

Integrated Systems Architecture

POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS

Laboratory 1: Filter design

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-filter-design>.

Contents

1	Filter design	3
2	Fixed-point C model	5
3	Base architecture design	5
3.1	Datapath	5
3.1.1	Color legend	5
3.1.2	Registers	6
3.1.3	Arithmetic operators	6
3.2	Control unit	7
3.3	Simulation	8
3.3.1	Testbench	8
3.3.2	Scripts	9
3.4	Synthesis	9
3.5	Place & route	10
3.6	Remarks	11
4	Look-ahead transform	11
4.1	Loop bound analysis	11
4.2	Pipelining and retiming	12
4.3	Datapath	13
4.3.1	Pipe registers	13
4.3.2	Parallelism	15
4.4	Control unit	15
4.5	Synthesis	15
4.5.1	Remarks	15
4.6	Place & route	16
4.6.1	Remarks	16

1 Filter design

Following the rules given in the assignment, the main specifications were derived:

- Filter type: IIR
- Filter order: $N = 2$
- Cutoff frequency: $f_c = 2 \text{ kHz}$
- Sampling frequency: $f_s = 10 \text{ kHz}$
- Data parallelism: $n_b = 12$

Then, using the provided example MATLAB script, the filter coefficients were found by means of the `butter` function. Real coefficients are then quantized as fixed point fractional number in the format $Q1.(n_b - 1)$ ($Q1.11$ in our case) and expressed as integers on n_b bits for the future C model and hardware filter. We will discover later that some care has to be taken when performing operations on the integer representation of fixed point numbers.

Quantization is performed by truncation (`floor` function), so that the maximum error is equal to:

$$\varepsilon_{max} = 2^{-(n_b-1)} = 2^{-11} = 0.049\% \quad (1)$$

We accept that this error could be reduced by rounding and that truncation introduces a negative bias by approximating always towards $-\infty$, because on the other hand truncation is much easier to implement in hardware, where it just represent an arithmetic shift.

The resulting difference equation is in the end:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2] \quad (2)$$

where

$$b_0 = 0.20654 \rightarrow 423$$

$$b_1 = 0.41309 \rightarrow 846$$

$$b_2 = 0.20654 \rightarrow 423$$

$$a_1 = -0.36963 \rightarrow -757$$

$$a_2 = 0.19580 \rightarrow 401$$

Figure 1 shows the transfer function of the filter computed from both the original coefficients and the quantized ones. The two curves cannot be told apart because the error is too small ($5.42 \cdot 10^{-7}$ in the worst case).

Figure 2 on the other hand shows the time domain waveforms of an input signal $x_1(t)$, in blue, containing two frequency components one in band and one out of band, and the corresponding output $x_2(t)$, in red, where only the in-band component survives.

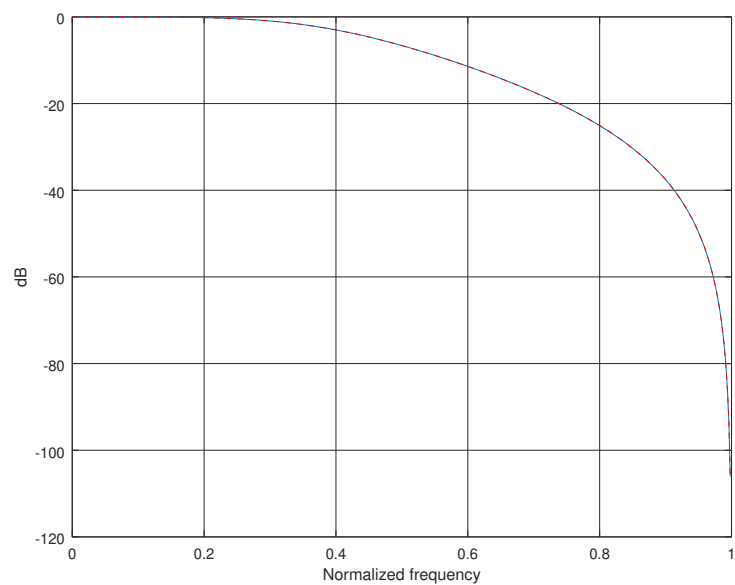


Figure 1: Bode plot of the filter frequency response

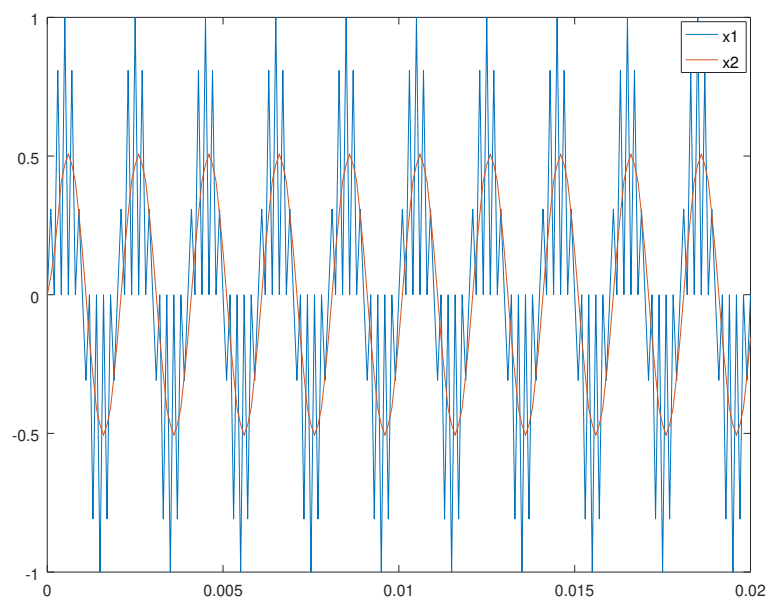


Figure 2: Input and output waveforms

2 Fixed-point C model

The next step consists of writing a software model of the filter in C, which mirrors the behavior of the hardware architecture to be designed next. In this regard, the main difference between the MATLAB model and this one is that the former uses quantized coefficients but performs the internal computation using the maximum precision allowed by the machine, while the latter performs computation always resorting to the original fixed parallelism of data (12 bits here).

The development of this software model started with the example provided in the assignment, tailored to our specifications. A [Python script](#) was developed and used to compare the results file of the two models. As expected, the comparison shows that the two models differ at most of one unity, that is the previously computed ε_{max} (1) in fractional form. Furthermore, results from the MATLAB model, when different, are always greater than the results from the C model, as the latter performs multiple truncations (rounding towards $-\infty$) in its computations.

3 Base architecture design

The development of the first architecture began with the definition of the Data Flow Graph of the filter, shown in figure 3, from which the number of hardware resources needed can be derived (i.e. two delay elements, four adders and five multipliers).

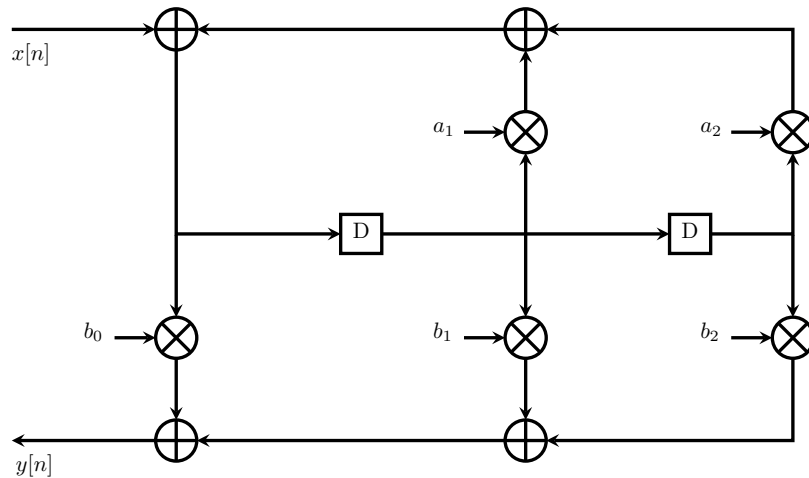


Figure 3: DFG of the filter

3.1 Datapath

From the generic DFG, a full datapath architecture was designed, shown in figure 4. Compared to the simple DFG of figure 3, this datapath explicitly shows all signal names used throughout the design and their parallelism, along with additional interface registers required by the specifications. The following paragraphs detail the key points of the design.

3.1.1 Color legend

All datapath schemes used in this report follow the same color coding:

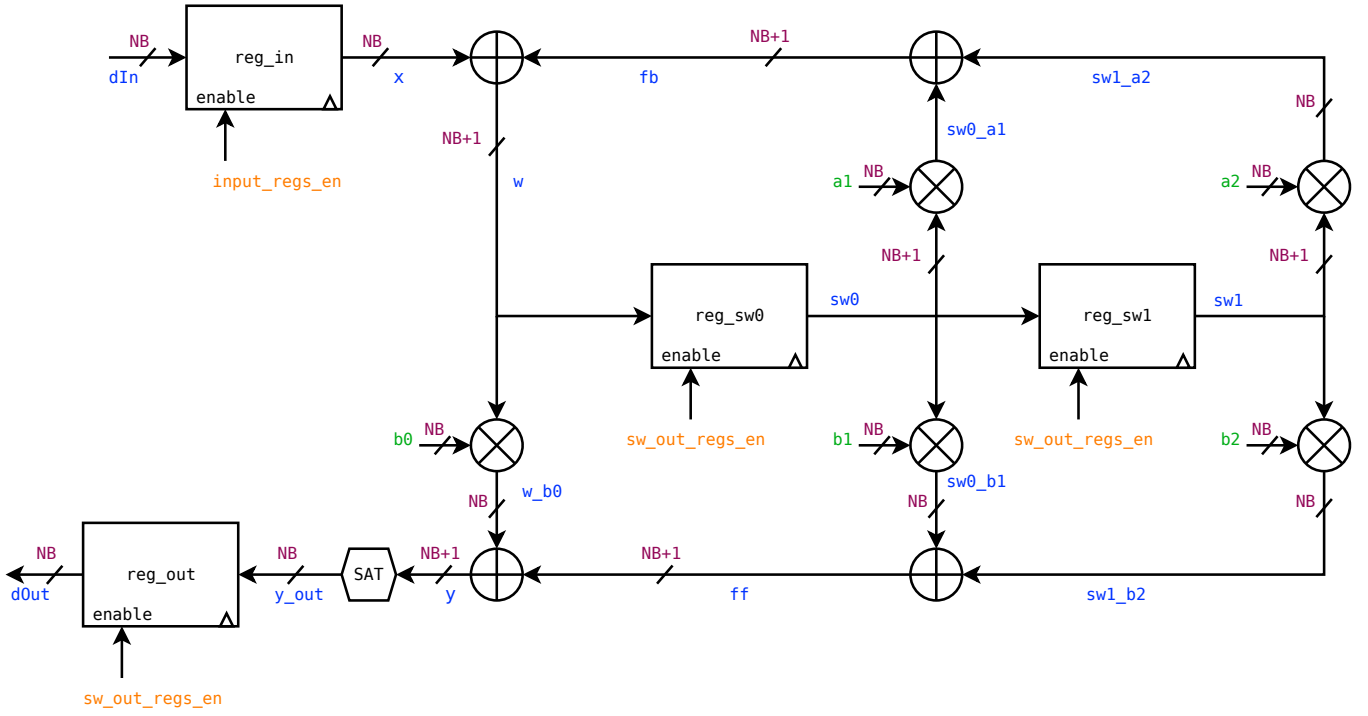


Figure 4: Datapath

- Blue signals are data signals
- Green signals are external coefficients
- Purple labels define the parallelism of the corresponding signal
- Orange signals are control signals coming from the Control Unit

3.1.2 Registers

In addition to the two core delay elements of the filter DFG, the complete datapath includes also border registers for each piece of data coming in or out of the architecture, specifically the input and output sample stream and all filter coefficients. Input registers for the latter are not actually shown in figure 4 to avoid too much visual clutter, but assume that each coefficient in green is actually the output of a register.

Each register has implicit clock and asynchronous reset signals, as well as individual enable signals. No synchronous clear is provided as it is of no use for the purposes of this design.

3.1.3 Arithmetic operators

As per specifications, arithmetic operators at this stage of the design are described as behavioral operators in VHDL. While no rounding is requested after addition, each multiplication must truncate back to the original number of bits, a VHDL package was written, containing a function called `multiplyAndRound` which performs potential sign extension of the operands, behavioral multiplication and truncation. In the same package, some constants and custom data types to make the design parametric were defined as well.

Addition is more easily implemented as simply the behavioral operator ‘+’, but the fact that addition can overflow and need an additional bit on the output posed the need for some considerations about internal parallelism. By performing an analysis on the specific values of filter coefficients and the range of data stream values, the conclusion was that the maximum internal parallelism that this specific architecture can need is:

$$n_b + 1 = 13 \text{ bit}$$

Contrary to the initial expectations however, the output value y was found to need $(n_b + 1)$ bits as well because of occasional overflow, while in the beginning it was expected to stay in the n_b range. Overflow on the output would of course cause very large errors in the result because of wrapping. So it was decided to implement a saturation approach which limits the error instead, by taking the maximum or minimum value allowed on n_b bits if the actual results is too large or too small to be represented correctly.

This decision led to the need of modifying the fixed-point C model as well, to provide the same results of the VHDL description to compare during simulation and testing.

3.2 Control unit

Given the simplicity of the design, it was initially thought that no explicit control unit would be needed. While it is actually possible to avoid it completely, some considerations about the ease of extending the design with subsequent transformations and pipelining (section 4) as well as the preference for modularity and clarity led to the decision to implement a simple control unit, which FSM is shown in figure 5.

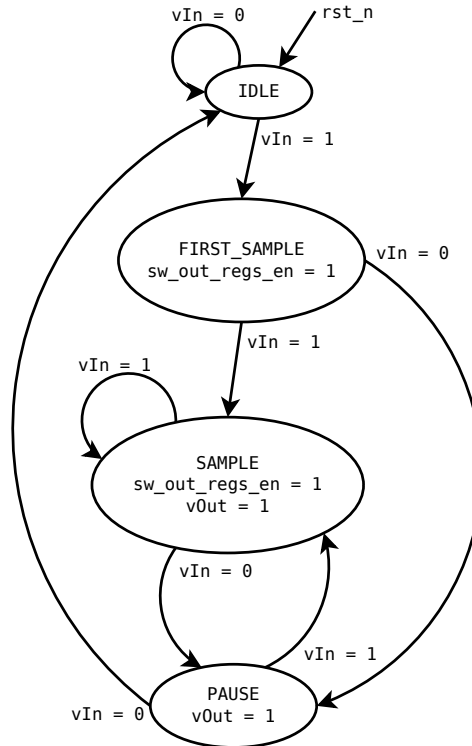


Figure 5: Filter control unit FSM

This control unit satisfies the specifications about the generation of the signal `vOut` to notify the presence of new data at the output and the need to take into account possible pauses during the input data stream. In practice, figure 6 shows the timing diagram that this FSM implements.

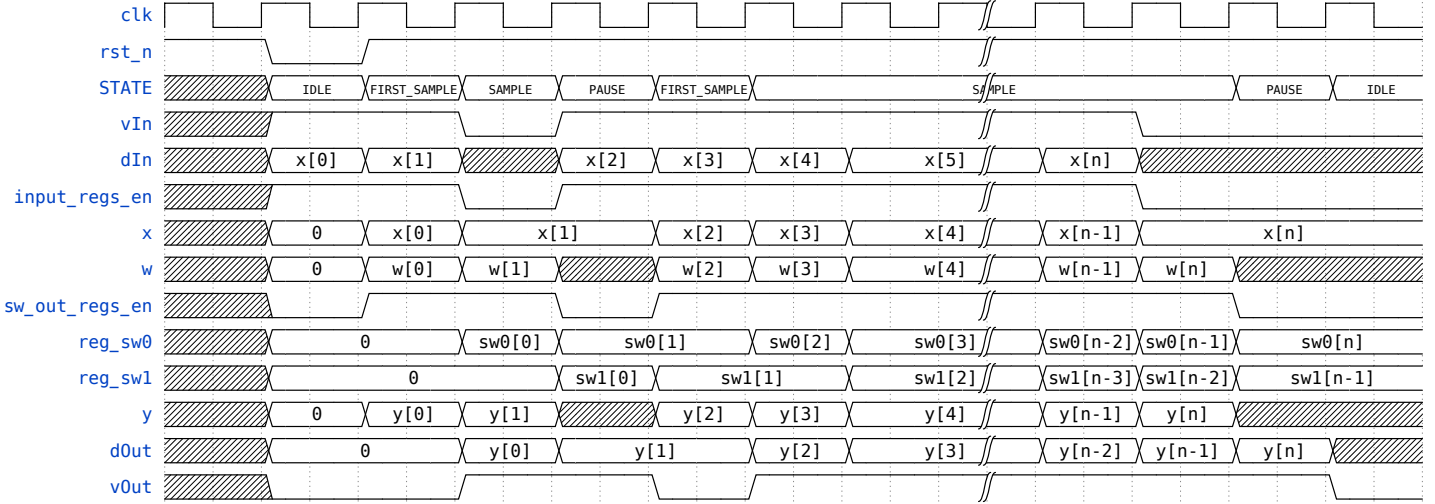


Figure 6: Timing diagram

3.3 Simulation

All simulations were performed using ModelSim both on our local machines and on the remote server, by means of a mixed-language testbench partly already provided in the course material. The procedure and tools described in the following sections apply to the simulation of all the RTL description as well as the netlists generated by other tools used for synthesis or place and route.

Each design or netlist has been simulated extensively multiple times using up to 100000 samples generated randomly or with special values, which would normally take around 20 minutes.

Regarding the simulation of the RTL description, no particular problems arose apart from the usual initial debugging and in the end the design worked as expected.

3.3.1 Testbench

The testbench is made up of several independent modules:

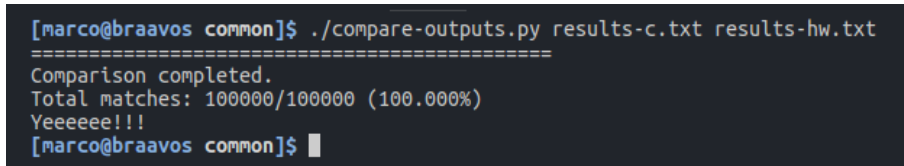
- A *clock generator* in charge of generating the initial reset signal and a periodic clock of specified period until an external `end_sim` signal arrives to notify the end of the simulation. This is useful to automatically stop the simulation process when all input data has been processed instead of fixing a specific time span a priori.
- A *data maker* which reads input data from a text file, converts it to a suitable format (**signed** in this case) and feeds it to the hardware description of the filter. It also allows to insert pauses in between samples to verify that such situation is handled correctly by the design. The number of pauses can be hard-coded or random, according to the value of a constant defined in the package.

- A *data sink* to read output data stream from the hardware block and write it to another text file.
- A top module written in Verilog (instead of VHDL as the previous ones) that simply connects the instance of the filter and all other testbench modules together. The need to describe this block in Verilog arises from the fact that synthesis and place and route tools generate Verilog netlists and are more comfortable using Verilog files.

3.3.2 Scripts

In order to try to automate as much as possible the simulation process, a number of scripts were developed, which can be found in the GitHub repository or in the material provided:

- **samples-generator.py** to generate a user-specified number of samples either random or special (meaning extremes and zero only).
- **auto-scp.sh** to automate the copy of source files and input samples on the server and results and netlists from the server.
- **compare-results.py** which compares the results of the simulation against the ones produced by the C model (figure 7).
- **auto-simulate.sh** which allows, by means of defining environmental variables, to choose whether to simulate locally or remotely and which design to simulate (RTL description, post-synthesis netlist or post-place and route netlist) and then performs several steps to automatically carry out the simulation, namely:
 - Generates samples using **samples-generator.py**.
 - Runs the C model on these samples.
 - Connects to the server (if the user has chose to run the simulation remotely) and copies the samples file.
 - Runs ModelSim CLI to perform the simulation.
 - Retrieves output files and compare the results.
- **sim-script.tcl** executed directly by ModelSim, which reads the environmental variables passed by **auto-simulate.sh** and issues the correct commands to the tool.



```
[marco@braavos common]$ ./compare-outputs.py results-c.txt results-hw.txt
=====
Comparison completed.
Total matches: 100000/100000 (100.000%)
Yeeeeee!!!
[marco@braavos common]$
```

Figure 7: Successful comparison

3.4 Synthesis

The synthesis of the design has been carried out using Synopsys Design Compiler, following the instructions given in the specifications. All the commands issued to

the tool were again put together inside a TCL script to ease the repetition of the process.

At first try, the comparison between the synthesized design and the original RTL description resulted in outputs being completely different. After some trial and error, which included confrontation with colleagues and Professors, the issue came out to be the issuing of asynchronous reset signal in the first state of the control unit to all registers. The solution was to remove those outputs from such state and instead wire up the reset of the register directly to the external reset signal, which rendered the initial state of the control unit an empty idle state, as seen in the final FSM of figure 5.

After successfully completing the synthesis and testing the correctness of the results from the netlist, we extracted two figures of merit: the first at the maximum achievable frequency, that is shown in table 1, and the second at a quarter of this frequency, shown in table 2

f_{max}	352.11 MHz
Area	4705.51 μm^2
Dynamic power	86.13 μW
Static power	16.53 μW

Table 1: Figures of merit post-synthesis at $f_{max} = 352.11$ MHz.

f	88.0 MHz
Area	3915.25 μm^2
Dynamic power	84.41 μW
Static power	966.11 nW

Table 2: Figures of merit post-synthesis at $f_{max}/4 = 88.0$ MHz.

These results show a reduction of 94% on the leakage power, that has now almost no impact on the total power consumption of the chip.

3.5 Place & route

The place and route phase was carried out using Cadence Innovus and the instructions provided in the lab track. As always, a script was put together to collect all commands issued to the tool, even if it was more difficult in this case as the GUI was heavily used.

All the place and route was done using a clock frequency equal to a quarter of the maximum one found during synthesis. At the end of the process, three layers of metal interconnections were used. The results are shown in table 3.

f	88.0 MHz
Area	3904.3 μm^2
Dynamic power	1.381 mW
Static power	78.7 μW

Table 3: Figures of merit post-place & route. Both area and power were computed at $f_{max}/4 = 88.0$ MHz as per laboratory guidelines.

3.6 Remarks

By comparing tables 2 and 3, it is possible to see how the early, post-synthesis estimation of the chip area is quite consistent with the place and route results. However, the reported dynamic and static power are quite different. In fact, the dynamic power is about 16 times higher, while the static power (leakage) is about 80 times higher, even worse than the expected leakage power of the same chip working at f_{max} . The explanation can be found in the fact that the synthesis doesn't take into account interconnections and power rings. The introduced capacitance might be a reason for the dynamic power to be increased. On the other hand, the huge increase in leakage power is something we are not able to explain properly. Innovus reported high leakage in the combinational logic (87% of total static power). Maybe this is related to the the necessity of driving the metal interconnections.

Comparing our results with those from other groups, we noticed that, in general, FIR filters brought to better results in terms of consistency between post-synthesis and post-place and route power reports. This might be explained by considering that FIR filter are usually bigger (higher order), to obtain a frequency response as steep as the one from an IIR filter. For this reason, the introduced capacitance and logic required to drive the interconnections in a FIR filter might have a lower impact on the overall power performance of the chip with respect to an IIR filter, that contains fewer combinational and sequential nodes.

4 Look-ahead transform

The next step consists of improving the previous basic design by means of formal transformations and pipelining. Being the case in exam a IIR filter, the method implemented is the so-called *look-ahead* transform, which consists of expanding the difference equation of the original filter (see 2) by making explicit the term $y[n-1]$ as follows:

$$y[n-1] = b_0x[n-1] + b_1x[n-2] + b_2x[n-3] - a_1y[n-2] - a_2y[n-3]$$

Resulting in this final form of the difference equation:

$$\begin{aligned} \rightarrow y[n] &= \rightarrow b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1(b_0x[n-1] + b_1x \rightarrow n-2] + \\ &\quad \rightarrow + b_2x[n-3] - a_1y[n-2] - a_2y[n-3]) - a_{\rightarrow}[n-2] \\ &= b_0x[n] + (b_1 - a_1b_0)x[n-1] + (b_2 - a_2b_1)x[n-2] + \\ &\quad - a_1b_2x[n-3] + (a_1^2 - a_2)y[n-2] + (a_1a_2)y[n-3] \end{aligned} \tag{3}$$

The green highlights the fact that new coefficients different from the originals arise from the look-ahead transformation, which will necessitate adapting the parallelism and the hardware component to these new values. Figure 8 shows the modified DFG that implements equation 3.

4.1 Loop bound analysis

The justification for performing this modification on the original DFG lies in the analysis of the loop bound constrain and the critical path of the architecture. Figure 9 shows the loop bound (in blue) and the critical path (in green) on the original DFG, which values are respectively:

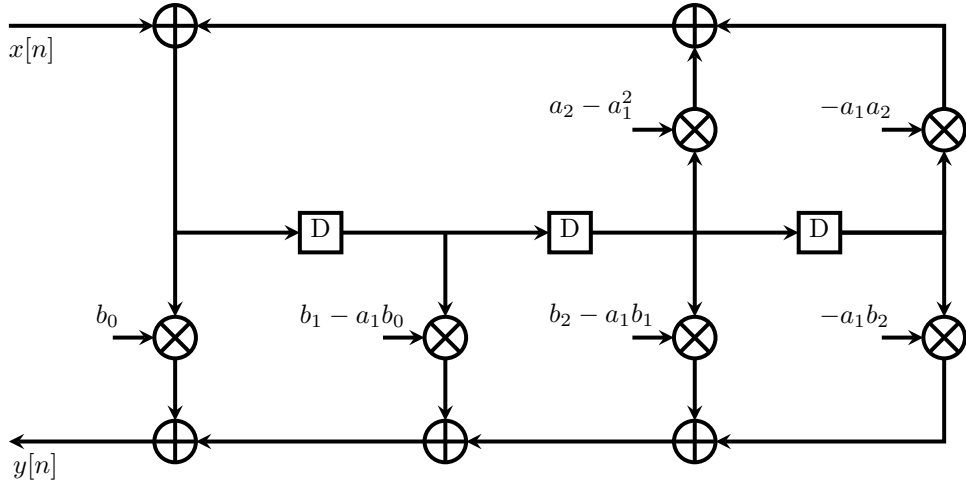


Figure 8: Look-ahead transformed DFG

- $T_{\infty} = 2T_a + T_m$
- $T_{cp} = 3T_a + 2T_m$

Being that $T_{\infty} < T_{cp}$, there is theoretically room for improvement even in the base architecture. In fact it is quite easy to see that pipelining could be applied in the FIR part of the DFG.

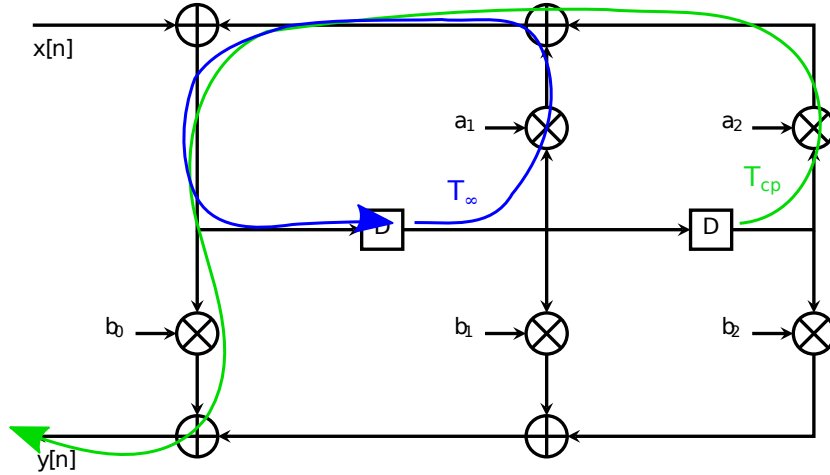


Figure 9: Loop bound and critical path of the base DFG

However, it makes more sense to apply improvements only on the modified DFG as the application of the look-ahead transform allows to halve the loop bound, as shown in figure 10, by introducing an additional delay element in the longest loop.

4.2 Pipelining and retiming

Figure 10 shows also that the critical path remains exactly the same after the transformation, which means that the look-ahead transform on its own does not lead to any improvement at all, but simply enables speed-up to be achieved in other ways.

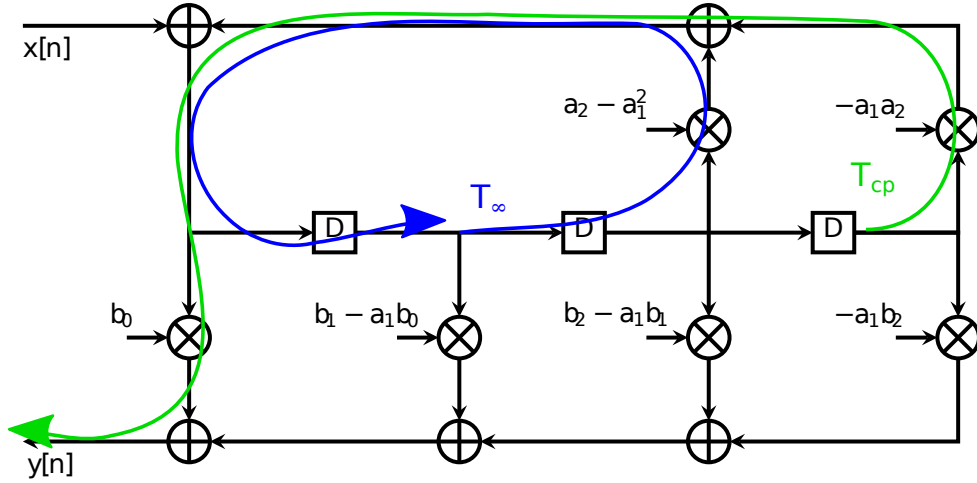


Figure 10: Loop bound and critical path of the look-ahead DFG

In this particular case two different improvements can be performed on two different parts of the DFG:

- **Pipelining:** in the feed-forward (or FIR) part of the filter two feed-forward cutsets can be identified and thus two levels of pipelining can be inserted. This reduces the critical path by excluding the last multiplier/adder couple from the total.
- **Retiming:** in the feedback (or IIR) part instead no feed-forward cutset can be identified (obviously), but retiming can be performed by splitting the second and third register in the delay line along the two forking branches and moving the copy in the IIR part after the two multipliers to exclude them from the critical path as well.

These transformations lead to the final DFG of figure 11, where of course the loop bound is unchanged but the critical is reduced to the delay of a single multiplier T_m ¹.

Note that the loop bound is still smaller than the critical path, but no further improvement can be achieved without fine-grained pipelining inside the arithmetic operators, which is beyond the scope of this laboratory experience.

4.3 Datapath

Figure 12 shows the detailed datapath of the new architecture. The color scheme used to illustrate signals is the same used before.

4.3.1 Pipe registers

Pipeline and retiming registers differ from the others as they do not have an *enable* signal. This way they are “free running” and always sample after the initial reset. This choice was made to keep the design and the control as simple as before.

¹This is valid only if one multiplier is slower than three adders, but this is probably the case.

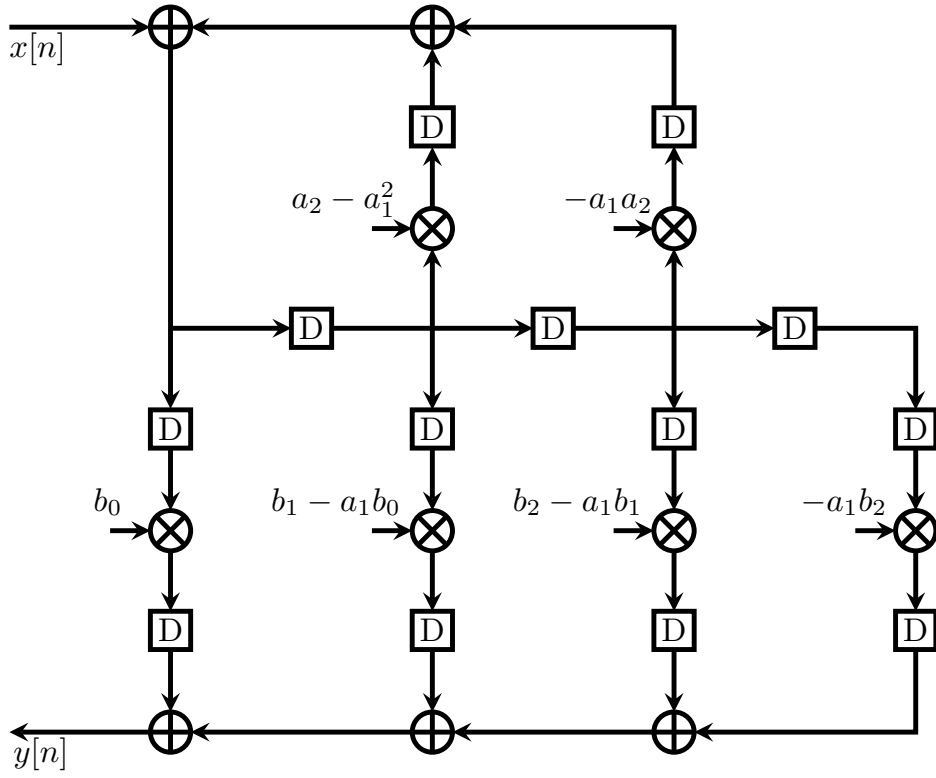


Figure 11: Final pipelined and retimed DFG

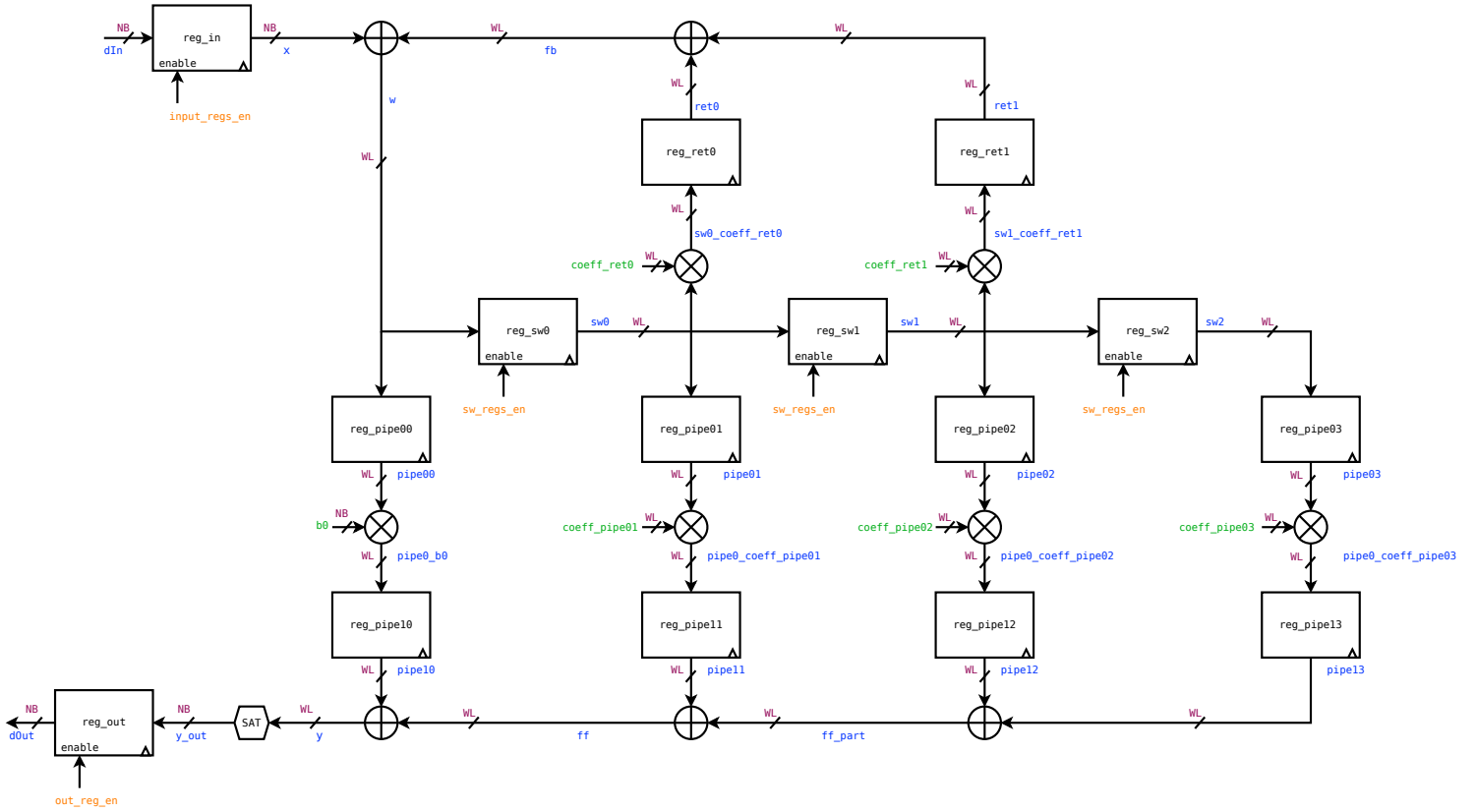


Figure 12: Look-ahead datapath

4.3.2 Parallelism

Having different coefficients expressed as sums and products of the original ones, the whole internal parallelism had to be recomputed.

The maximum number of bits that these new coefficients require was found to be 22 bit ($Q1.21$ format). Following the reasoning done for the original architecture, the multiplication function inside the VHDL package was modified to take care of the different parallelism and all the internal computations were chosen to be performed on 23 bits ($Q2.21$ format) to account for potential addition overflow. This number of bits was called *word length* being the one used in all internal signals and is the one represented by the label WL in figure 12.

One last point that needed care was the alignment of external data streams (as well as the only coefficient that remained the same, b_0) which continue to be expressed in $Q1.11$ format, and thus require proper fixed point alignment and extension in both directions before getting summed with internal signals.

As usual, the output is saturated in case of internal overflow.

4.4 Control unit

The great advantage of such pipelining process is that the control unit remains exactly the same as before, nothing at all needs to be modified.

The only change resides in the top level design, where a couple of signals coming from the control unit (namely, the output register enable and the data valid signal) have to be delayed of a number of clock cycles equal to the depth of the pipeline, to account for the additional latency of the output.

This was very easy to implement, so that additionally the possibility to keep parametric the depth of the pipeline in the VHDL code was provided.

4.5 Synthesis

Table 4 and table 5 show the results obtained by the synthesis process on the look-ahead architecture. The procedure used for this as well as for the extensive simulation is exactly the same as before, so no additional detail have to be pointed out.

f_{max}	689.7 MHz
Area	18 839.45 μm^2
Dynamic power	3.028 mW
Static power	57.55 μW

Table 4: Results of synthesis for the look-ahead architecture at $f_{max} = 689.7$ MHz.

f	172.41 MHz
Area	15 337.56 μm^2
Dynamic power	159.26 μW
Static power	1.80 μW

Table 5: Results of synthesis for the look-ahead architecture at $f = f_{max}/4 = 172.41$ MHz.

4.5.1 Remarks

Comparing these results, we can see how the dynamic power is reduced by 94% when working at a quarter of the maximum frequency. Notice how this is not consistent

with the theoretical expectations. The dynamic power is a direct function of the clock frequency:

$$P_{dyn} \propto f \Rightarrow P_{dyn}(f/4) \approx P_{dyn}(f) = \frac{1}{4}P_{dyn}(f)$$

That is $1 - 1/4 = 75\%$ lower. However, besides the reduction of the clock frequency, also the reduction in the number of logic elements and the type of arithmetic components must be considered too. In fact, the load capacitance of the entire circuit depends on these. So, the total dynamic power is reduced more than expected when considering only the reduction of the frequency.

Regarding the static leakage power, the reduction is consistent with the one obtained for the base architecture.

4.6 Place & route

Again, the procedures followed to perform the place and rout and the related simulations on the lookahead implementation are the same as for the base architecture. Table 6 shows the obtained results. These were computed at $f = f_{max}/4 = 172.41$ MHz.

f	172.41 MHz
Area	15 231.4 μm^2
Dynamic power	4.511 mW
Static power	323.6 μW

Table 6: Results of place and route for the look-ahead architecture at $f = f_{max}/4 = 172.41$ MHz.

4.6.1 Remarks

Comparing table 5 and 6, the conclusion are very similar to the ones already expressed for the base architecture. The post-synthesis area estimation was quite accurate, while the dynamic power has increased by $28\times$ after the place and route analysis. The increase in the base architecture was $16\times$ higher. The look-ahead architecture has more logic components with respect to the base one, so if the thesis of section 3.6 was correct, the increase after place and route should be lower than before. This could be explained with the different components being instantiated during the synthesis. Also static power is, again, very different from the expected one from the synthesis process. However, the order of magnitude of the place and route results is the same as the ones from the first implementation. This supports the thesis expressed in section 3.6, that can be applied here as well. Of course, the datapath of the lookahead implementation is more complex than the original one, and this explains the increase in static and dynamic power.