02GQCOQ - Integrated Systems Architecture

A.A. 2020/2021

Prof. Maurizio Martina
Prof. Guido Masera

LAB1

Design and implementation of a digital filter

Group 18

https://github.com/wackozz/isa/tree/main/lab1

Federica Bongo	s292395
Stefano Rizzello	s288013
Francesco Vacca	s279895

Contents

1 Filter design

1.1 Introduction and specifications

The goal of this lab is to design and implement a fixed point **IIR** (infinite impulse response) digital filter. The applied design methodology is to obtain the fastest minumum delay.

A general expression for this type of filter is reported in Equation 1.

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] - \sum_{j=1}^{N} a_i y[n-j] \tag{1}$$

Specifications The provided specifications are resumed in Table 1, where the parameters N and n_b were calculated according to the reported instructions.

Parameter	Description	Value
f_c	Cutoff frequency	2 kHz
f_s	Sampling frequency	10 kHz
N	Order of the filter	2
n_b	Number of bits $[n_b]$	9
THD	Total Harmonic Distortion	-30dB

Table 1: Specifications for the IIR Filter.

The filter had to be implemented in direct form II, minimizing the filter's memory elements. The IIR formula (Equation 1) can be rewritten for the specific implementation as the system of equations reported in Equation 2. A DFD for the device has been derived from the equations, as reported in Figure 1.

$$\begin{cases} w[n] = x[n] - a_1 w[n-1] - a_2 w[n-2] \\ y[n] = b_0 w[n] + b_1 w[n-1] + b_2 w[n-2] \end{cases} \tag{2}$$

1.2 Filter design

The design of the filter was done using the provided MATLAB script `my_iir_filter.m`. The script creates an input data vector consisting in two superimposed sinusoidal waves sampled at 10 kHz. Given that the cutoff frequency for the filter is $f_c = 2$ kHz, the first sinusoid falls in-band at $f_1 = 0.5$ kHz, while the other one is out of band at $f_2 = 4.5$ kHz.

The filter is then applied to the input data and its coefficients (2) are printed to screen. Indeed, the script converts both input and output vector from floating point to Q1.8 and saves them into dedicated files. Those files have been used to build and verify the low-level software model developed in C.

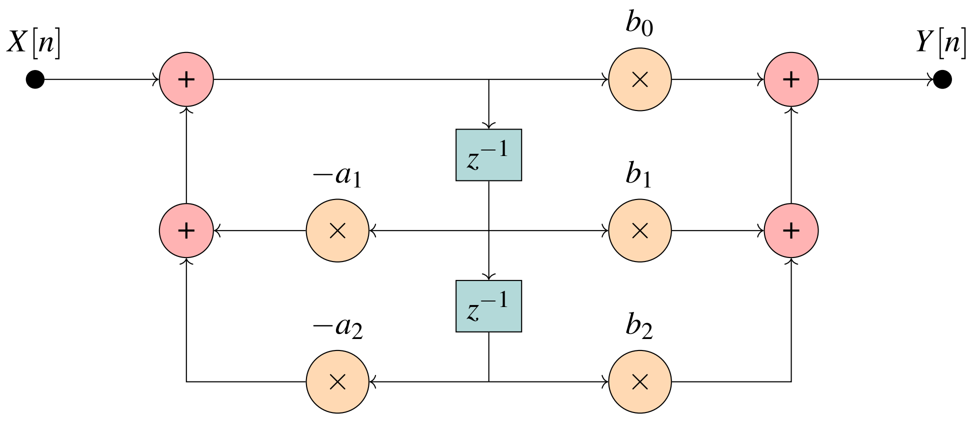

Figure 1: Data flow diagram for the IIR Filter in DF2, N=2

Coefficient	**Value (Floating point)**	**Value (Q1.8)**
b_0	0.203	52
a_1	-0.371	-95
b_1	0.410	105
a_2	0.195	50
b_2	0.203	52

Table 2: Filter coefficients

1.3 Software model

The software model was written in C starting from the provided code in `myfilterII.c`. The model has been used as a reference to verify the proper implementation of the HDL and to validate the output results.

Code changes Small changes to the code have been made in order to better mimic the behavior of the hardware. Mostly, the signs of the a_i coefficients were inverted in order to avoid subtractions in the C code, to reflect what has been implemented in the HDL. Input, output and coefficient have all been opportunely shifted in order to be compliant to the chosen internal numeric format.

Some `printf()` have been added in order to display the content of the buffers and the intermediate results of sums and multiplications; those data were particularly helpful during the verification process. The code for the modified `myfilter()` function is reported below.

```
#define N 2   /// order of the filter
#define NB 7 // number of bits

const int SH = 9 - NB;
const int b0 = (52>>SH);          /// coefficient b0
const int b[N] = {(105>>SH), (52>>SH)};  /// b array
const int a[N] = {(95>>SH), (-50>>SH)}; /// a array
[...]
 //// myfilter():
 [...]
  /// compute feed-back and feed-forward
  fb = 0;
  ff = 0;
  for (i = 0; i < N; i++) {
    printf("\t q[%d]*a[%d] =%d (%d)\n", i + 1, i + 1, (-sw[i] * a[i]),
           (sw[i] * a[i]) >> (NB - 1));

    printf("\t q[%d]*b[%d] =%d (%d)\n", i + 1, i + 1, (sw[i] * b[i]),
           (sw[i] * b[i]) >> (NB - 1));

    fb += (sw[i] * a[i]) >>(NB-1);
    tmp_fb = (sw[i] * a[i]);
    ff += (sw[i] * b[i]) >> (NB - 1);
  }
  /// compute intermediate value (w) and output sample

  w = (x>>SH) + fb;
  printf("\t tmpa=%d \n", w);
  printf("\t tmpa*b0=%d (%d) \n", w * b0,(w * b0) >> (NB - 1));
  y = (w * b0) >> (NB - 1);
  y += ff;
  printf("\t tmpb=%d \n", y);

  /// update the shift register
  for (i = N - 1; i > 0; i--) sw[i] = sw[i - 1];
  sw[0] = w;

  return y;
```

Output results and THD The output results of the C model are very close to the ones provided by the MATLAB script. The small differences between the two are due to the floating point arithmetic used in MATLAB for intermediate calculation, with discretization only applied to the end result. An example of comparison between the output of the models is reported in appendix, Table 10.

The total harmonic distortion factor (THD) of the filter was calculated in MATLAB using the homonymous function from the DSP package.

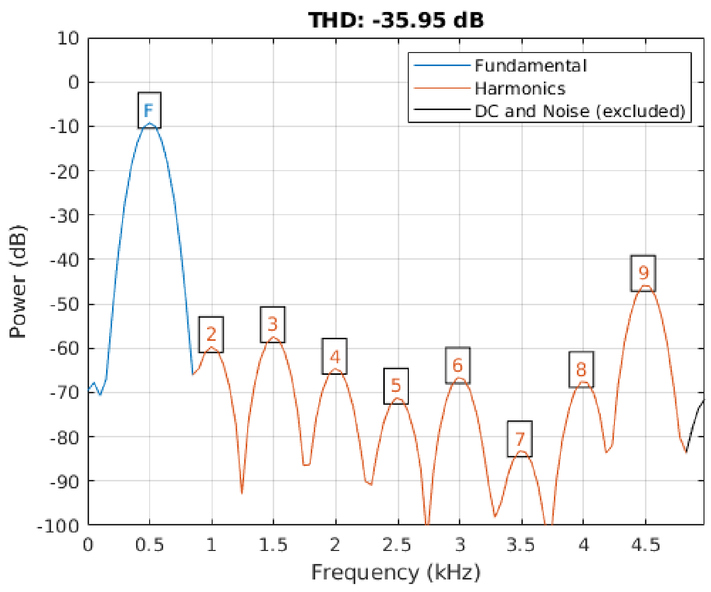

Figure 2: Total Harmonic Distortion, $n_b = 9$

```
thd(output_data,10e3,9);
```

The computation of the THD ranges from the mininum to the maximum frequency of the input data. As showed in Figure 2 and Equation 3, the factor with $n_b = 9$ is compliant to the specification of THD_{max}.

$$THD_{9bit} = -35.95\,\text{dB} \le -30.00\,\text{dB} \tag{3}$$

1.3.1 Data width optimization

In order to minimize area and delay for both arithmetic blocks and registers, it has been decided to reduce the internal numeric resolution (n_b) of the filter until the THD remained compliant to the THD_{max} specification. From the tests, it has been found that the value $n_b = 7$ (Q1.6) was enough to get a $THD \le THD_{max}$ (see Figure 3 and Equation 4).

$$THD_{7bit} = -30.94\,\text{dB} \le -30.00\,\text{dB} \tag{4}$$

2 Hardware description and implementation

The hardware model of the filter was described in VHDL, starting from the DFD illustrated in Figure 1. The top entity pinout is represented in Figure 4, while the detailed

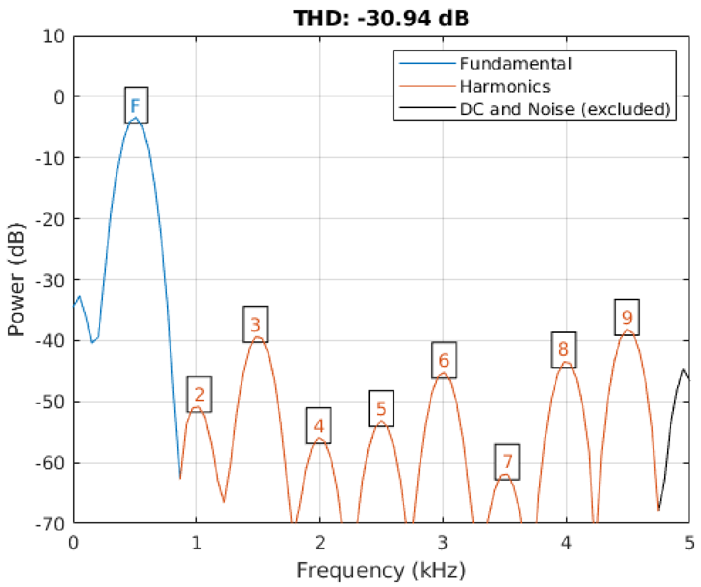

Figure 3: Total Harmonic Distortion, $n_b = 7$

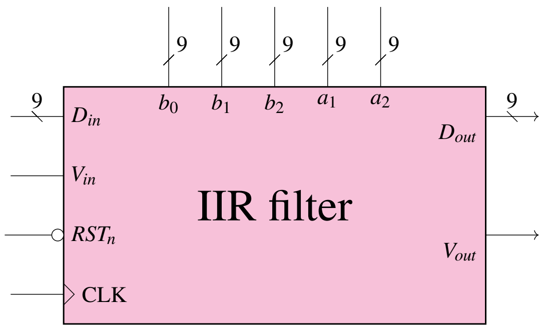

Figure 4: Top entity view (pinout)

internal schematic is reported in Figure 5. Input and coefficients are given to the filter in format Q1.8 and internally shifted to Q2.6. Output has been extended to Q2.7 as requested by specification (D_{out} = 9bit).

$$w[0] = x[0] = -1 \qquad \text{(First clock cycle)}$$
$$w[1] = x[0] - a_1 w[0] = -1.371 \qquad \text{[OWFL](Second clock cycle)}$$

Adders, coefficient sign In order to save the need of additional hardware for the sign conversion or implement subtractors in the design, the coefficients a_1 and a_2 are directly fed into the filter inverted in sign ($-a_1$, $-a_2$).

Parallelism and overflow As stated in the previous section of the report, internally the filter can work using as low as 6 bits of fractional resolution. If the input value is $x \approx |1|$,

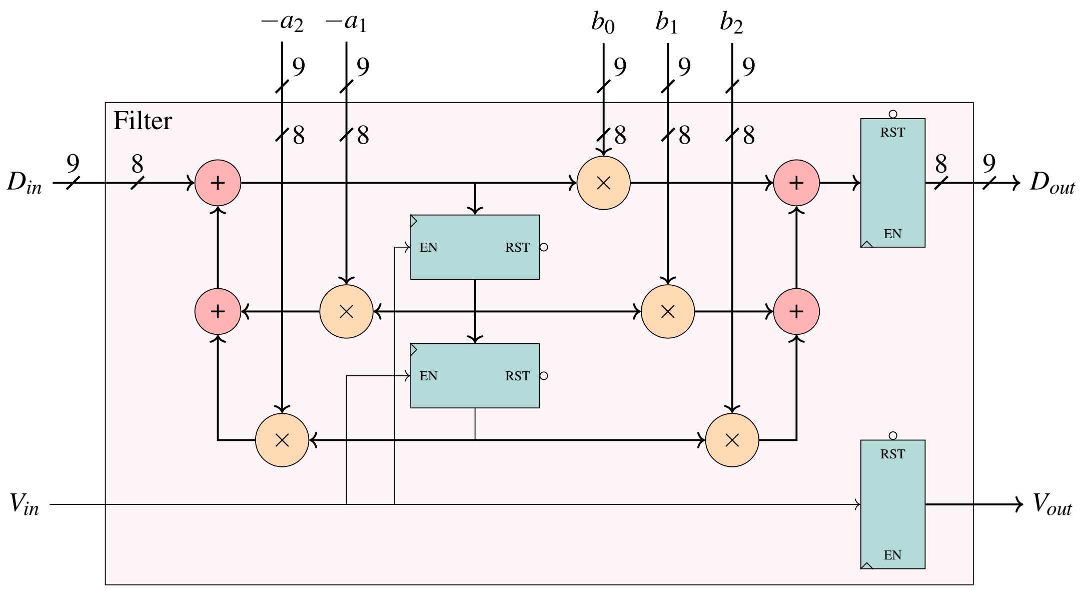

Figure 5: Top entity view (internal schematic)

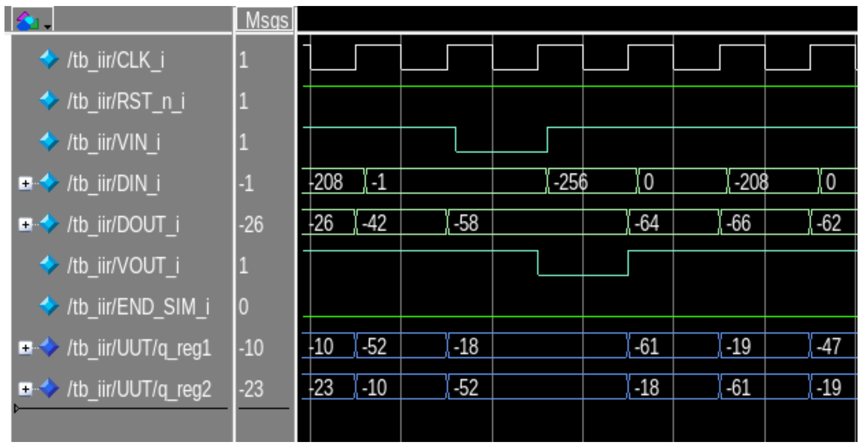

Figure 6: ModelSim simulation showing data validation.

the first intermediate sums can exceed Q1.6 range of values. Assuming $x[n]$ always equal to -1, the computation for the second equation of the system (Equation 2) becomes:

Given this result, the internal parallelism of the filter has been set to Q2.6 (8bit). In this way, it can be proved that neither output or any intermediate results will generate overflow.

Input validation From the received directives, the output should be valid only when the signal V_{in} is high and both V_{out} and D_{out} must be sampled by registers before output.

In order to implement this behavior, it has been decided to use V_{in} as the enable signal for every register, freezing the filter internal buffers and the output register when the signal is low. V_{in} is sampled by a FF and its value is directly brought in output with a delay of one clock cycle as V_{out}. The behavior of the filter is showed in Figure 6

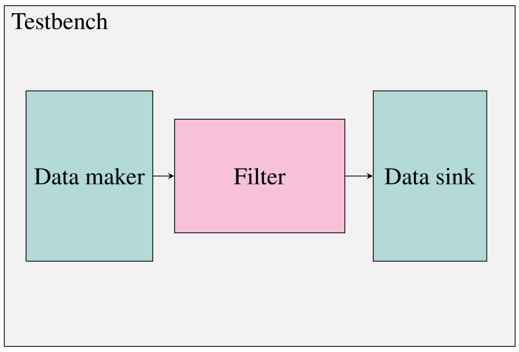

Figure 7: Testbench structure

Verification The correctness of the design has been verified via ModelSim, using the testbench structure showed in Figure 7. The filter is fed by a data generator block, which provides both cofficients and input data. The output samples of the filter are then collected by the data sink block and printed to a file. The RTL model was validated checking that the output samples, the content of the buffers and the intermediate arithmetic operations were exactly the same as the ones of the C model. TCL scripting was adopted to automate the simulation steps.

3 Synthesis

The synthesis process was carried on using Synopsys' Design Compiler. The minimum clock period the design can achieve is equal to 2.99 ns, with a critical path T_{cp} equal to:

$$T_{cp} \approx 2T_m + 2T_a = 2.82\,\text{ns}$$

To automate the task of getting the minimum clock period, a custom python/tcl script was coded. Starting from $T_ck = 0$, the script iteratively re-synthesized the RTL, increasing each time the clock value until a zero slack value was found.

The summary of the `report_area` command output is reported in Table 3. After the synthesis, the produced netlist has been validated through modelsim.

3.1 Power estimation

After the synthesis, it is possible estimate the power consumption of the circuit through backannotation. To do so, the Synposys generated `.sdf` files and the new verilog netlist are supplied to ModelSim to run a new simulation.

A *.vcd* file is obtained as result, which contains the nodes activity. This file is converted to a *.saif* using the `vcd2saif` utility, then fed back to Design Compiler in order to generate a power estimation from the netlist. The results of the `report_power` command output is summarized in Table 4.

Number of	T_{ck} =2.99 ns	T_{ck} =11.96 ns
Ports	383	383
Nets	1550	1300
Cells	1037	741
Combinational cells	993	697
Sequential cells	25	25
Buf/inv	113	96
References	14	13

Area [μm^2]	T_{ck} =2.99 ns	T_{ck} =11.96 ns
Combinational	1537	1358
Buf/Inv	65	52
Noncombinational	133	133
Total	1670	$\boxed{1491}$

Table 3: Design compiler area report for $T_{c,min}$ and $4T_{c,min}$

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Register	17	5.4	2.2	25	12.22%
Combinational	82	73	26	181	87.78%
Total	99 μW	78 μW	28 μW	$\boxed{206\,\mu W}$	

Table 4: Design compiler power report. $4T_{c,min}$.

4 Placing and routing

Placing and routing of the filter was conducted with Cadence Innovus.

To complete the design flow, the following steps have been taken, following the supplied guideline:

1. Importing the design
2. Floor planning
3. Power planning and routing
4. Cell placing
5. Signal routing
6. Timing and design analysis

A snapshot of the circuit, taken after the last step, is reported in Figure 8. The total area, accounting for the interconnesions, has been found to be 1478 μm^2.

Power estimation The steps illustrated in subsection 3.1 have been repeated after the P&R procedure. This time, there was no need to use `vcd2saif` because Innovus direclty reads *.vcd* files. After the procedure, the new netlist has been validated through modelsim.

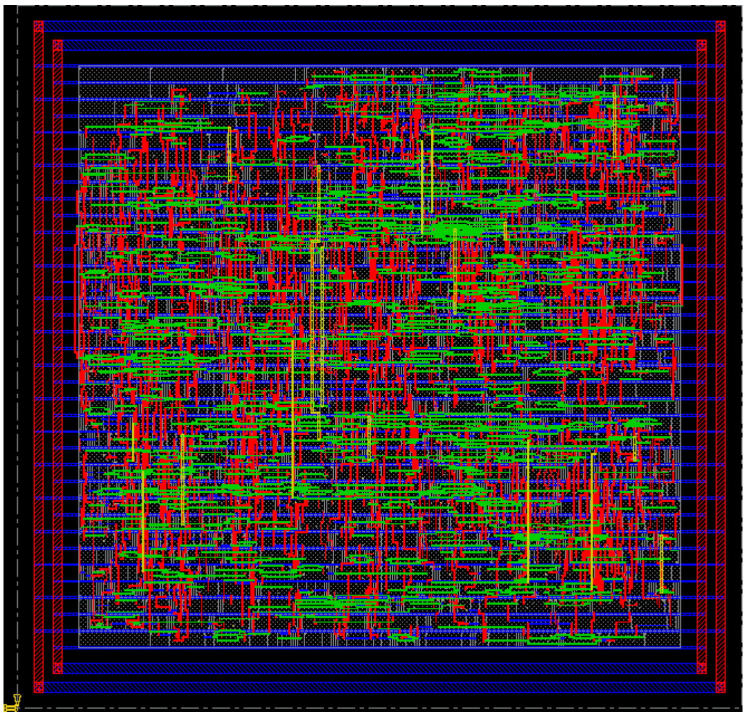

Figure 8: Cadence Innovus' post-P&R snapshot

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Sequential	0.028	0.013	0.0021	0.043	9%
Combinational	0.22	0.19	0.025	0.43	91%
Total	0.25 mW	0.20 mW	0.028 mW	$\boxed{0.48\,\text{mW}}$	100%

Table 5: Innovus power report. $T_c = 4T_{c,min}$

A summary of the power report generated by Innovus is reported in Table 5.

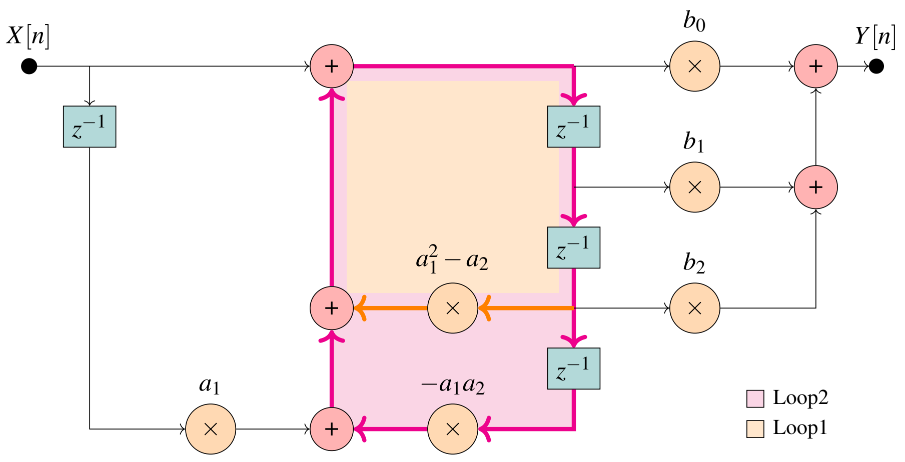

Figure 9: IIR LookAhead architecture

5 Look-ahead optimization

5.1 Implementation

It is possible to improve the performance of the IIR filter by applying the Look-Ahead technique. The goal is to unroll the loop equation, obtaining a new DFD where universal techniques can be applied. Starting from the general DF2 equation reported in Equation 2, it is possible to calculate the term w[n-1].

$$w[n-1] = x[n-1] - a_1 w[n-2] - a_2 w[n-3] \tag{5}$$

The feedforward part of the filter can be improved using pipeline, so there's no need to substitute (Equation 5) in the upper equation of the system. Replacing the term only in the lower one brings to the following system:

$$\begin{cases} y[n] = b_0 w[n] + b_1 w[n-1] + b_2 w[n-2] \\ w[n] = x[n] - a_1 x[n-1] + (a_1^2 - a_2) w[n-2] + a_1 a_2 w[n-3] - a_2 w[n-2] \end{cases} \tag{6}$$

A DFD for the new equation is reported in Figure 9.

Optimizations To improve the critical path, three levels of pipes have been applied to the feedforward part, putting one row of registers before the feedback loop and two rows after, framing the multipliers (see final implementation in Figure 10).

Then, it has been proceed to calculate the maximum loop bound of the filter T_∞ and the critical path T_{cp}, in order to optimize the feedback part.

$$T_\infty = max\{T_{L,i}\} = \left\{ \frac{T_i}{W_i} \right\} \tag{7}$$

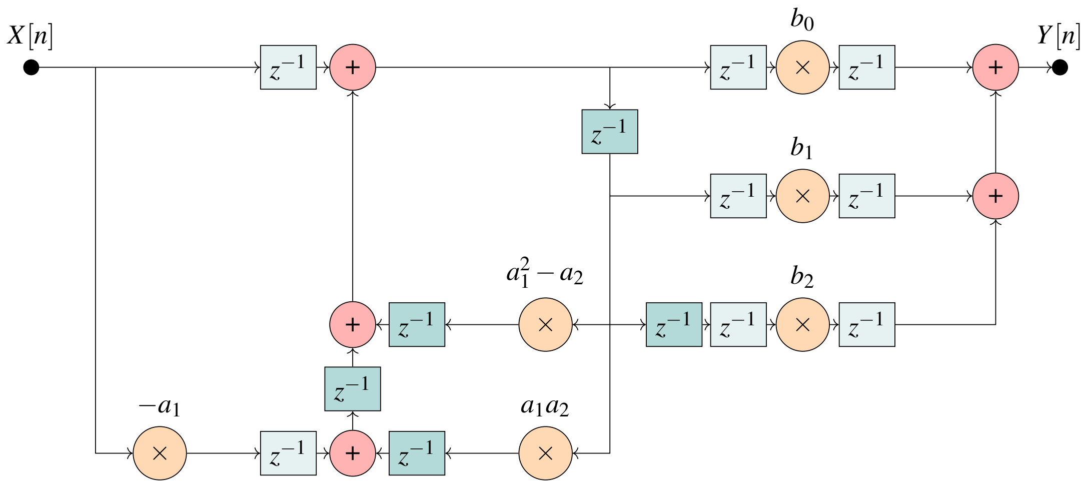

Figure 10: IIR LookAhead architecture, optimized with pipeline and retiming

In the Equation 7, T_i represents the total delay along the i-th loop path, while W_i is the number of registers present in the same path. In this case, two loops L_1 and L_2 can be identified from the DFD.

$$\begin{aligned} T_{L1} &= \frac{2T_a + T_m}{2}; \\ T_{L2} &= \frac{3T_a + T_m}{3}; \\ \implies T_\infty = T_{L1} &= \frac{2T_a + T_m}{2} \end{aligned}$$

The critical path for the loop, assuming the insertion of pipe registers, is equal to:

$$T_{cp} = 3T_a + T_m \tag{8}$$

The value of T_{cp} is greater than T_∞, so there is room for improving the DFD using universal techniques. Given that it is not possible to place pipeline registers on the feedback part, it has been opted to apply the retiming technique, moving the registers across the paths. The final architecture with optimizations is reported in Figure 10.

As it can be seen, the method has improved the T_{cp}, which now consists in a single multiplier delay.

$$T_{cp} = T_m \tag{9}$$

The value of T_{cp} is still a little higher than T_∞, but it is possibly the best that can be achieved with the technique. A better result may be achieved breaking the combinatory path inside the arithmetic blocks, but in this laboratory it was mandatory that the blocks were automatically implemented through $+$ and $\times$ operator from `ieee.numeric`.

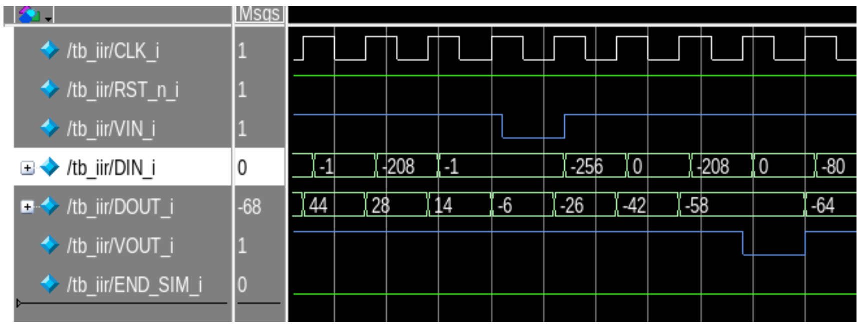

Figure 11: ModelSim simulation showing data validation for the optimized implementation.

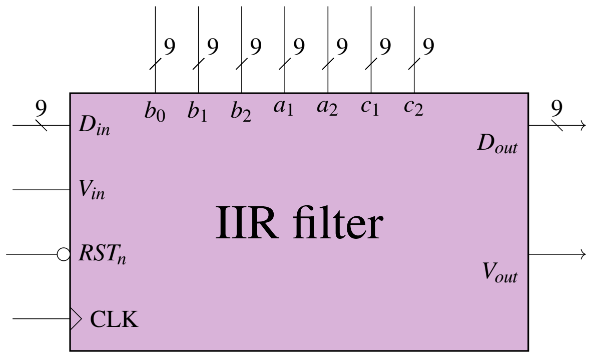

Figure 12: Top entity for IIR look-ahead filter

RTL design Both the optimized and the not optimized architectures have been implemented starting on the previous filter VHDL description and the DFDs in Figure 9 and Figure 10. The top entity for the implementations is reported in Figure 12

Implementation edits With regard to the scheme in Figure 5, two new coefficients in input, an extra register and one more multiplier are needed in order for the filter to work.

Moreover, in the optimized version, V_{in} is brought in output through a cascade of four flip-flops, delaying the validity of the output data of three T_{ck} to account for the pipeline insertion. In order to get the proper behavior, $V_{in}[n]$ is used as enable for the first level of pipe registers and the loop registers, $V_{in}[n-1]$ for the second level of pipe and $V_{in}[n-2]$ for the third one; indeed $V_{in}[n-3]$ enables the output register. The correctness of the validation behavior is showed in Figure 11.

Considerations on parallelism and overflow remain unaltered from the initial design.

Verification and accuracy The hardware descriptions were again validated comparing the output results with the ones obtained with the fixed point C model.

IIR DF2	**LookAhead**	**Error**
-66	-68	≈ 0.016
32	30	≈ 0.016

Table 6: IIR DF2 vs look-ahead output

The filter accuracy is now slighty lower than before,because the two new coefficients

$$c_2 = -a_1^2 + a_2$$
$$c_3 = -a_1 a_2$$

are represented with the same resolution as the other ones, even though they would need double resolution to be properly represented. From the simulation, these numerical approximations will reflect in output only for two output values, as showed in Table 6.

Synthesis As before, the architectures have been synthesized on `Synopsys`. The minimum clock period has been found to be 3.05 ns for the not optimized version and 1.72 ns for the optimized one. The measured critical path delay for the two architecture are:

$$T_{cp} \approx 2T_a + T_m = 2.95\,\text{ns}; \tag{10}$$
$$T_{cp,opt} \approx T_m = 1.62\,\text{ns}; \tag{11}$$

Area reports for the architectures are reported in Table 7, while power estimation, done after the netlist validation, are reported in Table 8.

Place and route Placing and routing procedure has been made for both architectures The total area has been found to be 1866 μm^2 for the non-optimized version and 2246 μm^2 for the optimized one. At the end of the operations, after having validated the new netlist, and the power consumption has been estimated as done for the previous implementation. The results are reported in Table 9.

6 Conclusions

The developed filter work as stated by specifications, and the design goal of reducing the critical path has been achieved.

The final look-ahead version is able to achieve a maximum clock period which is $\approx 43\%$ lower than the one in the initial implementation. The cost for this improvement is a rise of $\approx 56\%$ in power consumption and of $\approx 52\%$ in area (post-placing).

The time delay of a multiplier block (T_m) is much higher than the one of an adder (T_a). For this reason, the main improvement in terms of performance was obtained breaking combinational paths that contained more than one multiplier.

Number of	T_{ck} =3.05 ns	T_{ck} =12.2 ns
Ports	486	486
Nets	1954	1638
Cells	1303	928
Combinational cells	1238	863
Sequential cells	41	41
Buf/inv	148	118
References	18	17
Area [µm^2]	T_{ck} =3.05 ns	T_{ck} =12.2 ns
Combinational	1885	1659
Buf/Inv	87	65
Noncombinational	219	218
Total	2104	$\boxed{1877}$

Number of	T_{ck} =1.73 ns	T_{ck} =6.92 ns
Ports	684	684
Nets	2321	2024
Cells	1516	1171
Combinational cells	1364	1019
Sequential cells	116	116
Buf/inv	170	143
References	29	29
Area [µm^2]	T_{ck} =1.72 ns	T_{ck} =6.88 ns
Combinational	1895	1657
Buf/Inv	105	82
Noncombinational	617	617
Total	2512	$\boxed{2274}$

Table 7: Post-synthesis area report for not-optimized (left) and optimized (right) Look-Ahead architectures

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Register	28	7.2	3.6×10^3	39	15%
Combinational	100.21	88.58	3.2×10^4	221	85%)
Total	128.30 µW	96 µW	3.6×10^4 nW	$\boxed{260\,\mu W}$	100%

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Register	143	39	1.02×10^4	191	34%
Combinational	181	160	3.6×10^4	377	66%
Total	323 µW	199 µW	4.6×10^4 nW	$\boxed{568\,\mu W}$	100%

Table 8: Post-synthesis power report for not optimized (top) and optimized (bottom) Look-Ahead architectures. $T_{ck} = 4T_{c,min}$.

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Sequential	0.037	0.014	0.0035	0.054	11%
Combinational	0.22	0.19	0.031	0.44	89%
Total	0.26 mW	0.20 mW	0.035 mW	$\boxed{0.49\,mW}$	100%

Power	**Internal**	**Switching**	**Leakage**	**Total**	**Percentage**
Sequential	0.20	0.072	0.01	0.28	24%
Combinational	0.48	0.40	0.034	0.91	76%
Total	0.69 mW	0.47 mW	0.044 mW	$\boxed{1.2\,mW}$	100%

Table 9: Post-placing power report for not optimized (top) and optimized (bottom) Look-Ahead architectures. $T_{ck} = 4T_{c,min}$.

Given how much area and energy overhead the pipeline registers add, in the view to get a good trade-off between speed enhancement, energy saving and area extension, a single pipe level implementation could be a good design choice.

7 Appendix

7.1 C vs MATLAB model

HDL numerical results are identical to the one of the C model.

	C	**MATLAB**	**Error**
1	0	0	0
2	16	16	0
3	37	38	0.004
4	67	69	0.008
5	100	103	0.012
6	115	118	0.012
...			

	C	**MATLAB**	**Error**
65	101	104	0.012
66	116	118	0.012
67	125	128	0.012
...			
199	-106	-105	0.004
200	-75	-75	0.008
201	-43	-41	0.008

Table 10: MATLAB vs C model output samples