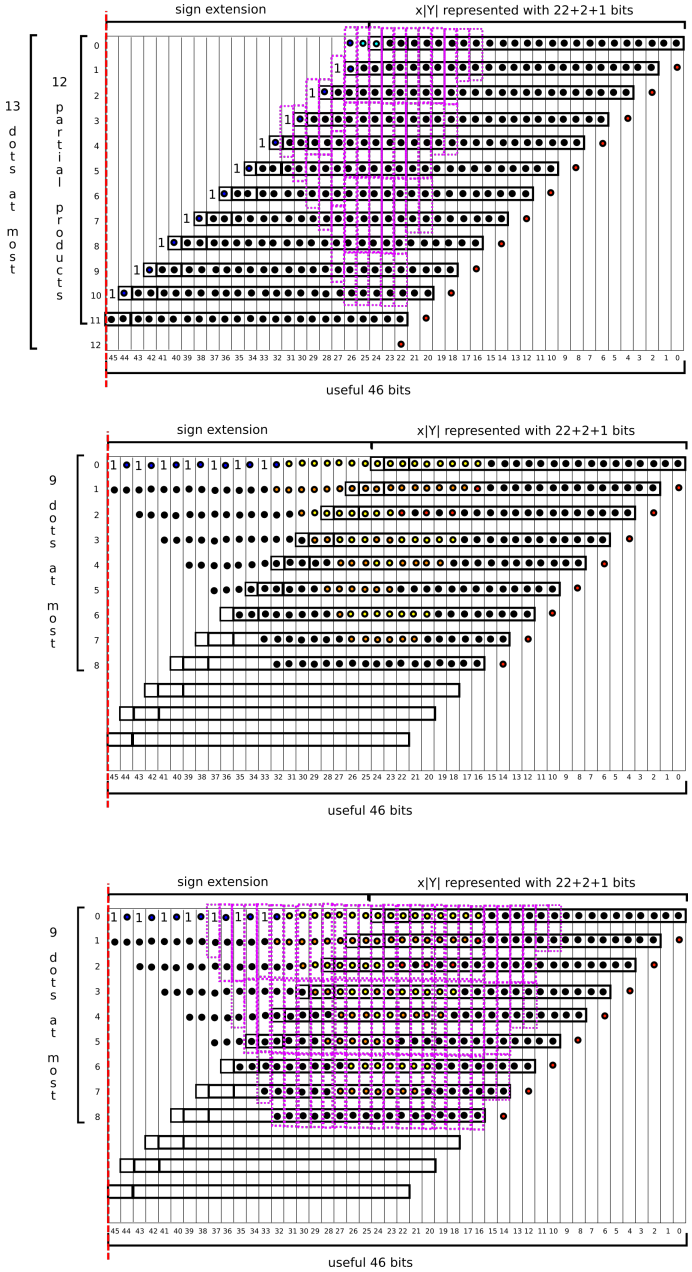


Figure 4: Applying the Roorda's method to reduce the numbers of adders

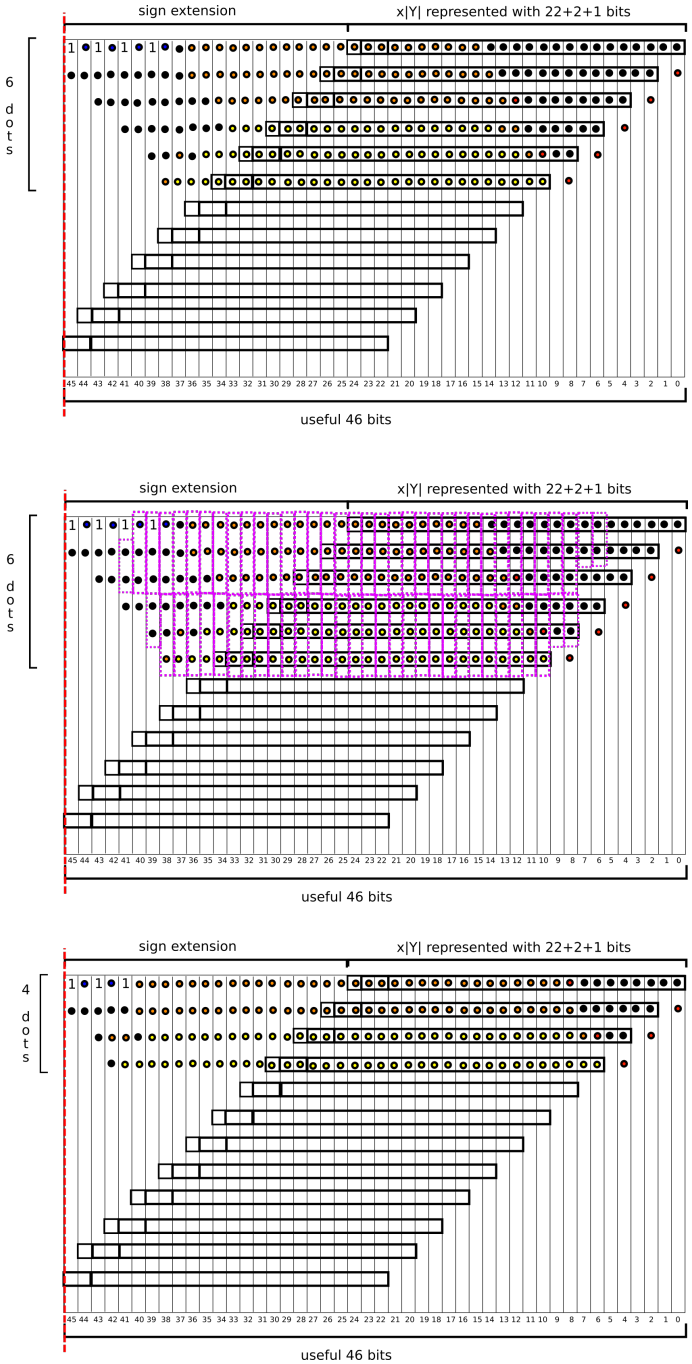


Figure 5: Applying the Roorda's method to reduce the numbers of adders

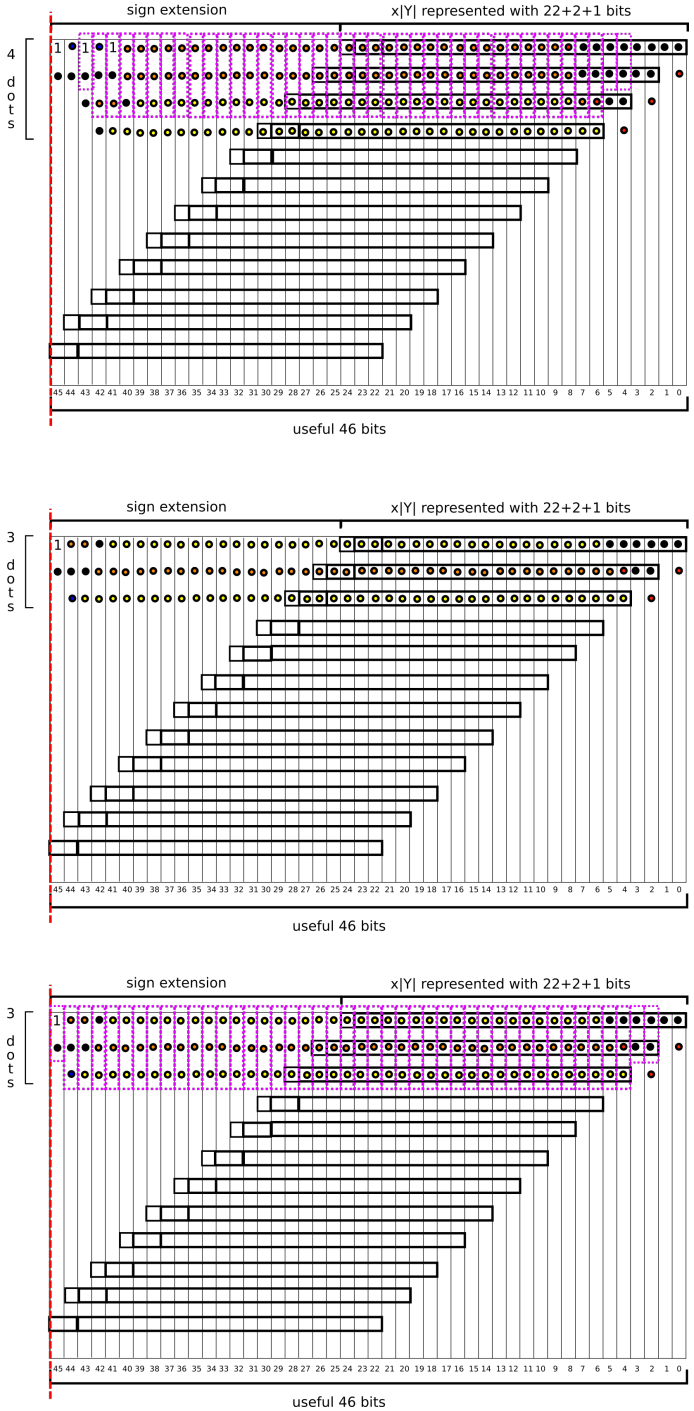


Figure 6: Applying the Roorda's method to reduce the numbers of adders

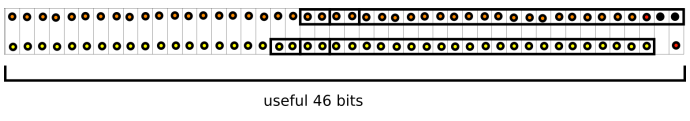


Figure 7: Applying the Roorda's method to reduce the numbers of adders

The last operation is 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 products 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 **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 dadda-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 strategy to have the minimum number of levels with an 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 (version 3 of the project). 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 when a VHDL is automatically written (figures 8, 9, 10, 11).

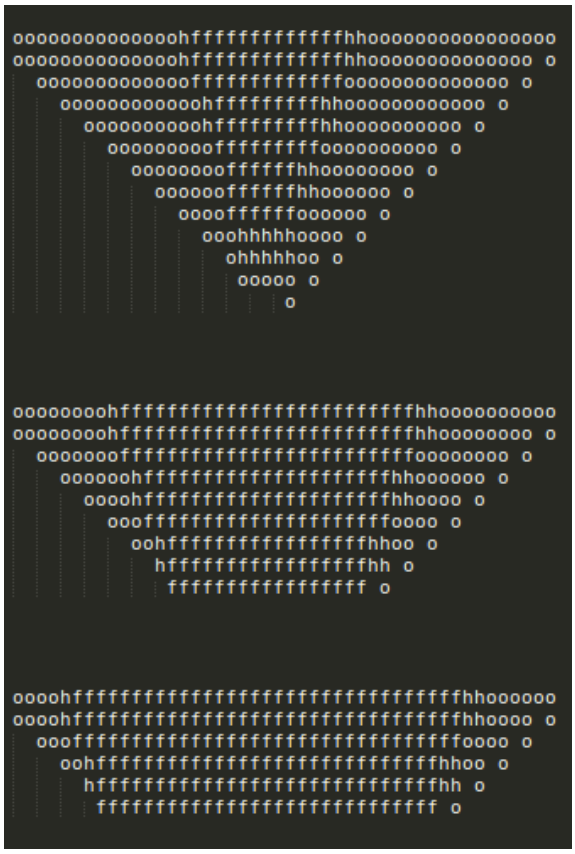


Figure 8: Dadda terminal drawing (0), compression level: 0%

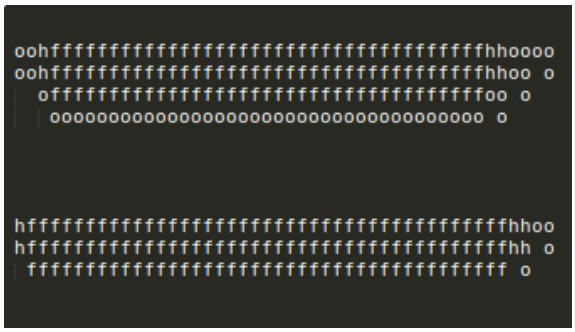


Figure 9: Dadda terminal drawing (1), compression level: 0%

Figure 10: Dadda terminal drawing (0), compression level: 70%

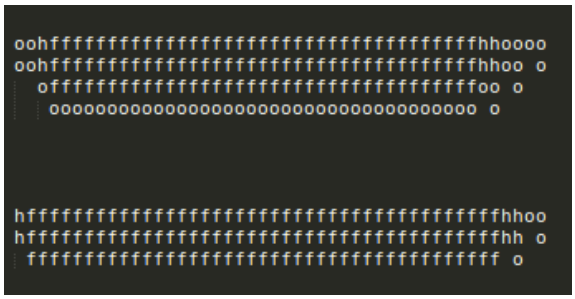


Figure 11: Dadda terminal drawing (1), compression level: 70%

4 Approximated architectures analysis

4.1 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.

4.2 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 a approximated compressors allocated on the *left* bits create an error higher than the one produced by compressors 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, the possibility to give as an argument a percentage of approximated adders to be inserted in the binding was introduced in the script. 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. So, the algorithm calculates how many compressors would be put if the copression level was 100%, then it inserts only the asked percentage of them.

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.

4.3 Data collection

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

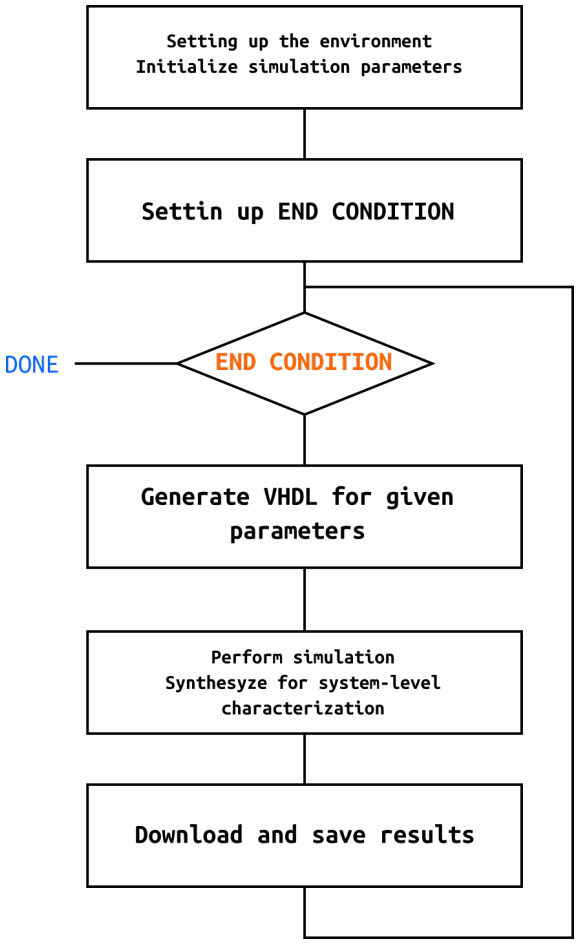


Figure 12: Data collection algorithm

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 The 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, useful 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.

4.4 Data Analysis

Starting from all the files downloaded after simulation and synthesis a third script computes error statistics from approximated-architecture outputs 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 both absolute and 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. The relative distance is calculated as:

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

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.
For the simulation we decided to generate 250000 couples of inputs for the multiplier and 250000 samples for the filter. The simulation was performed on the VHDL/Verilog model before the synthesis.

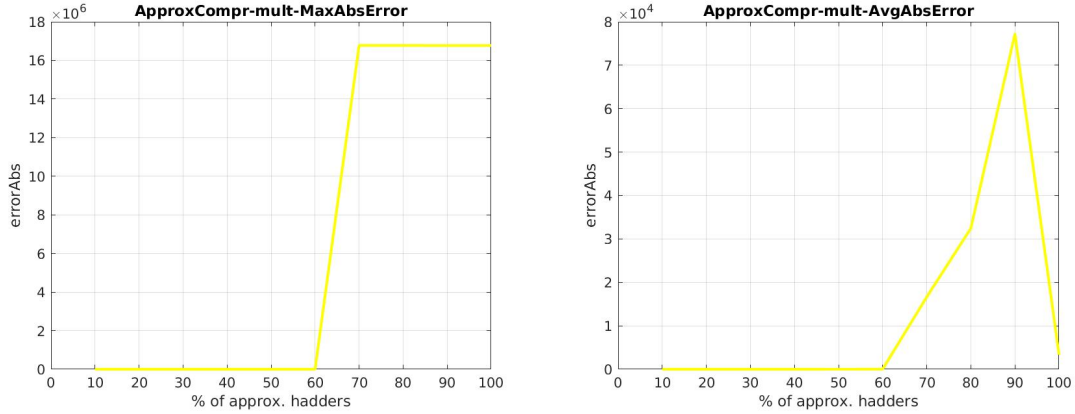
4.5 Multiplier with compressors

An useful metric to evaluate an approximated multiplier is the **NMED**. This parameter assumes a mean over the errors of all the outputs relative to all the inputs. This is a huge task when we have to deal with high parallelisms. So we used a high set of inputs and calculated the mean and maximum error. Calculating the NMED would be an improvement for our work.

Simulation Ten multipliers have been tested besides the original, each of them with a different percentage of inexact compressors among the correct ones. The relative and absolute errors are reported in the following tables and plots. The numbers were read as integers, but it's easy to find the results in terms of Q2.22 numbers only multiplying by $2^{-k_x^F} = 2^{-22}$.
The multiplier under verification, as we said above, is only capable of doing multiplications between two numbers in format Q2.22, but one of them has to be lower than the unity. This is because it was designed and optimized with the two upper MSBs directly linked to the MSBs of the result, to enhance the area occupation and the power consumption of the design of the IIR filter.
To correctly test the design the inputs of the multiplier have to respect this constraint: they were generated to stay on 23 bits. This ensures the result is representable on 46 bits at maximum. With the last 22 bits which are discarded, the real output can stay always on 24 bits. This is only a part of the behaviour of the multiplier: the correct test would have been a test in which one input is free of being a number in the range [0, 2^(24)-1], whereas the other with an absolute value fixed strictly under the threshold of 1.
With the increasing of the compression, the possibility of incurring in overflow increases. When an overflow occurs, the error skyrockets fastly. Maybe a guard bit could be useful to limit this phenomenon.

Multiplier compression level	*Max error [LSB]*	*Avg. error [LSB]*	*Max relative error*	*Avg. relative error*
10%	1.0	-0.012	-4.85e-05	-7.20e-08
20%	1.0	-0.071	1.1e-03	8.80e-07
30%	-3.0	-0.239	-2.1e-02	-2.12e-05
40%	-7.0	-1.016	1.2e-03	-2.51e-06
50%	-18.0	-4.754	-2.1e-01	-2.300e-04
60%	-75.0	-28.254	-6.38e-01	-7.4e-04
70%	16777090.0	16646.29	356959.36	356.95
80%	16776178.0	32447.17	356939.95	376.83
90%	16774518.0	77205.154	356667.61	387.53
100%	16773938.0	3354.786	356892.30	366.37

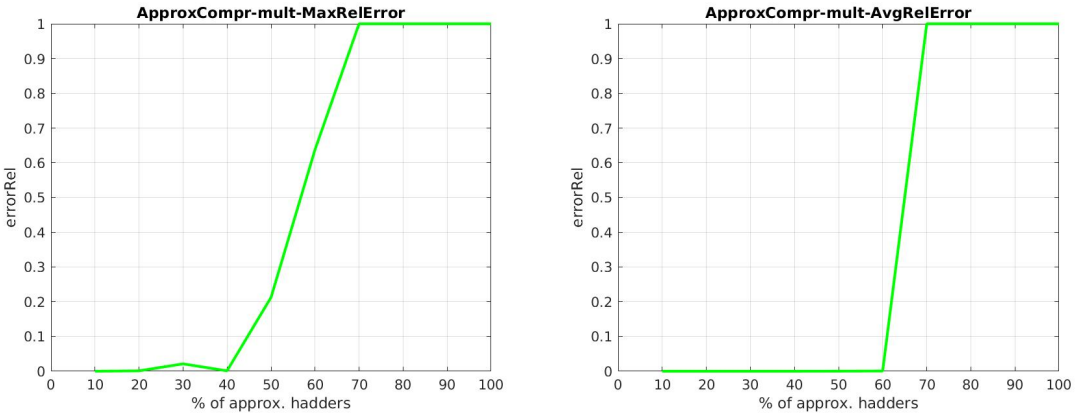
Table 7: Multiplier, simulation and error results



(a) Max Absolute error

(b) Avg Absolute error

Figure 13: Absolute errors for multiplier architecture



(a) Max Relative error

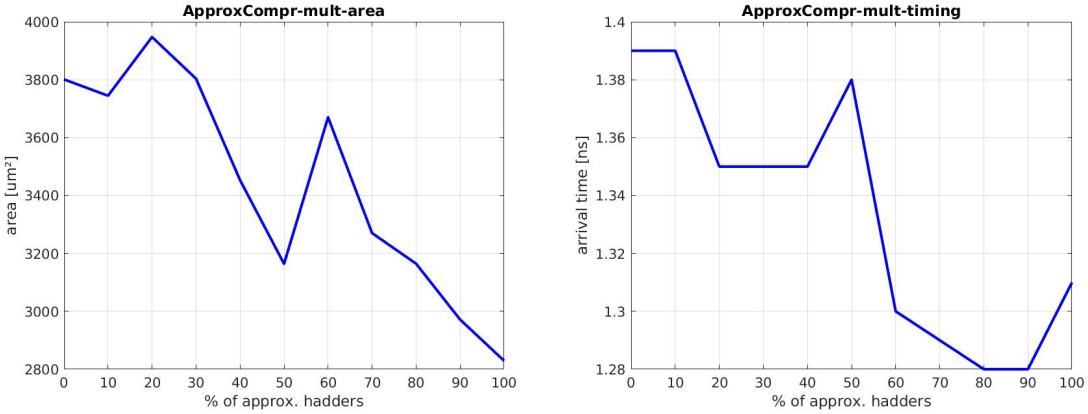
(b) Avg Relative error

Figure 14: Relative errors for multiplier architecture

Synthesis All the multipliers were synthesized and informations of area and timing reported. The minimum delay was found in theoretical way asking Design Compiler to reach a null arrival time.

Multiplier compression level	*Area* *[um^2]*	*Delay* *[ns]*	textit Max. frequency [MHz]
0%	3801	1.39	719
10%	3744	1.39	719
20%	3947	1.35	740
30%	3803	1.35	740
40%	3453	1.35	740
50%	3164	1.38	724
60%	3670	1.30	769
70%	3270	1.29	775
80%	3164	1.28	781
90%	2971	1.28	781
100%	2829	1.31	763

Table 8: Multiplier, area and timing synthesis reports



(a) Area

(b) Arrival time of crithica path

Figure 15: Reports after multiplier synthesis

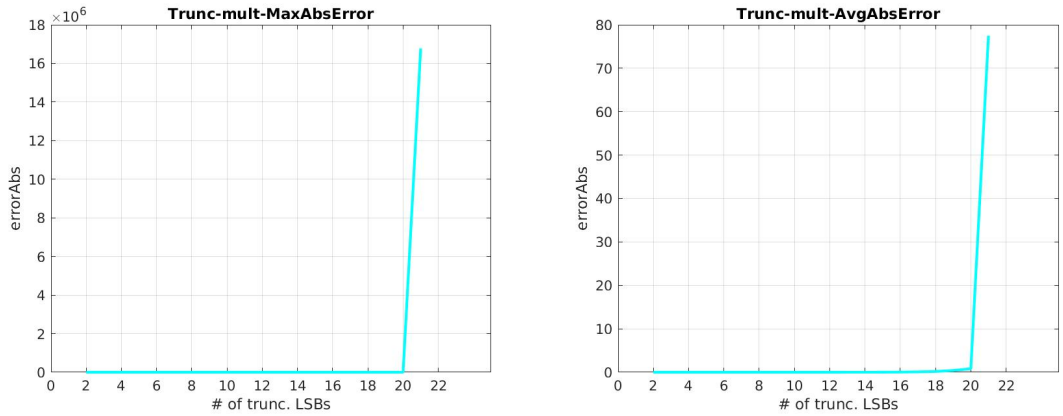
4.6 Multiplier with truncated bits

Another way of simplifying the multiplier, reducing its area, delay and power consumption is to remove a certain number of LSBs from the computation before the tree, after the partial products production.

Simulation The results of the simulation show that the overflow is a danger and has to be avoided. As we said it would be useful to reduce the input dynamic to stay away from the two extremis, or to use a guard bit. This way, if the maximum error is kept lower than the distance between the maximum reachable output value and the two ends of the dynamics, no overflow can occur. Anyway, the error is low, and this seems to be a good way for the approximation of the result of a multiplier.

Multiplier truncated bits	*Max error [LSB]*	*Avg. error [LSB]*	*Max relative error*	*Avg. relative error*
2	0	0	0	0
6	-1.0	-4.72e-06	-1.06e-06	-5.02e-12
10	-1.0	3.2e-04	-7.75e-05	-9.15e-10
14	-1.0	-8.3e-03	-3.5e-03	-9.73e-08
18	-1.0	-1.8e-01	-2.4e-02	-1.59e-06
21	16777214	77.50	16777214	79.3

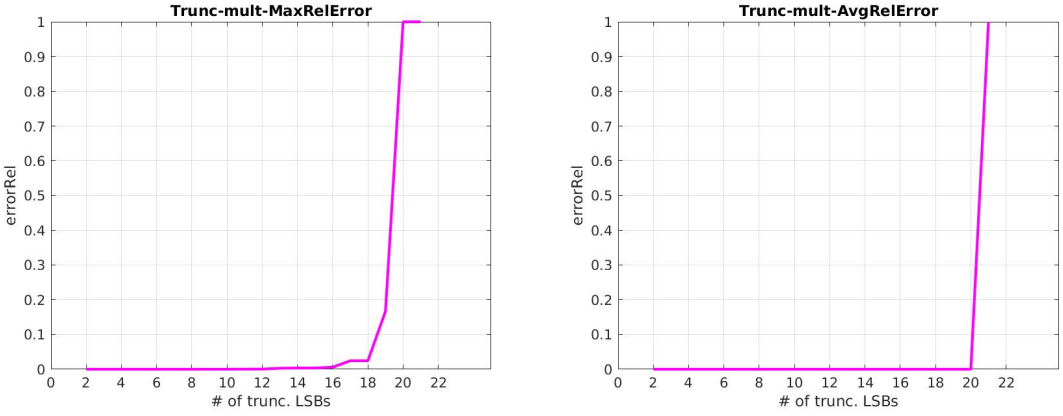
Table 9: Multiplier with truncated bits, simulation and error results



(a) Max Absolute error

(b) Avg Absolute error

Figure 16: Absolute errors for multiplier architecture



(a) Max Relative error

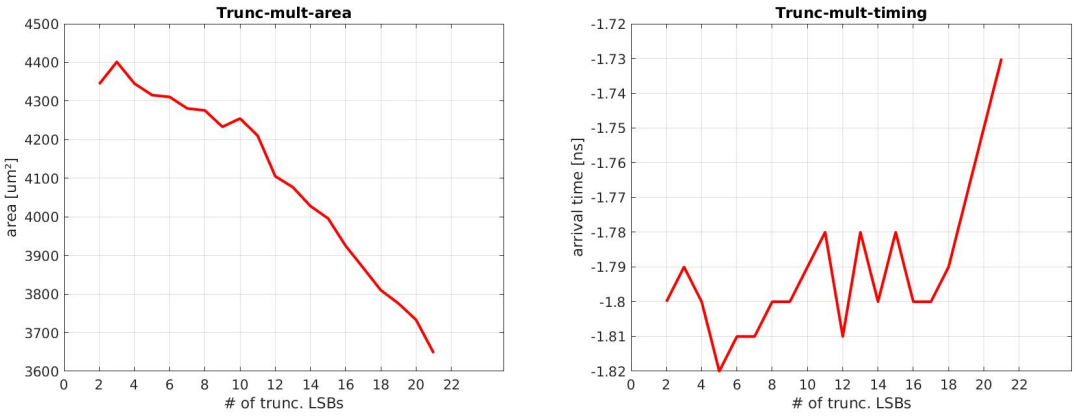
(b) Avg Relative error

Figure 17: Relative errors for multiplier architecture

Synthesis 18 truncated bits seems a good compromise, even better than the choice of the compressors. In the IIR filter it will be clear that the behaviour of the multiplier is really interesting.

Multiplier truncated bits	*Area [um2]*	*Delay [ns]*	*textbf Max. frequency [MHz]*
2	3738	1.41	709
6	3780	1.35	740
10	3612	1.37	729
14	3471	1.37	729
18	3030	1.35	740
21	2815	1.3	769

Table 10: Multiplier with truncated bits, area and timing synthesis reports



(a) Area

(b) Arrival time of crithica path

Figure 18: Reports after multiplier synthesis

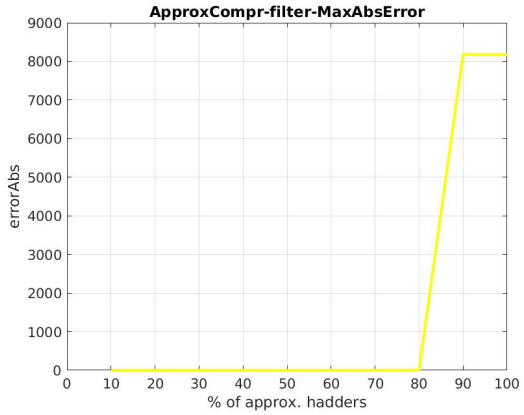
4.7 IIR filter

For each multiplier, a filter was tested and then synthesized.

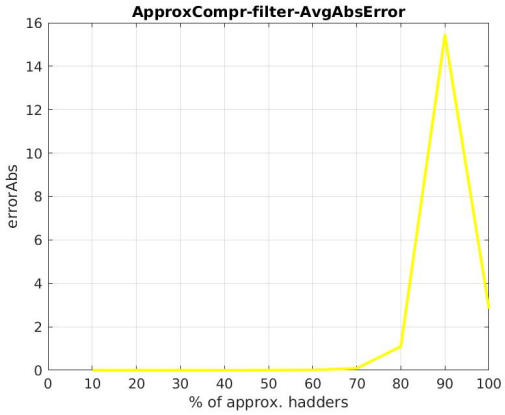
Simulation As it has already been said, 250000 samples was given in input to the filter with the multipliers. The constraints for the multiplier are respected, because it was shaped for working with this filter. Even here the overflow can occur. It can be linked to high probability of an overflow in the multiplier and also with the high avarage error, which is dangerous in the IIR filters because can accumulate and make the accumulation point diverge.

IIR filter compression level	*Max error [LSB]*	*Avg. error [LSB]*	*Max relative error*	*Avg. relative error*
10%	0	0	0	0
20%	0	0	0	0
30%	0	0	0	0
40%	1	0.001	7.79e-04	7.78e-07
50%	-1	-0.005	5.08e-03	-6.40e-06
60%	-1	-0.014	2.22e-02	-3.48e-05
70%	-1	-0.086	1.42e-01	2.70e-04
80%	-4	-1.093	-1	3.87e-03
90%	8183.0	15.44	8177.0	10.95
100%	-8177.0	2.84	371.04	0.63

Table 11: IIR filter with approximated compressors, simulation and error results

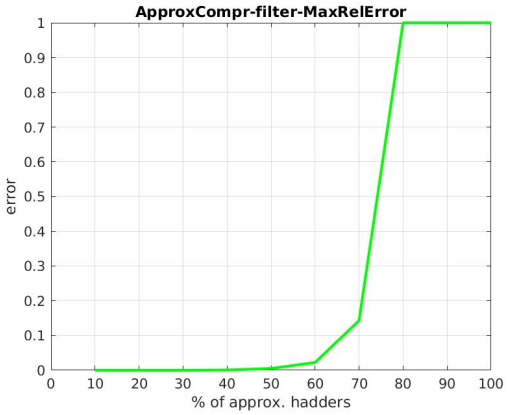


(a) Max Absolute error

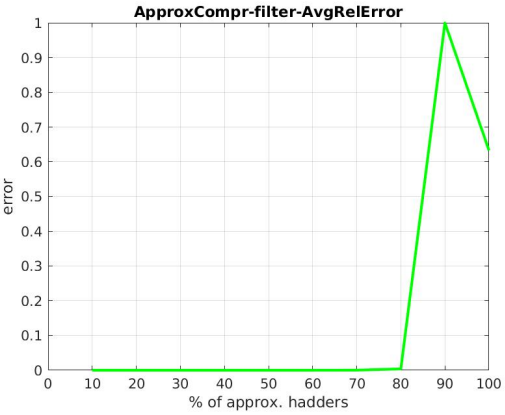


(b) Avg Absolute error

Figure 19: Absolute errors for filter architecture



(a) Max Relative error



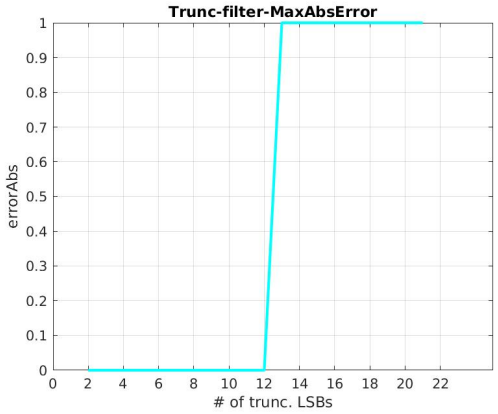
(b) Avg Relative error

Figure 20: Relative errors for filter architecture

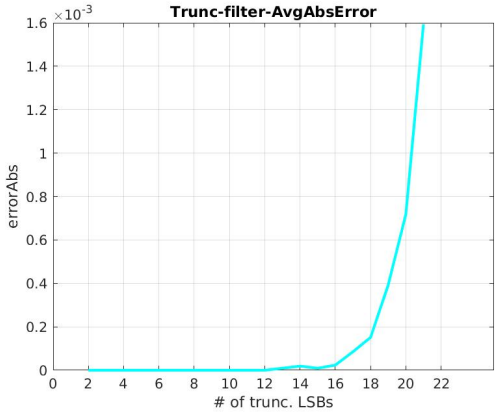
From the table of the IIR filter and the truncated bits multiplier is clear that the choice of the truncated bits is preferrable.

IIR filter with truncated bits multiplier	***Max error [LSB]***	***Avg. error [LSB]***	***Max relative error***	***Avg. relative error***
2	0	0	0	0
6	0	0	0	0
10	0	0	0	0
14	-1.0	-1.89e-05	-9.6e-03	4.46e-08
18	-1.0	-1.5e-04	-7.7e-02	-4.53e-07
21	-1.0	-1.6e-03	-1.0	1.05e-05

Table 12: IIR filter with truncated bits, simulation and error results

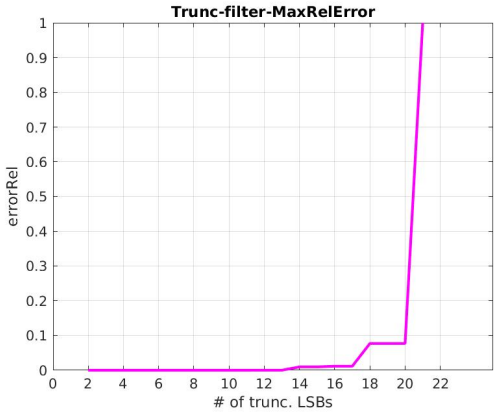


(a) Max Absolute error

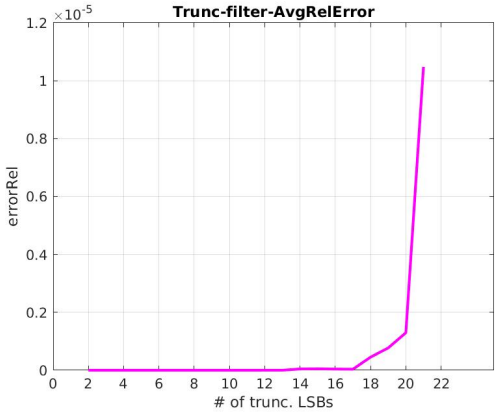


(b) Avg Absolute error

Figure 21: Absolute errors for multiplier architecture



(a) Max Relative error



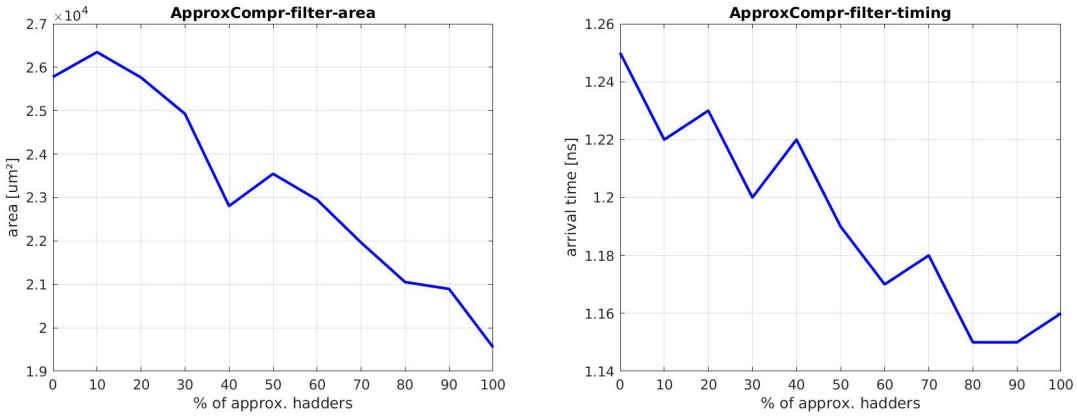
(b) Avg Relative error

Figure 22: Relative errors for filter architecture

Synthesis The results of the synthesis are visible in the following table and figure. The synthesis was done letting the program to perform retiming, this is why the delay is lower than for a single multiplier. These results were found without giving the *compile_ultra* command, only to have an idea of the trend.

IIR filter compression level	***Area*** *[um^2]*	***Delay*** ***[ns]***	textit **Max. frequency [MHz]**
0%	25778	1.25	800
10%	26348	1.22	819
20%	25763	1.23	813
30%	24925	1.20	833
40%	22804	1.22	819
50%	23546	1.19	840
60%	22949	1.17	854
70%	21962	1.18	847
80%	21053	1.15	869
90%	20894	1.15	869
100%	19551	1.16	862

Table 13: IIR filter, area and timing synthesis reports



(a) Area

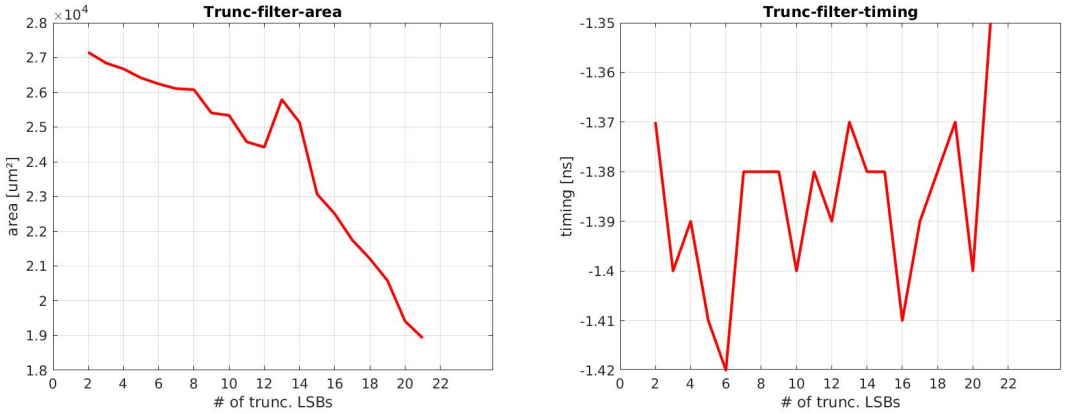
(b) Arrival time of crithica path

Figure 23: Reports after filter synthesis

The precision of the architecture is high even if a lot area is saved. The frequency is also increased, even if it seems that the compressors do a better job with the timing.

IIR filter truncated bits	***Area*** *[um^2]*	***Delay*** ***[ns]***	*textbf Max. frequency [MHz]*
2	26644	1.25	800
6	26451	1.22	819
10	24079	1.26	793
14	23496	1.22	819
18	20626	1.22	819
21	17540	1.19	840

Table 14: IIR filter with truncated bits multiplier, area and timing synthesis reports



(a) Area

(b) Arrival time of crithica path

Figure 24: Reports after filter synthesis

It was chosen one multiplier of each type to stay below a certain error on the result and the synthesis was repeated with the *compile_ultra* command active. It seems that the multiplier with the 60% of the compressors could be a good compromise: in the filter the maximum error is of 1 LSB, and it is at maximum the 2.2% of the value. On avarage it makes mistakes not greater than the 0.0035% of the real value. The other multiplier we take to synthesize with compile_ultra actived is the 18 truncated bits ones. This is also better than the previous. Let's see tha data.

Multiplier with approx compressor and compile_ultra	***Area [um^2]***	***Delay [ns]***	***Frequency [MHz]***
60%	3149	1.23	813

Table 15: Multiplier with compile_ultra, area and timing synthesis reports

IIR filter with multiplier approx compressor with compile_ultra	***Area [um^2]***	***Delay [ns]***	***Frequency [MHz]***
60%	20490	1.10	909

Table 16: IIR filter with approx compressors multiplier with compile_ultra, area and timing synthesis reports

Multiplier with truncated bits with compile_ultra	***Area [um^2]***	***Delay [ns]***	***Frequency [MHz]***
18	2808	1.29	775

Table 17: Multiplier trunc. bits with compile_ultra, area and timing synthesis reports

From this comparison seems that the best compromise is the 18 truncated bits multipliers. It allows to keep an uncertainty lower than the -0.0000453%, with a lower delay (the frequency, even if this parameter is only indicative, is increased of the 12.5%) and a very little area wrt to the original architecture (18717 μm^2 against 25778 μm^2, which is almost 30% of saving).

IIR filter with multiplier truncated bits with compile_ultra	***Area [um^2]***	***Delay [ns]***	***Frequency [MHz]***
18	18717	1.11	900

Table 18: IIR filter with trunc. bits multiplier with compile_ultra, area and timing synthesis reports

4.8 Limits and possible improvements

To improve our analysis it would be better to extend the multiplier to create a general one without the limitation of the input dynamics for one of the two inputs: this would allow us to fully characterize the device.
As already said an useful metric to be calculated would be the NMED: the MED normalized on the maximum output of the correct design. To calculate the MED for a deterministic design where a single input vector can generate only one output, it is possible to take the mean (if the inputs are equiprobable) of the Error Distances (ED) of all the possible outputs related to all the possible inputs [3].
If each input vector i appears with a probability p_i and has a related output ED_i, then the MED can be calculated as follows:

$$MED = \sum_{i=1}^{N} [ED]_i \; p_i \tag{6}$$

To obtain useful results it is necessary an analysis on the probabilities of the input vectors, and then a simulation with each of them.

Another enhancement could be to calculate the power dissipation of the designs. It is very important to quantify this parameter, especially because the hardware reduction is easily followed by a power decreasing.

The calculation of the partial products in the truncated multipliers is another thing can be improved. Indeed, it is possible to calculate only the useful bits and save on the hardware which would work for producing bits which will be discarded later.

References

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

[2] P. Yin, C. Wang, W. Liu, E. E. Swartzlander Jr., and F. Lombardi. Designs of approximate floating-point multipliers with variable accuracy for error-tolerant applications. *Springer Journal of Signal Processing Systems*, 2018

[3] Liang, J., Han, J., & Lombardi, F. (2013). New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on Computers*, 62(9), 17601771