Politecnico di Torino

Lab 1: Design and Implementation of a Digital Filter

Integrated Systems Architecture

Master degree in Electronic Engineering

Group 14

Guella Flavia, Romero Rondon Gabriel, Valente Simone

November 20, 2021

Contents

CHAPTER 1

Introduction

1.1 IIR Filters

The purpose of this laboratory is to perform the complete design flow for a digital filter, starting from a high level model (Matlab and C), up to a final implementation for which synthesis and place and route are performed. In addition to that, at the end, some techniques are applied in order to improve the overall performances of the digital filter.

The filter that is required to be implemented is an IIR filter (Infinite Impulse Response). This type of digital filter, differently from the FIR (Finite Impulse Response), contains a feedback from the output to the input. The arithmetic function that represents the IIR filters is reported below:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] + \sum_{k=1}^{M} a_k y[n-k] \tag{1.1}$$

Where N is the order of the filter and M indicates the feedback order. The straightforward implementation of the algorithm, namely Direct Form I, is presented in figure 1.1.

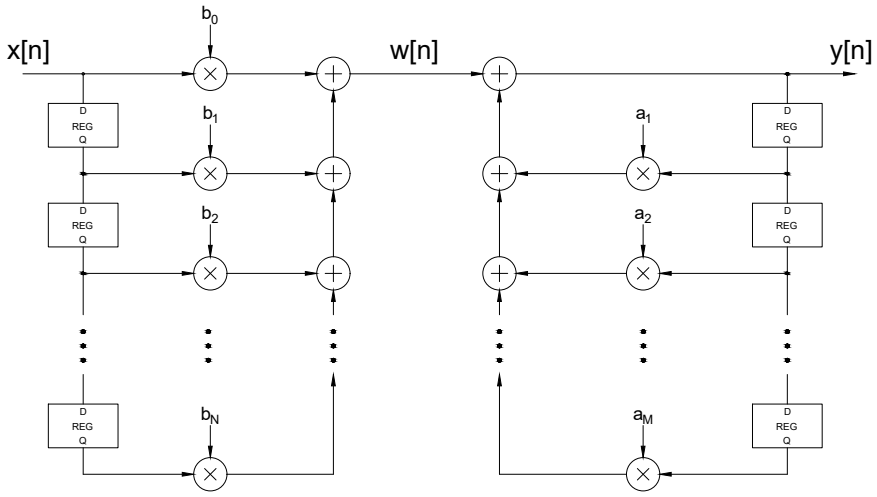

Figure 1.1: IIR Filter: Direct Form I

1.2 Source code

All the files that will be discussed in this document are stored in a GitHub public repository. For this particular laboratory, the files are all stored in the following link: Lab1 Files.

The following tree describes the directory organization, highlighting the most relevant directories and files.

```
Files
├── VHDL (Folder containing the VHDL files for development and simulation)
│   ├── sim (Scripts and results of simulation)
│   ├── src (Source VHDL files)
│   ├── tb (Testbench files)
│   └── look_ahead (Folder corresponding to the look-ahead architecture)
│       ├── sim
│       ├── src
│       └── tb
├── VHDL_synth (Folder containing the VHDL files for synthesis and PnR)
│   ├── syn (Folder with the synthesis results, reports and scripts)
│   └── look_ahead
│       ├── syn
│       ├── innovus (Folder with the PnR results, reports and scripts)
│       └── netlist (Folder with the netlist files resulting from synthesis)
├── IIR_filter.m (Matlab file with filter function and also the script that uses it)
├── main.c (C program for representing the IIR filter behaviour)
├── samples.txt (File containing the samples used as input of the filter)
├── resultsc.txt (Response of the filter using program in C)
└── resultsm.txt (Response of the filter using Matlab script)
```

CHAPTER 2

Reference model development

2.1 Filter design and coefficient quantization

In this case, it is required to design an IIR filter of order N=1 and with M=N. Moreover, the filtering operation to be performed is represented by equation 2.1. Notice that the sign of the second term has changed in this equation, this is related to the notation that is used by Matlab.

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{M} a_k y[n-k] \tag{2.1}$$

The architecture to be implemented is not the Direct Form I but it is the Direct Form II (canonical direct form), whose block diagram is showed in figure 2.1.

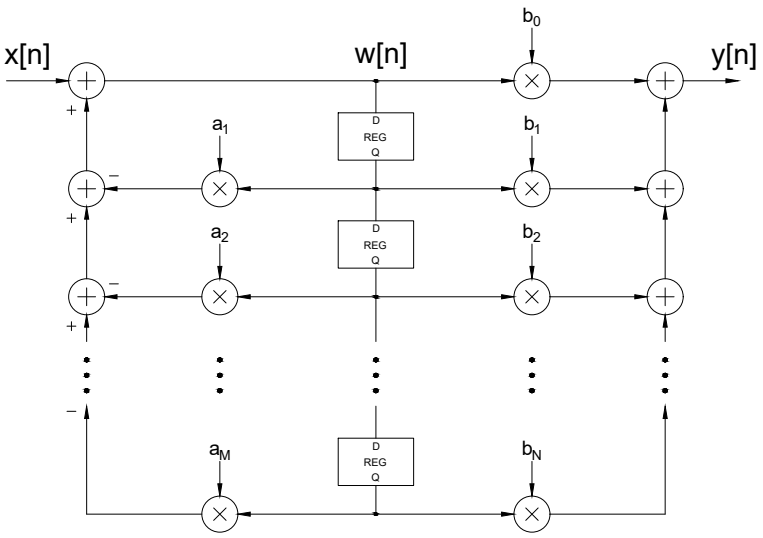

Figure 2.1: IIR Filter: Direct Form II

Comparing figures 1.1 and 2.1, it is possible to notice some differences. The main difference between Direct Form I and Direct Form II is that the number of registers is halved. This improvement is due to the fact that it is observed that the chain of registers delaying the input samples, and the one delaying the output can be shared in a single line, applying the commutative property among the two

parts (the two sums) of our expression.

Considering that in real systems we are constrained by the data precision, which is represented by a finite number of bits, then it is impossible to practically implement a filter with an infinite precision, it is necessary to quantize the input samples and the coefficients.

The usual implementation of digital filters in DSPs is based on fixed-point numbers, rather than on floating points. The latter category of numbers can be better exploited to model real quantities, and thus is more accurate in describing the real world. However, the cost of a floating point implementation is in general too high to be paid for a normal DSP. On the other hand, fixed-point numbers have an intermediate complexity between integers and floating point and are cheaper and easier to manage. Obviously, there is a cost to be paid choosing fixed-point numbers over floating points, as the first are less precise. Precision is lost on the right part of the decimal number, by reducing the width of the fixed point number. Overall, fixed-point propose a good trade-off between precision and cost and so they are generally adopted in digital filters domain.

The number of bits of precision (n_b) for our implementation is evaluated according to the following expression (2.2):

$$n_b = (y \, mod 7) + 8 \tag{2.2}$$

where $y = 6$ (i.e. numbers of characters in the surname "Romero").
The formula used to evaluate the order of the filter, $N = 1$ is (2.3):

$$N = 2^p \cdot [(x \, mod 2) + 1] + 6 \cdot p \tag{2.3}$$

Here $p = 0$ (even group number) and $x = 6$ (i.e. numbers of characters in the surname "Guella").
Given the value $N = 1$, the filter to design will have three coefficients: b_0, b_1, a_1 (a_0 is always assumed to be 1). All these coefficients will be quantized on $n_b = 14$ bits to obtain a fixed-point implementation of the IIR filter.

2.2 Matlab pseudo-fixed-point

Starting from a high-level model of the filter, it is possible to perform an initial evaluation of the behaviour and the performances of the filter. The first model of our digital filter is written using Matlab; the purpose of this step is also to obtain the values of the coefficients of the filter a and b. Specifically, two built-in functions provided by Matlab are very useful in the process of designing the filter, these functions are: *butter* and *filter*.

The prototype of the function *butter* is:

$$[b, a] = butter(N, Wn) \tag{2.4}$$

Where N is the order of the filter and Wn is the normalized cut-off frequency. By calling this function, the a and b coefficients for our filters are evaluated as floating-point numbers.

In addition to that, in the Matlab script *myiir_design.m*, the coefficients are then converted into integers, according to the designated precision ($n_b = 14$), in order to be compliant with the other

filter implementations, both for C and VHDL.

For our specific values of N and n_b, the function *butter* returns the following coefficients:

$$b_0 = 0.4208 \qquad b_1 = 0.4208 \qquad a_0 = 1 \qquad a_1 = -0.1584 \tag{2.5}$$

Then, we converting these coefficients into integers on $n_b = 14$ bits, obtaining:

$$b_0 = 3447 \qquad b_1 = 3447 \qquad a_0 = 8192 \qquad a_1 = -1298 \tag{2.6}$$

Notice that the coefficient a_1 is negative, because the equation used for the filter corresponds to equation 2.1. This point will be highlighted in the C implementation.

In figure 2.2, the frequency response of the Butterworth filter implemented by Matlab (red dashed line) and the one of the quantized (in blue) filter are shown.

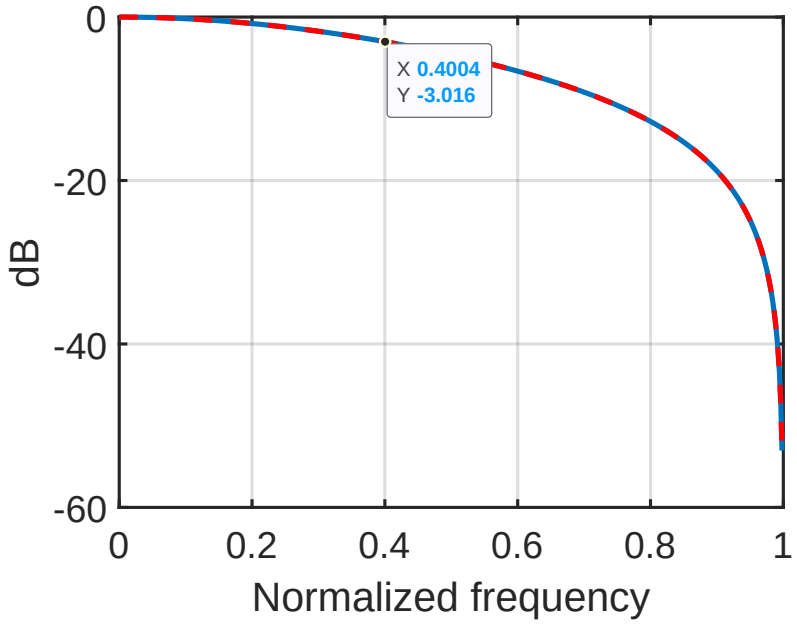

Figure 2.2: Butterworth filter: Frequency response

As already known, the Butterworth function, implements a low-pass filter with maximally-flat bandwidth, and according to our specifications the cut-off frequency should be at 2 kHz. We can conclude that the requirements are met, as shown by the label in Figure 2.2, where the frequency of the -3dB point is highlighted. The x-axis of the figure corresponds to the normalized frequency; the normalization of the frequency is obtained after dividing by the Nyquist frequency (i.e. half of the sampling frequency). In this particular case, the sampling frequency (f_s) is 10 kHz, then, being the normalized frequency of the -3dB point close to 0.4, we confirm that the requirement is fulfilled.
It can be further observed that the two transfer functions differ in an subtle way.

At this point, the devised function is tested on a specific signal that is the superposition (average) of two sinusoidal waves at different frequencies. In particular, the two sinewaves have the frequencies

$f_1 = 500\ Hz$ and $f_2 = 4500\ Hz$. This means that the first component is within the pass-band of the filter, whereas the second component (at frequency f_2) is falling out of the pass-band of the designed filter. This mismatch between the two components of the input signals will have a visible effect on the output signal.

The obtained input signal is sampled with a sampling frequency $f_s = 10\ kHz$, in order to be given as an input to the digital filter and the observation window used is $T = 2\ ms$.

Figure 2.3 and 2.4 show the two starting signals and the resulting one. Notice that both the initial signals and the resulting input signal are is normalized between -1 and 1.

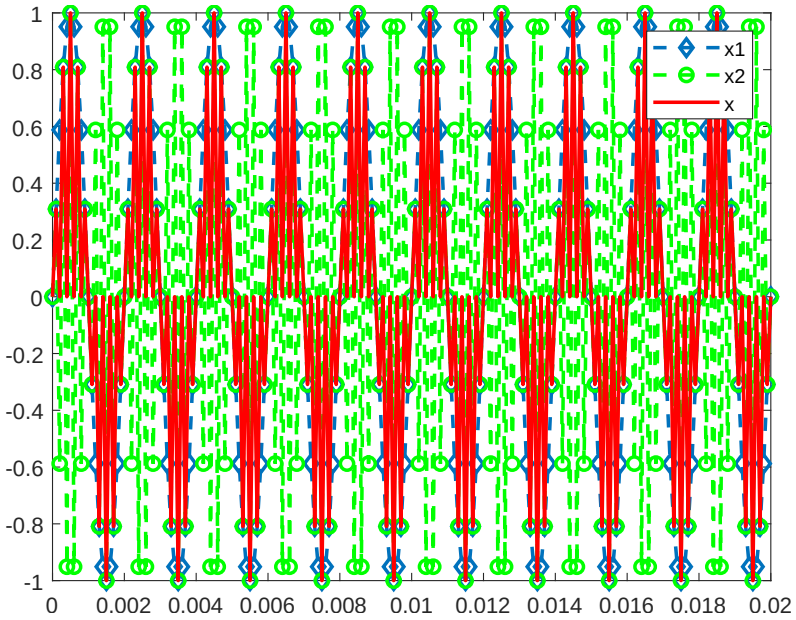

Figure 2.3: Input signal

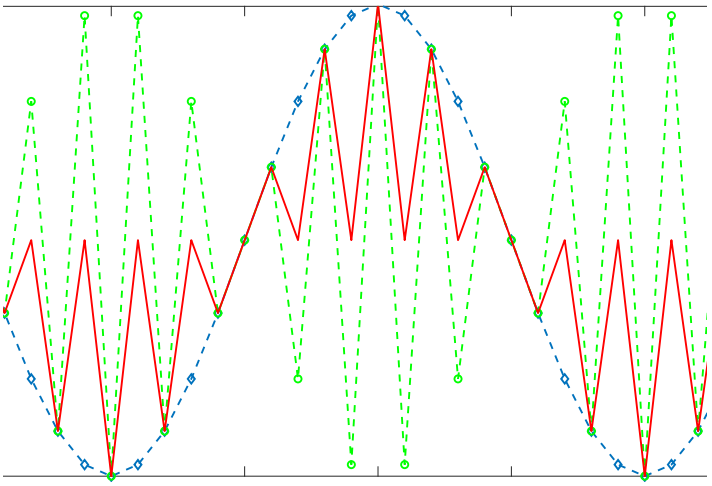

Figure 2.4: Input signal zoom

The filtering operation is implemented by the *filter* built-in function whose prototype is:

$$y = filter(b, a, x) \tag{2.7}$$

The output signal is reported in figure 2.5.

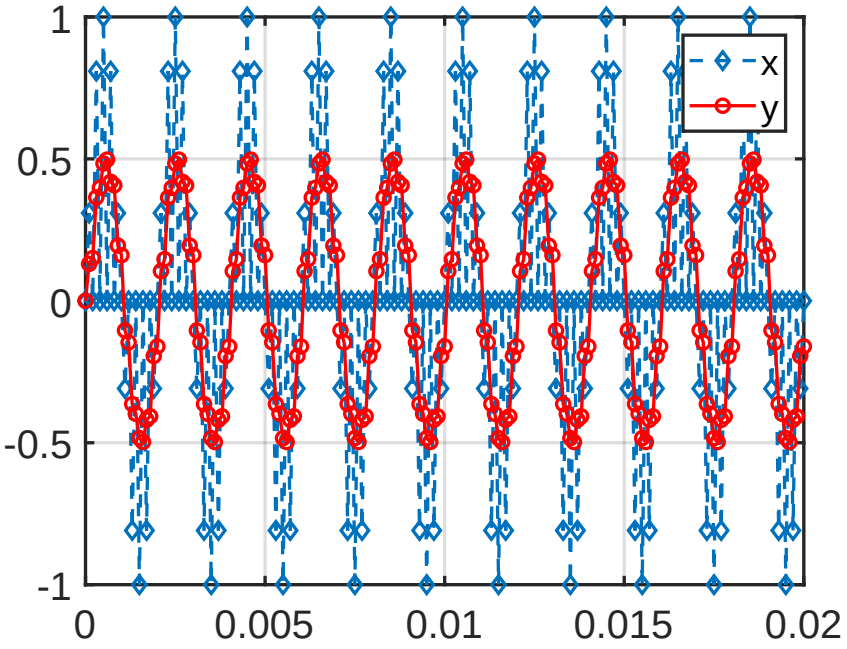

Figure 2.5: Input and output signals

It can be noticed that the output signal is smaller in magnitude compared to the input one, due to the fact that the filtering operation attenuates for sure the high frequency component, related to x_2, that falls out of the band.

2.3 Fixed-point C model

In the Introduction section, we already mentioned that the aim of this laboratory is to implement a digital filter of order $N = 1$ performing the function 2.1, with a precision of $n_b = 14$. In particular, the IIR filter has to be designed according to the Direct Form II model (Figure 2.1).

The C function is taking as input each of the samples of the input signal (exported from the Matlab script) and it returns the corresponding output sample, computed by going through the feed-back and feed-forward nodes, and the intermediate values w and sw. sw corresponds to the set of registers that are connected to the node w (figure 2.1), for our architecture there is a single register. At the first call of the function *myfilter*, the chain of FFs is cleared, while for successive calls we need to update the variable sw as $sw[0] = w$. Notice that sw variable is declared as static so that it maintains its value through consecutive calls. Moreover both w and sw are integer of full size in this initial phase.

In the main function, the *myfilter* function is invoked for every input sample of the x signal, whose values are coming directly from its integer quantized version x_q, evaluated in Matlab.
The coefficients a and b of the filter are declared as integer values, and they come from the results of

the *butter* Matlab function, redefined as integers on 14 bits (i.e. the values in 2.6).

Regarding the coefficients, while for the b values no change need to be made, for a coefficient, as already anticipated in the previous section, a change of sign is needed. This is necessary in order to obtain results that are similar to the Matlab output and to implement the minus in the IIR algorithm. Further will be said about this issue in the following.
The output samples evaluated through this implementation are then saved into an output file and plotted with Matlab.

In the following part a comparison is present between the results obtained using a positive or negative value for the coefficient a_1 (auto-regressive part coefficient).
Figure 2.6 shows the output of the C program, by setting $a_1 = -1298$, while figure 2.7 shows the output of the C program, by setting $a_1 = 1298$. In both cases, the plot also shows the quantized output signal (pseudo fixed-point) obtained through Matlab.

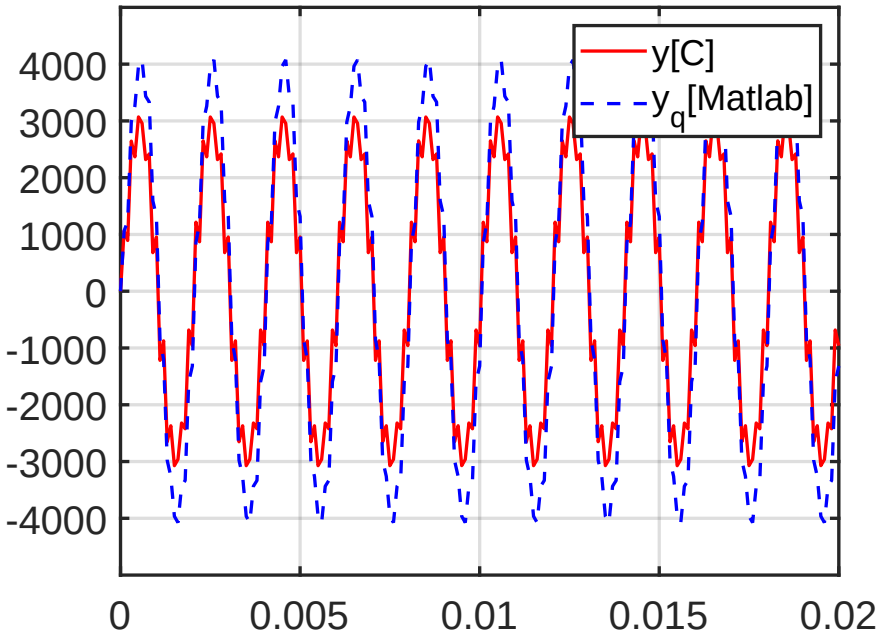

Figure 2.6: C($a_1 = -1298$) vs Matlab output signal

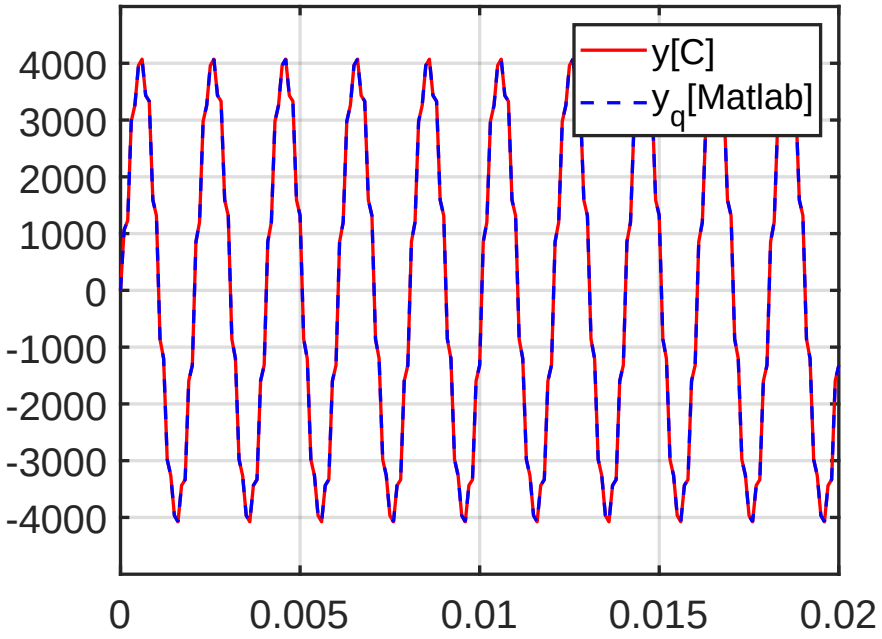

Figure 2.7: C($a_1 = 1298$) vs Matlab output signal

Comparing the two plots, it can be immediately observed that the figure 2.6 shows quite a different behaviour with respect to the Matlab output. On the other hand, figure 2.7 shows two results that are almost overlapped, and that highlight an initial distinction between the fixed-point C implementation and the Matlab pseudo-fixed point implementation that is almost negligible in terms of loss of precision. For this reason, since we are not able to directly see the implementation of the *butter* and the *filter* functions by Matlab, we assume that there is a change in the sign that has to be performed in the C code to obtain the filtering action required.

Further step that is performed through Matlab built-in function *thd*, is the evaluation of the Total Harmonic Distortion (THD) of the output of the C program. The THD is defined as:

$$THD_{\%} = 100 \cdot \frac{\sqrt{\sum_{n=2}^{\infty} V_{n\_rms}^2}}{V_{1\_rms}} \tag{2.8}$$

As defined in expression 2.8, it is easy to understand that the smaller is the THD the less the output signal is distorted (the harmonic components different from the fundamental are not as relevant as the first).

Figure 2.8 shows the frequency spectrum of the signal obtained interpolating the output samples coming from the C implementation. Then, as shown also in the figure, the THD obtained by the Matlab function is $-76.99\ dB$.

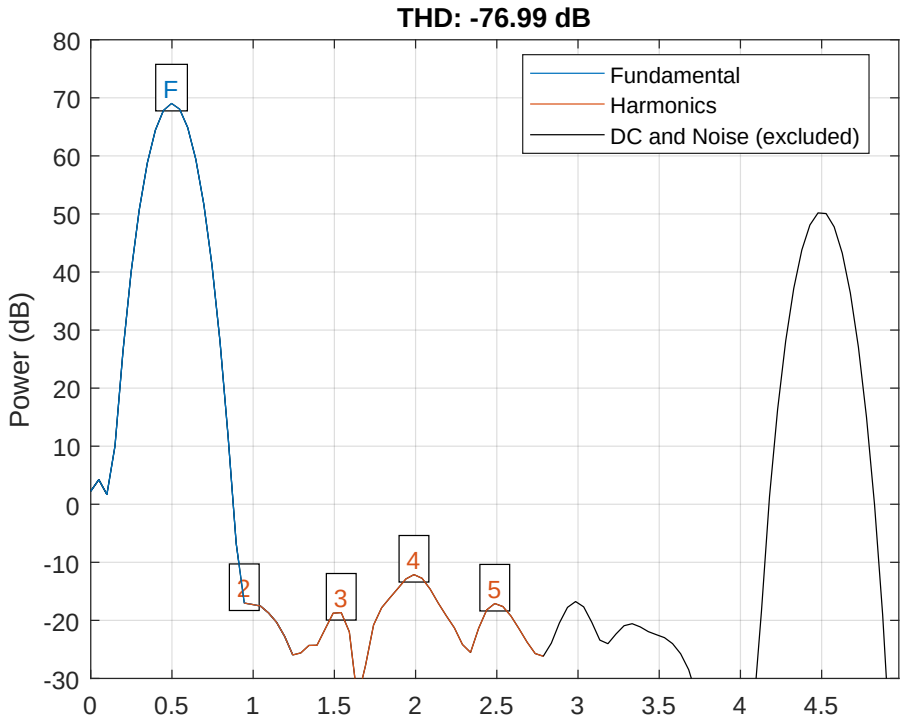

Figure 2.8: Spectrum in frequency of C output

2.3.1 Area optimization

From the perspective of a physical implementation of our design, a further refinement and optimization of the high-level C model is performed in terms of area. Achieving this objective requires a reduction in the internal sizing of the signals, in order to reduce the bit-width, thus the area. As an additional improvement, we also expect that the bit-width reduction would also reduce the delay of the different processing units.

The C code was already devised to allow this optimization, by reducing the size of each product. To exploit this possibility it is defined a macro R. The result of each multiplication is shifted right of $n_b - 1 + R$ bits. R is tuned through Matlab simulation, choosing the largest value that is able to provide an output signal while fulfilling the given constraint for the distortion: $THD < -30dB$.

It is found that for $R = 7$ the corresponding distortion is $THD = -36.15dB$, as shown in the plot in figure 2.9. We know that this is the maximum value of R that allow us to maintain the THD specification, because $R = 8$ would give a THD that is above -30 dB.

```
#define R 7
...

for (i=0; i<N; i++)
{
    fb -= (sw[i]*a[i]) >> (NB+R-1);
    ff += (sw[i]*b[i]) >> (NB+R-1);
}
y = (w*b0) >> (NB+R-1);
```

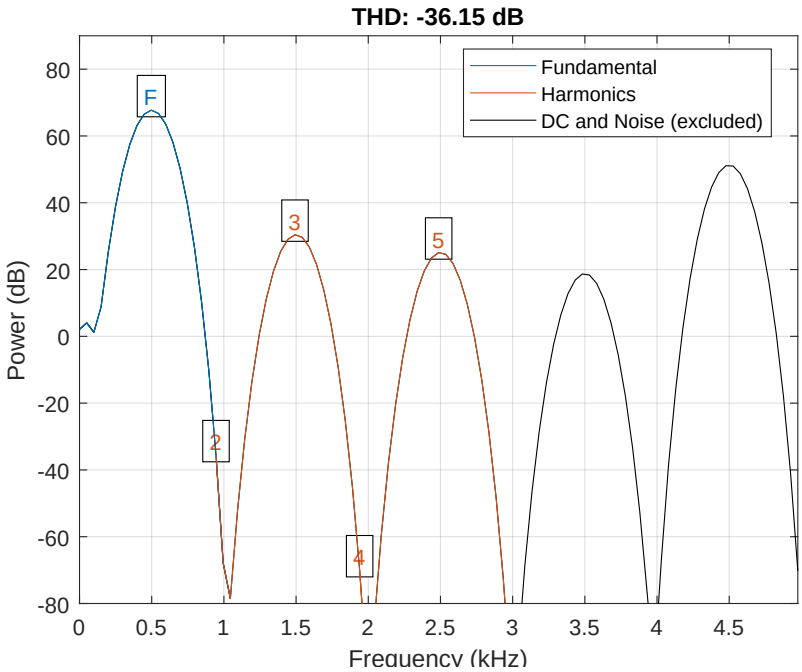

Figure 2.9: Spectrum in frequency of C output

Some consideration can be made on this approach. It is decided, as a design choice, not to directly act on the bit-width of the input, output and coefficients, but only to optimize the internal signals, and, specifically, the output of the multipliers. In this way, as it will be more clear in the VHDL description, the size of the adders/subtractors and of the multipliers is significantly reduced and still a good filtering function is obtained.

In figure 2.10 the output of the filter for the same test signal used in the previous section is showed for this new implementation and is compared to the original Matlab filter output. To perform a significant comparison, the output of the C function is rescaled, by shifting it left of R bits.

The input discrete-time signal (x), as already mentioned, is normalized between -1 and 1. It is then transformed into integer multiplying by $2^{(n_b-1)}$. The samples received as an input by the C code are thus limited in amplitude in the range $[-8192; 8191]$, meaning they are represented on 14 bits as signed numbers. The same kind of quantization is applied, as already seen in equation 2.6, to the a and b coefficients. In the initial C implementation, the result of the multiplications were re-scaled shifting right of $n_b - 1$ (=13) bits. So, strictly speaking, since *int* type is on 32 bits, we re-scale it on 19 bits. In the new implementation, instead, we are re-scaling on the least significant 12 bits. Further considerations on the actual number of bits that are necessary and on this resizing operation will be made in the following chapter, to better understand the VHDL description of our architecture.

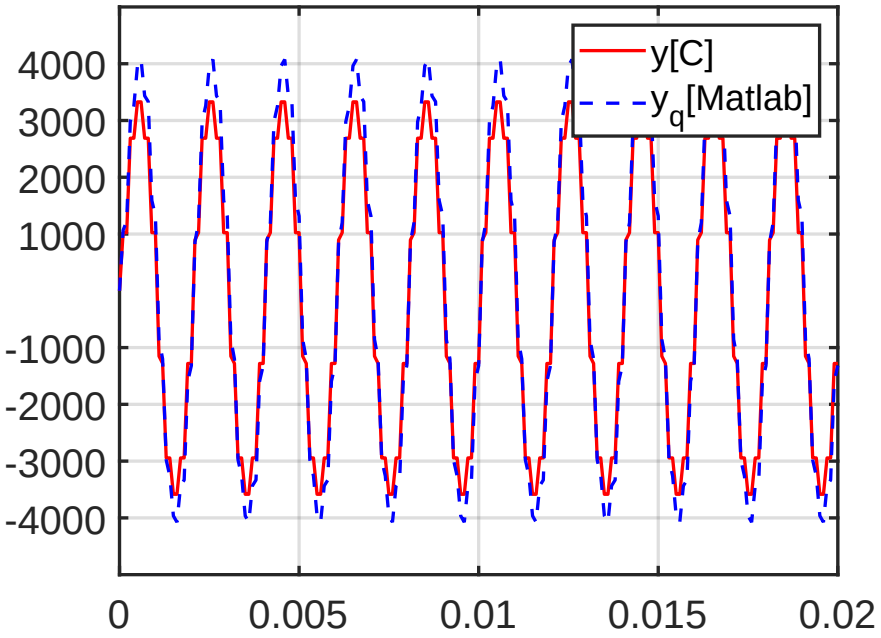

Figure 2.10: Output signal

CHAPTER 3

VLSI implementation

3.1 Starting architecture development

In the following section the steps performed in order to create a circuit description of the IIR filter are summarized. The designed circuit must be compliant with the previously discussed specifications and constraints. The most critical phases of the VHDL circuit development will be highlighted, as well as the design choices made.

The starting point of the design is the high-level C model described at the end of the previous chapter, where some of the internal signals sizes were reduced to optimize the area of the design. Then, the most critical step consists in correctly sizing and managing those internal signals in the VHDL code. As already mentioned, the design choice is to leave the external interface unchanged, so DIN, $DOUT$ and all the filter coefficients will be 14-bit wide. As in the C code, the internal resize is performed only after the multiplications are carried out. Basically, of the whole result of each multiplication only a part is kept. It is important to correctly select the bits that are truncated, in order not to lose precision with respect to the C implementation.

The interface of the filter block is presented in the following listing. This implementation, besides the usual input and output ports (DIN and $DOUT$), also includes two additional flags for indicating when the input and output are ready. These flags are the ports VIN and $VOUT$. Also, notice that the filter also includes the coefficients a and b as inputs, thus allowing to set the behaviour of the filter from the outside of the filter block.

```
entity IIR_FILTER is
  port(
    DIN: std_logic_vector(nb-1 downto 0);
    VIN : in std_logic;
    RST_n : in std_logic;
    CLK : in std_logic;
    a0 : in std_logic_vector(nb-1 downto 0);
    a1 : in std_logic_vector(nb-1 downto 0);
    b0 : in std_logic_vector(nb-1 downto 0);
    b1 : in std_logic_vector(nb-1 downto 0);
    DOUT : out std_logic_vector(nb-1 downto 0);
    VOUT : out std_logic
  );
end IIR_FILTER;
```

3.1.1 Sizing the internal representation

It is easy to understand that the case in which the truncation at the output of the multiplier is more critical is the feed-back loop of the filter. This is critical because if we do not limit the size of the multiplier output in the feed-back chain the required sizes for the internal signals start to increase at every cycle, and at some point overflow will occur. Moreover, not limiting the multiplier result would force the size of the first adder/subtractor to increase as well, but if it increases also the multipliers size has to increase and so on and so forth in a positive feed-back situation.

The reasonable choice, considering the C results, to limit the multiplication result has several advantages on the overall performances of the architecture. The shift operation performed in the C code leaves only the 12 MSBs of the product (range *31 downto 20*, since *int* type is on 32 bits, and we shift right of 20 positions), but not all of them are actually needed.

Figure 3.1 shows the block diagram of the IIR filter with all the width of the different internal signals specified, in order to have a more clear understanding of this section.

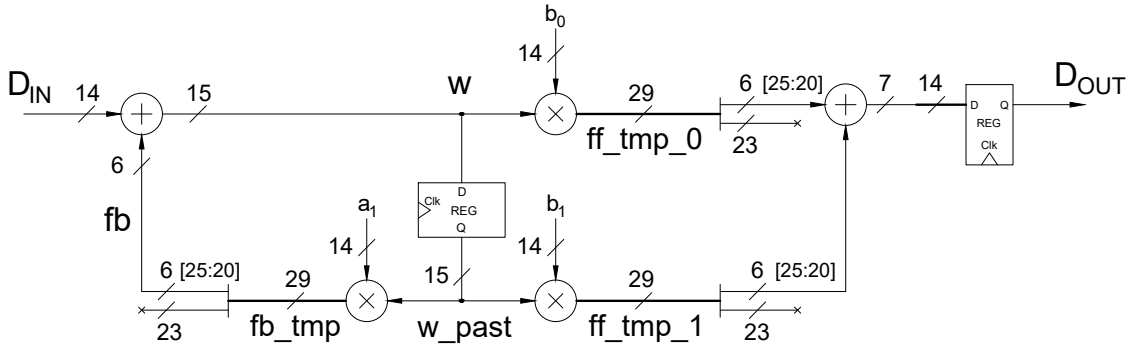

Figure 3.1: Block Diagram, IIR filter (N=M=1)

The main point here is to understand how to perform the selection of the resulting bits coming from the multiplier outputs.

The first step in the VHDL consists on the sizing of the w signal, i.e. the node/wire connecting the feed-back and feed-forward parts, as shown in Figure 2.1. This wire is carrying the result of the sum between the input samples (on 14 bits) and the result of a multiplication (truncated to less than 14 bits, as in C already). w signal will thus be on 15 bits, in order to take into account a possible carry out from the sum/sub, and so to manage *overflow*. It can be observed that w will never assume the maximum or minimum value on 15 bits, because one of the operands in the sum/-sub is on much less than 14 bits. This is a useful information in order to perform a worst-case analysis.

The two input operands of the multipliers are one on 14 bits and the other on 15 bits, thus the result will be on 29 bits. The point is how many of those bits are useful for us and carry useful information.

For a worst-case analysis, we can consider b_0 as one of the multiplier inputs (since $b_0 = b_1 > a_1$) and the biggest number that w could possibly have. As already anticipated, the maximum (in absolute value) value for w is actually not -2^{14} but it is, at most, the largest value of DIN plus the largest value of fb. Then, since we assume to take only 12 bits of the output of the multiplier, as in

the C implementation, the largest value of fb is -2^{11}. This derives in a maximum value of w equal to $-2^{13} - 2^{11}$.
Indicating with x_{worst_case} the number of bits of the products in the worst case:

$$x_{worst\_case} = |(-2^{13}) \cdot (-2^{13} - 2^{11})| < 2^{27} \tag{3.1}$$

But, actually, we can particularize even more, considering that b_0 is not -2^{13}, but it is smaller $(3447 < 2^{12})$.

$$x_{worst\_case} < |(2^{12}) \cdot (-2^{13} - 2^{11})| < 2^{26}; \tag{3.2}$$

The conclusion that can be drawn from the equation 3.2, is that in the result of the multiplications, the bits from 28 to 27 (considering the VHDL complete result is on 29 bits) are "useless" for us, they will be only a sign-extension. Even in the worst-case these bits will not store any relevant information. Putting together what just demonstrated and the C constraint, the significant bits of our result will be *26 downto 20*, where the 20 comes from the C code and the 26 from this analysis.
The results of the multiplications are thus on 7 bits, rather than on 12, as initially deduced from the C code, thus the size of the products can be further shrunk.

Now, considering the actual value of the coefficients it is possible to reduce the bitwidth of the signals as demonstrated by equation 3.3.

$$x_{worst\_case} = |3447 \cdot (-2^{13} - 2^{6})| < 2^{25}; \tag{3.3}$$

Then, the width of the signal *fb* will be of 6 bits, rather than 7. The six relevant bits corresponds to the bits in the range *25 downto 20* of the full product. Using the constants defined in the package, it is possible to have a generic representation of this range, as shown in the following listing.

```
fb <= fb_tmp(nb*2-3 downto nb*2-2-(nb-r)+1);
ff <= std_logic_vector(signed(ff_tmp_0(nb*2-2 downto nb*2-2-(nb-r)+1)) + signed(ff_tmp_1(nb
    *2-2 downto nb*2-2-(nb-r)+1)))
```

The reported VHDL code shows also how the *ff* signal is evaluated. Since *ff* is defined as the sum of two 6-bits signals, it will be on 7 bits. The sum of the two branches implemented using the '+' operator and converting the different signals, that were defined std_logic_vector into signed, and then converting back the result. The same approach is followed also to implement the multiplications and the other sum.

As a final remark, since the output needs to be on 14 bits a constant called *zero_padding*, made of 7 bits all at 0, is defined and it is concatenated to the right of the output signal.

```
constant zero_padding : std_logic_vector (r-1 downto 0):= (others => '0') ;
...
DOUT <= ff & zero_padding;
```

3.1.2 Further implementation details

Another important point to be remarked is related to the synchronisation of the output signal, by means of a register. In the following it is reported the process, synchronous to the clock signal that manages the sync part of our design. The process is synchronous to the clock rising edge, and the reset signal is synchronous as well.

```
filter : process(clk)

  begin

  if(clk' event and clk='1') then --SYNCR RESET SIGNAL
    if(RST_n ='0') then
    --model reset behaviour
      VOUT <= '0';
      DOUT <= (others =>'0');
       w_past <= (others =>'0');
    --distinguish the two cases in which a new input sample is available (Vin=1) or not
    elsif (VIN='1') then
      --perform the task;normal operation of the filter
       w_past <= w;
      DOUT <= ff & zero_padding;
      VOUT <= '1';

    elsif (VIN='0') then
    --no valid output computed
      VOUT <= '0';
    end if;

  end if;

end process;
```

The problem of the validation of the data is managed through the *VIN* signal, when it is at '0' the input data is not valid, thus in the next clock cycle the *VOUT* signal will remain '0' (no valid output available), while the registers are all disabled. To summarize, *VIN* signal is acting as an enable signal for all the registers in the architecture. *DOUT* is implemented as well as *VOUT* going through a final register as required.

On the other hand, the choice for the input is to rely on the testbench (for the simulation) or on the upstream block in the real case, and assume that the input will arrive in a synchronous way, one every clock cycle, thus avoiding to put an additional register at the input of the design, and delaying of a further clock cycle the computation.

3.2 Simulation

3.2.1 Testbench

In order to simulate our design, the testbench taken as a model is the one already provided. In it there is a module (*data_maker*) that is taking the samples of our test signal (the one generated with Matlab for example), and it is providing it together with the validation signal to our filter. Then an output module is sampling the output signals of the filter and storing them into a file.

It is important to observe that the *data_maker* module is also providing the block with the coefficients. The coefficients are set as shown in the following listing.

```
    H0 <= conv_std_logic_vector(1,nb);  --a0
    H1 <= conv_std_logic_vector(1298,nb); --a1
    H2 <= conv_std_logic_vector(3447,nb); --b0
    H3 <= conv_std_logic_vector(3447,nb);  --b1
```

Notice that a_1 is defined following the C implementation. This means that the coefficient has a positive value and the '+' is used in the feed-back node to obtain the required filtering function represented in equation 2.1.

3.2.2 Simulation results

There is complete equivalence between the results obtained in from the C description and the results of the simulations, stored in the *results.txt* file. Similarly, the following figures represent some interesting snapshots of the simulation, highlighting the behaviour of the design at the beginning and at the end of the simulation, thus when *VIN* is going low-to-high and high-to-low.

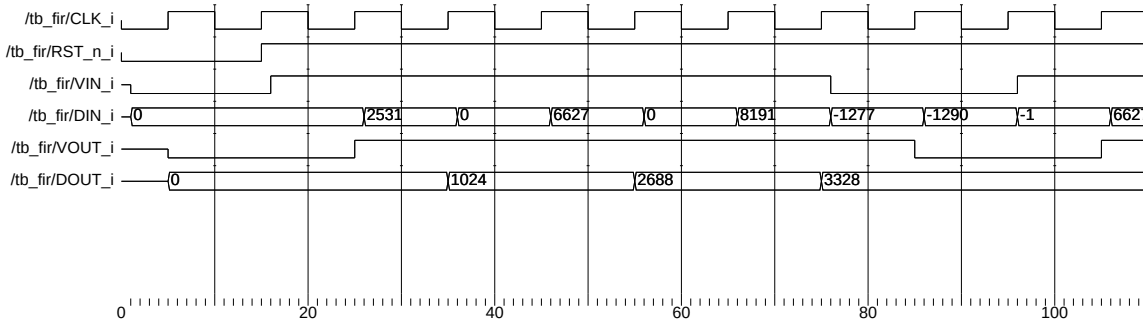

Figure 3.2: Start of simulation

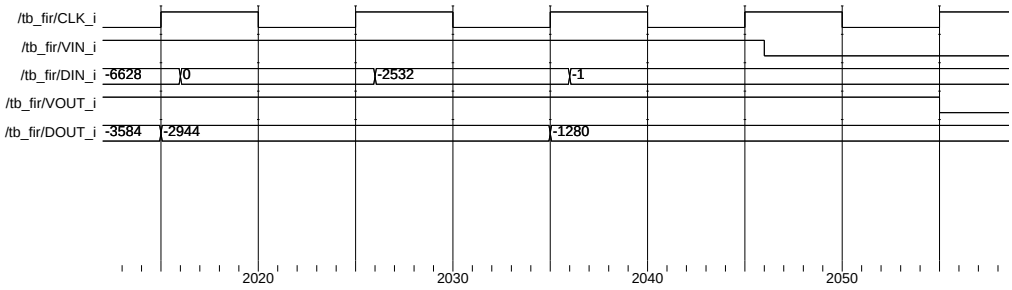

Figure 3.3: End of simulation

Figure 3.2 shows the behaviour at the beginning of the simulation, so the reset signal (active low) and then the first sampled input value and output produced (with *VOUT* going high). Moreover, the aspect to be highlighted is mostly related to the preservation of the correct behaviour when *VIN* goes to '0' (invalid input value). At around 70 ns, *VIN* goes low and, as a consequence, also *VOUT* becomes low after the following rising-edge of the clock, while *DOUT* keeps its old value.

In order to verify that the behavior of the architecture is compliant to the specifications, the testbench is modified adding a piece of code that gives the possibility to easily trigger this corner case.

```
--additional instructionts to perform the behaviour with Vin=0
--add in samples.txt these two values in whatever position
if (x=-1277 or x=-1290) then
    VOUT<='0' after tco; --VIN for our architecture
```

We simply added to the *sample.txt* file coming from Matlab test signal, two samples that contains values not corresponding to any other sample in the file. When one of these two values is encountered, the *VIN* signal (*VOUT* for *data_maker*) is turned low. Obviously this is not a general method, as it works only with this specific input file, and has to be commented out to ensure a correct behaviour

of the testbench in other cases, but it makes *data_maker* able to simply prove the correctness of the response of the filter, that is general, and stands for all input signals.
Figure 3.3 instead shows the behavior at the end of the simulation; the clock cycle after *VIN* goes low, also *VOUT* goes low and the last valid output is sampled by the data sink.

3.3 Logic Synthesis

In order to assess the performance achievable by a physical implementation of the IIR filter we proceed to synthesize it using Design Compiler. The sequence of commands needed to do this are all listed in the file *Files/VHDL_synth/syn/run.scr*, which can be executed as a shell script.

The first two lines prepare the environment for the synthesis, by compiling the design files using ModelSim and placing the output in the work library. After that, a script named *max_performance_DC.scr* is called and executed by Design Compiler. Such script analyzes the design files, elaborates the architecture of the filter and sets a clock period of 3.09 ns in order to analyze the sequential behaviour of the unit.

The value 3.09 ns is the lowest possible value for the clock period that returns a zero slack. With such a clock period the performance is maximum and the area needed to realize such a filter is reported in the file *area_max_perf.txt* as being equal to 2990 units.
Then a similar script named *first_DC.scr* is executed still with Design Compiler but this time with clock period set to 12.36 ns. The design is then compiled and the following information about it is saved in the corresponding files:

- post-synthesis netlist described in Verilog.
- post-synthesis netlist described in VHDL.
- delay information file.
- ports constraints file.
- timing report.
- area report.

The constraint on the timing (set by the clock period) is relaxed, indeed the area occupied by the filter is now lower as 2551 units.
Once this script is over, the execution returns to the run.scr and the next step is that of compiling, using again ModelSim, the testbench already used in the previous steps for testing the design files and the post-synthesis netlist just produced, in order to test the synthesized design.

After that, the *vsim* command is issued in order to execute another script, named *power.scr*, together with two labels, one needed for linking the physical library that Design Compiler used to synthesize the design to the analyzed files and one needed for taking into account the delay file mentioned above. In the *power.scr* script a simulation of the synthesized design is launched, after opening a value-change-dump file (needed for the next step of power estimation) called *IIR_syn.vcd* and adding to it all the signals in the unit under test. The simulation will create a file named *results.txt* that can be compared to the one obtained simulating the VHDL design files: they appear to be in complete agreement, therefore the synthesis of the design can be considered successful.
Once the simulation is over the execution returns once again to *run.scr*. Here the *IIR_syn.vcd* file produced in the previous step is converted in *.saif* format. Then Design Compiler is issued a second

time, in order to estimate the power consumption of the filter. The commands for it are listed in the *second_DC.scr*: the post-synthesis netlist is read, then the *.saif* file is read and a clock is created. Eventually, the power report is saved in *sw_pwr.txt* and from it the power consumption can be obtained:

- **Internal Power** = 0.37 mW
- **Switching Power** = 0.29 mW
- **Leakage Power** = 0.054 mW
- **Total Power** = 0.71 mW

3.4 Place and Route

With the tool Cadence Innovus the place and route of the cells composing the filter is performed, following the steps present in the course guide. As suggested there, all the files related to this step are saved in the *VHDL_synth/innovus* folder.

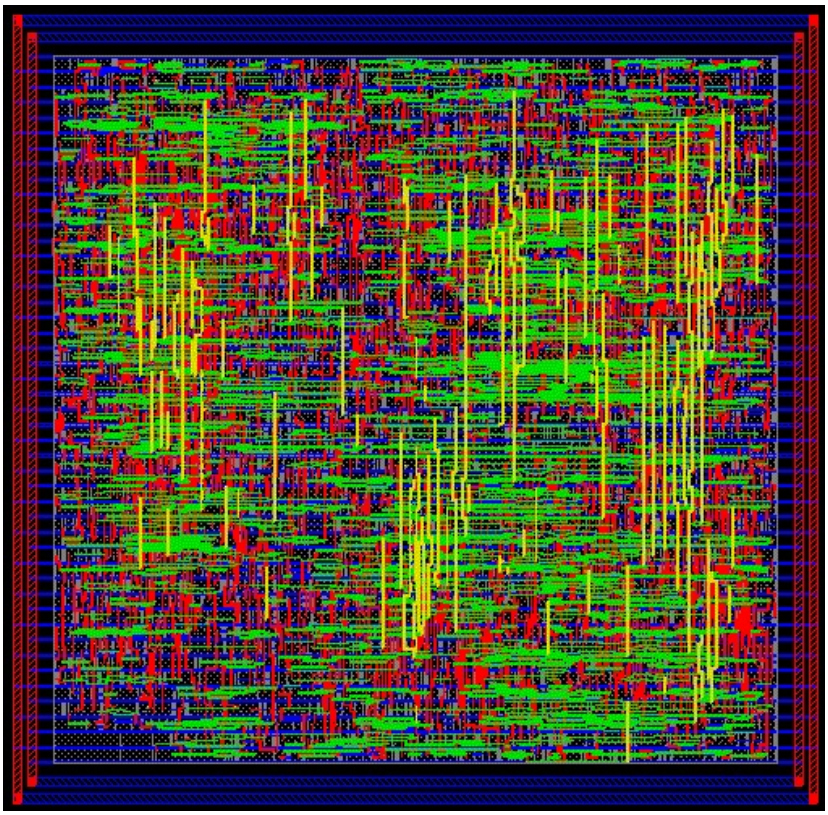

Figure 3.4: Place and Route result of the IIR Filter.

In figure 3.4 we can observe a graphical representation of the result of the place and route procedure. The design from which we started is the one saved in *netlist/synth_netlist.v*, that is the post-synthesis netlist obtained at the previous step, that with a relaxed clock period.
The area occupied by the filter after place and route is saved in the file *IIR_FILTER.gate_count* that reports:

- **Gates** = 3192

- **Cells** = 1304
- **Area** = 2547.5 μm^2

The post-place-and-route netlist can be simulated to ensure the final implementation actually works as expected. To do so a script is prepared, named *post_PnR_sim.scr* that using ModelSim compiles the testbench and the post-place and route netlist saved previously by Innovus in the file *IIR_FILTER.v*. Then the *vsim* command is launched, with the same labels used in previous steps to link the design to a physical library and to account for the delay information (useful for the power estimation performed later). The script executed by *vsim* is called *power.scr* and it opens a *.vcd* file to store all the changes on the signals of the unit under test and then runs a 1200 ns simulation, that thanks to the testbench architecture again produces a file named *results.txt* that stores the output samples of the post-place and route design. Those samples can be seen to be in agreement with the ones output by the other implementations analyzed before.

After the *.vcd* file has been filled we can go back to Innovus and perform a power consumption estimation. The result is stored in a file named *power_report.txt*. The results are here summarized:

- **Internal Power** = 0.51 mW
- **Switching Power** = 0.42 mW
- **Leakage Power** = 0.052 mW
- **Total Power** = 0.98 mW

Notice both the area and the power estimates are larger after the place and route procedure, but still in agreement with those values found after the synthesis.

CHAPTER 4

Advanced architecture development

4.1 Introduction

In this chapter we will apply some optimization method to improve the performances of our digital filter. In the case of the IIR filter, it is required to apply a non-universal method, the *J-look-ahead*, with $J = 1$.

The look-ahead is mostly exploited when no further reduction in the critical path delay is possible using universal methods. This technique consists in a sort of unrolling of the arithmetic operation performed by the block, in order to change its DFG and possibly allow the use of different techniques to further reduce the delay of the architecture.
For the IIR filter, the starting point is again *Direct Form II* and starting from the block diagram in figure 2.1, for N=1, the expression can be unrolled:

$$w[n] = x[n] - a_1 \cdot w[n-1] \tag{4.1}$$

$$y[n] = w[n] \cdot b_0 + w[n-1] \cdot b_1 \tag{4.2}$$

The unrolling is performed on $w[n]$ (equation 4.1) in order to have more room for optimization.

$$w[n-1] = x[n-1] - a_1 \cdot w[n-2] \tag{4.3}$$

We stop at this level, since J=1. Different solutions are possible, since $w[n-1]$ is present in different terms of our expressions, but the chosen approach is to modify only the feed-back part of the DFG. We leave unchanged the feed-forward part (that is the one where the multiplications by b_0 and b_1 are performed and the sum giving $y[n]$) while we act on the part containing the loop.

$$w[n] = x[n] - a_1 \cdot (x[n-1] - a_1 \cdot w[n-2]) \tag{4.4}$$

Whereas the output $y[n]$ is still obtained as in 4.2.

On top of this new architecture, retiming and pipelining can be exploited to reduce the delay of the critical path, thus improving performances. The DFG representing this behaviour is showed in figure 4.1. Two adders are used on the left part of the DFG, because as already explained in chapter 1, the sign of a_1 has been inverted in order to obtain the requested initial function for the digital filter (see equation 2.1).

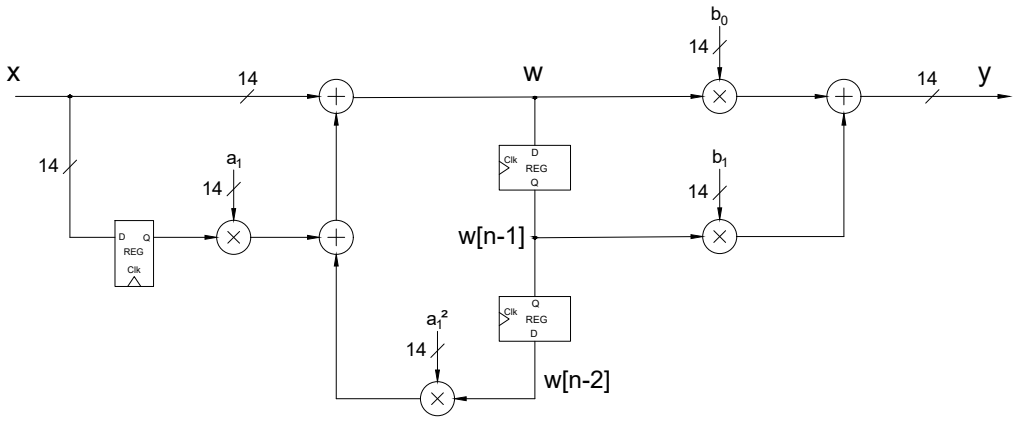

Figure 4.1: IIR filter, look-ahead

4.2 Preliminary analysis

The implementation of the look-ahead is not straightforward. As already mentioned, there are different possibilities and substitutions that can be exploited. A preliminary analysis is thus performed in order to understand which one among different possibilities could be the most convenient.

1 **Substituting w[n-1] unrolled both in w[n] and in y[n]**
The resulting DFG, after retiming, is composed of 5 multipliers, 4 adders and 4 registers and the maximum delay ignoring the contribution due to registers is $t_d = T_A + T_M$

2 **Substituting w[n-1] unrolled only in w[n]**
This is the solution that is chosen; after retiming there are 4 multipliers, 3 adders and 3 registers and $t_d = 2T_A + T_M$. This appears to be worse in terms of delay with respect to the case 1, but including also pipeline the delay is further reduced.

3 **Substituting w[n-1] unrolled only in y[n]**
After retiming there are 4 multipliers and 3 adders and 4 registers. $t_d = 2T_A + T_M$; also here with pipeline good improvements can be made.

Notice that in any of those cases, $w[n]$ is never substituted inside $y[n]$ expression, but the two expressions are implemented separately in the DFG, as it is done for the standard Direct Form II.

4.3 Architecture development

The VHDL netlist has to be modified in order to support look-ahead. The interface to the external world, as suggested by figure 4.1 contains an additional input coefficient: a_1^2.

```
entity IIR_FILTER is
  port(
    DIN: std_logic_vector(nb-1 downto 0);
    VIN : in std_logic;
    RST_n : in std_logic;
    CLK : in std_logic;
    a0 : in std_logic_vector(nb-1 downto 0);
    a1 : in std_logic_vector(nb-1 downto 0);
    a1_2: in std_logic_vector (nb_a-1 downto 0);
    b0 : in std_logic_vector(nb-1 downto 0);
```

```
    b1 : in std_logic_vector(nb-1 downto 0);
    DOUT : out std_logic_vector(nb-1 downto 0);
    VOUT : out std_logic
  );

end IIR_FILTER;
```

The coefficient a_1^2 is on $n_b\_a = n_b$ bits. The values of these constants that define the bitwidth of the signals are set in the package under the name lookah_pkg. As in the previous architecture, all the registers are described through a single process that is synchronous to the clock and they are only updated when the input data, *DIN*, carries a valid value (i.e. when *VIN*=1).

Look-ahead is, as already said, a technique that comes together with other optimization methods. In the following, the structure that is taken as a reference is the one in figure 4.2.

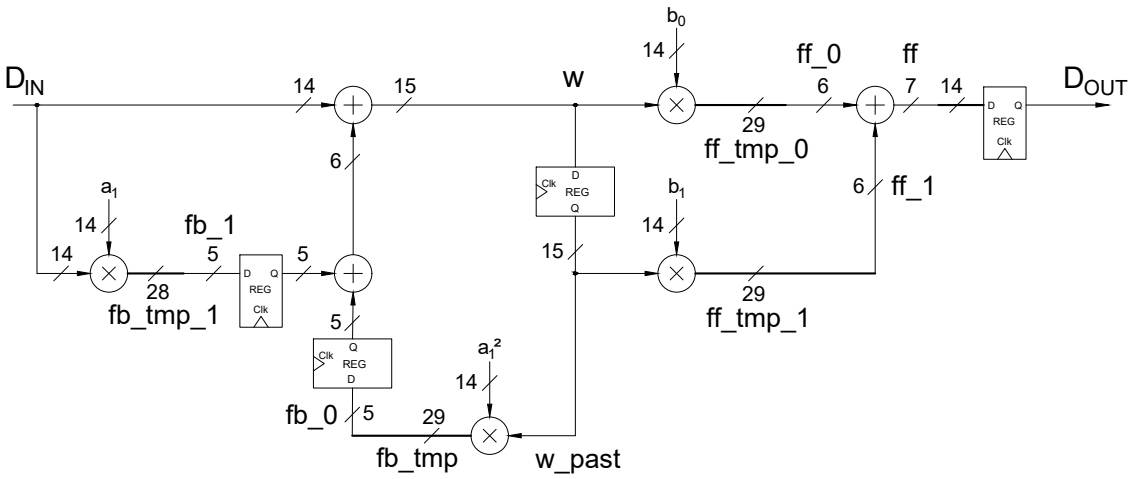

Figure 4.2: IIR filter, look-ahead and retiming

With respect to figure 4.1 registers are moved around, applying retiming through direct inspection. Notice that the two registers upstream the adder (the ones related to signals fb_0 and fb_{-1} could be "transformed" into a single register downstream the adder. This is not done to exploit better performance with pipeline, but this possibility will be however described in the following.

4.3.1 Sizing the internal representation

As in the initial architecture, some optimization to the bit-width needs to be carried out, otherwise the risk is to incur into overflow. It is suggested to refer to figure 4.2 to better understand the reference to the internal signals.

Again, the approach is the same, that is "truncating" the representation at the output of the multipliers. The golden model is still the C implementation, that would allow us to reach the specification on the THD. Therefore, the steps that are next described, are executed in order to obtain again results that are identical to the C and first VHDL implementation, and compliant to the requirements.

The additional coefficient a_1^2 is estimated starting from the real value of $a_1 = 0.1584$ coefficient calculated by Matlab, then rescaled to 14 bits, as we did for all the other filter coefficients. Then,

$a_1^2 = 2.509 \cdot 10^{-2}$ that rescaled as 14 bit fixed-point becomes $a_1^2 = 205$. Observe that this coefficient is smaller than all the others, so it will actually have less significant bits than 14, thus allowing to reduce the number of significant bits in the multiplication result.

The output of the multiplier, fb_tmp_0 is on 14 bits plus the bits associated to the w signal. In order to be compliant with the previous implementation, and taking into account that the result of the multiplication in the feedback loop (by a_1^2) is truncated, it is reasonable to extend w on 15-bits, in order to manage a possible overflow in the sum operation. As we described in the previous chapter, a worst-case analysis is performed to carefully size the range of bits to save. The signal fb_tmp_1, that is the result of the multiplication $x[n] \cdot a_1$ is on 28 bits, but they are significant up to bit 24, since $|1298 \cdot (-2^{13})| < 2^{24}$. Thus, fb_1 that is the truncated version of fb_tmp_1 will start from bit 24.

The same considerations did in the previous chapter can be extended here to the signals ff_1 and ff_0 for which the most significant bit to be taken into account from the multiplication result is the bit 26, with no constraints on w value, 25 if we consider that w will never assume the maximum value (as it is the result of a sum where an operand is on less than the full range).
The multiplication by a_1^2 will give a most significant bit that is actually at position 22, considering w equal to -2^{14}.

At this point, we need to understand where to truncate. The approach is to relate to what was found in the C implementation and eliminate the bits below 20, while still guaranteeing the required precision and THD.

The following listing shows the assignment of the outputs of the multiplications to the truncated signals. Notice that signal fb_0 (output of multiplier by a_1^2) is starting from 21 and going below 20. Due to the fact that w is the sum of a number on 14 bits (input sample) and fb signal, that is on 6 bits, its value is at most $2^{13} + 2^5$ from which it can be derived that fb_tmp_0 (result of the multiplication under analysis) most significant bit is actually bit 21,carrying the sign. Moreover since the result of the sum of fb_0 and fb_1 past values is on $nb\_fb = 6$ bit we decided to extend fb_0 on 5 bits as well, since in any case the other input is on 5 bits.

```
constant r : integer := 7;
constant nb_fb : integer := 6;
...
fb_1 <= fb_tmp_1(24 downto 20);
ff_0 <= ff_tmp_0(20 + nb_r-2 downto 20);
ff_1 <= ff_tmp_1(20+nb_r-2 downto 20);
fb_0 <= fb_tmp_0(21 downto 21-nb_fb+2);
```

It is moreover noticed that of the two contributions added up giving fb the most significant one is quite always coming from the left branch, except for when w_{past} carries a big value with respect to the entering sample. Obviously this is due to the fact that the right branch comes from a multiplication for a quite small coefficient compared to the other one.

As in the basic implementation, the output signal is re-extended on the original bit-width (14 bits) by appending to the right of ff signal $r = 7$ zeroes.

```
DOUT <= ff & zero_padding;
```

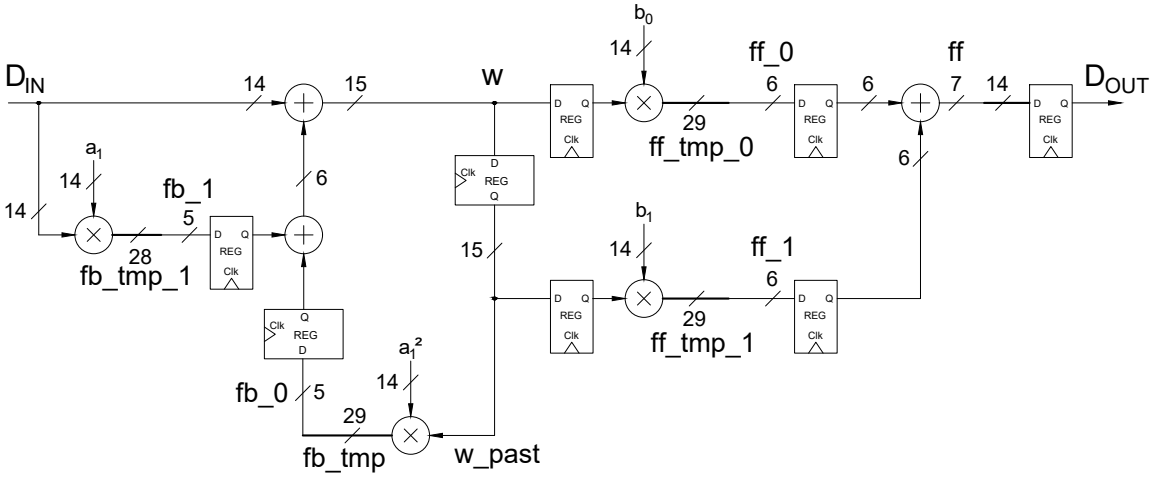

Figure 4.3: IIR filter, retiming and pipeline

4.3.2 Pipeline

To complete the optimization of our digital design, generally after look-ahead a combination of retiming and pipelining is applied, to obtain the best performances.

We have analysed the DFG derived from retiming, showed in figure 4.2. Following the rule based on *feed-forward cut-set* pipeline can be applied, in order to reduce the length of the critical path, in the feed-forward part, where there are no loops. In particular we can split the critical path going from the input to the output thanks to additional registers placed in-between. The resulting DFG after pipeline application is shown in figure 4.3.

The pipeline is composed by two stages, and now the critical path has a delay $t_d = max\{T_A, T_M\}$, with a consistent improvement with respect to the previous architecture. Also, the addition of the pipeline requires to modify the "propagation" of the *VOUT* signal, that is the validation bit for the outputs. It needs to be delayed accordingly to the number of pipeline stages introduced.

There is now the need to make sure that when an invalid input sample comes (i.e. *VIN*=0), the output samples related to the two previous inputs are still correctly detected.
The flip-flop that has *DOUT* as output, and the two flip-flops in the last stage of the pipeline will not be "enabled" by the *VIN* signal directly, but by its delayed version. In particular, *DOUT* register will be enabled by a delayed version of 2 clock-cycles, while the other two registers, in the forward branches, will be enabled by *VIN* delayed only of one clock-cycle. Below, some portions of the VHDL code implementing this feature are reported.

```
pipe_reg: process(clk)
begin
    if(clk='1' and clk' event) then
        ...
        elsif (VOUT_pre2 = '1') then
      ff_0_past <= ff_0;
      ff_1_past <= ff_1;
        ...
end process;

output_reg: process(clk)
```

```
begin
    if(clk='1' and clk' event) then
        ...
        elsif (VOUT_pre1 = '1') then
        DOUT <= ff & zero_padding
        ...
end process;

-- Process for delaying the input signal Vin and controlling the registers in the pipeline

delay_vin: process (clk)
begin
  if(clk='1' and clk' event) then
    if (RST_n ='0') then
      VOUT_pre2 <= '0';
      VOUT_pre1 <= '0';
      VOUT <= '0';
    else
      VOUT_pre2 <= VIN;
      VOUT_pre1 <= VOUT_pre2;
      VOUT <= VOUT_pre1;
    end if;
  end if;
end process;
```

This last feature allows for the architecture to take into consideration the pipeline for the enable signal at the input of the filter.

To summarize, there have been three different implementations of the IIR filter with look-ahead:

1. IIR filter with look-ahead and retiming, represented in figure 4.2.
2. IIR filter with look-ahead and pipelining, represented in figure 4.3.
3. IIR filter with look-ahead and pipelining, but moving the registers connected to *fb_0* and *fb_1* to the output of the adder between these two signals. This architecture is represented in figure 4.4.

The following section refers to the simulation results of the IIR filter with the look-ahead architecture.

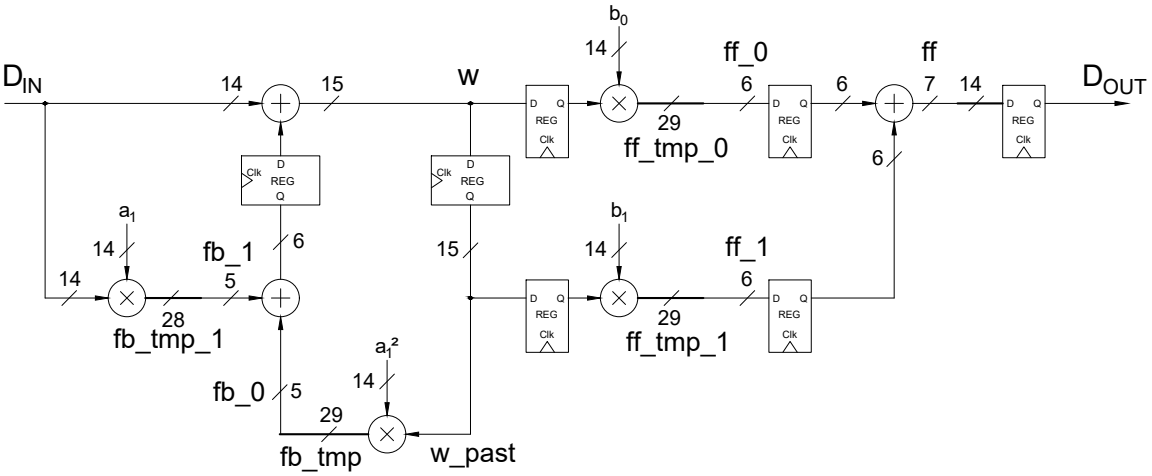

Figure 4.4: IIR filter, pipeline

4.4 Simulation

For the simulation of the IIR Filter with the look-ahead implementation the same testbench of the previous chapter was used. Then, the testbench is still composed of the four building blocks: *clk_gen*, *data_maker*, *IIR_FILTER* and *data_sink*. There is only a small change in the *data_maker*, since now this block needs to provide an additional coefficient, corresponding to a_1^2. This coefficient is added with the aforementioned value using the function conv_std_logic_vector as shown in the following listing:

```
H4 <= conv_std_logic_vector(205,nb_a); -- a1^2
```

Notice that the number of bits in this case is defined by the constant *nb_a*.

The three implementations of the IIR Filter discussed in the previous section (figures 4.2, 4.3 and 4.4) were simulated using the proposed testbench. The output samples of the three implementations showed a perfect match with respect to the values at the output of the C implementation.

Also, the behaviour of the filter was tested even for the case where *VIN* goes to zero. As expected, the filter is able to respond to this event and it is able to interpret that the incoming samples are not valid. The interesting case corresponds to the case where the pipeline is implemented and, even if *VIN* is zero, then there are still some values within the the pipeline that are valid and need to be processed. This case is represented in figures 4.5 and 4.6. These figures, in contrast with the results shown in figures 3.2 and 3.3, show that the signal *VOUT* is delayed with respect to the arriving signal *VIN*. In this case, due to the three registers used in the path from *DIN* to *DOUT*, we see that the changes in the signal *VOUT* arrives three clock edges after the signal *VIN* changes. Notice also that even after the arriving data is not valid, the output is still valid since it is showing the results of the inputs that are being processed by the pipeline.

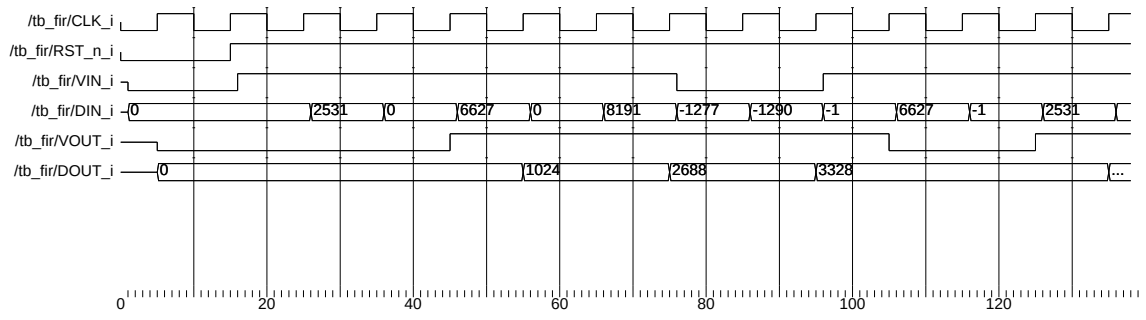

Figure 4.5: Start of simulation (look-ahead with pipeline)

Then, after verifying the correct behaviour of the three architectures proposed for the IIR filter with look-ahead, we can proceed with the synthesis of the circuits.

4.5 Logic Synthesis

The synthesis of the IIR filter with look ahead will allow us to highlight the differences among the three different used implementations of the filter. Similarly, it will be possible to tell if the obtained results regarding the delay of the architectures correspond to the expected delay obtained from the analysis of the architectures.

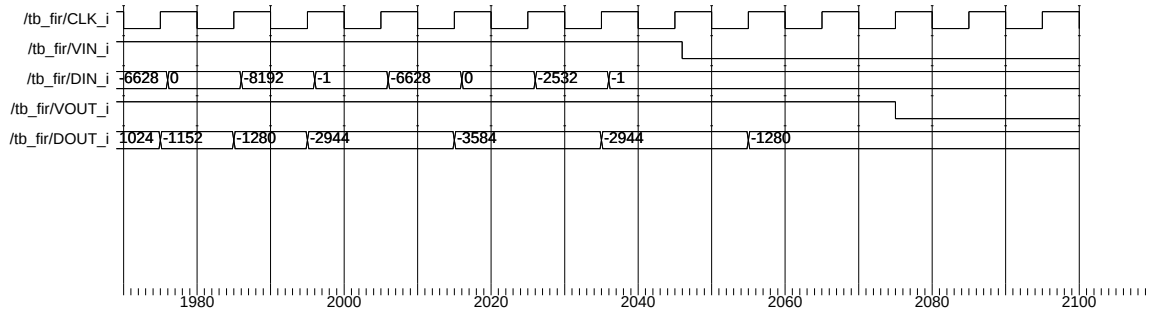

Figure 4.6: End of simulation (look-ahead with pipeline)

For the logic synthesis we use the same steps discussed in the previous chapter. Then, the results of the synthesis of these three architectures are summarized in the table 4.1. The most immediate result corresponds to the fact that the fastest of the three architectures is the second architecture. This result corresponds to the expected one, since the delay is only $max\{T_A, T_M\}$, i.e. the maximum between the adder and the multiplier delay. However, this architecture is also the largest among all, this is because this architecture contains the largest number of registers, as shown in figure 4.3.

Similarly, it is also expected that the slowest among all the architectures is the one that does not include pipeline in the circuit. This is because, as shown in figure 4.2, this architecture has a critical path that contains three adders and a multiplier. But also, this is the architecture that contains the least number of registers among the three possibilities, therefore, the results show that the total area of the cells is the minimum among the three architectures.

Finally, table 4.1 confirms the hypothesis that the architecture shown in figure 4.4 exhibits another point in the trade-off between area and frequency. Since this architecture has a faster critical path than the architecture without pipeline but also uses less registers than the architecture with pipeline.

Based on the previous results, the architecture chosen for performing the place and route corresponds to the fastest architecture, i.e. the architecture with pipeline and the largest number of registers. The following section summarizes the results of the place and route process.

Architecture	**Timing (Tmin)**	**Total cell area**
Look-ahead + Retiming (Fig. 4.2)	3.10 ns	3467
Look-ahead + Pipeline (Fig. 4.3)	2.22 ns	3769
Look-ahead + Pipeline - Alt (Fig. 4.4)	2.35 ns	3682

Table 4.1: Comparison of the synthesis results for the different architectures.

The results of the synthesis in terms of power and timing reports are contained in the folder *VHDL_synth/look_ahead/syn*. The files ending with NoPipe, Pipe and Pipe2 show the results of the three architectures summarized in table 4.1 respectively. The netlist resulting from the synthesis is contained in the file *synth_netlist.vhdl*, this corresponds only to the netlist resulting from the synthesis of the fastest architecture. The behaviour of this netlist file was tested, and the results showed that the behaviour of the filter was the same of the pre-synthesis case. This results were confirmed by the output shown by the filter.

4.6 Place and Route

The place and route of the circuit was performed following the steps provided by the guide, then, the tool used was Cadence Innovus. All the files related to the place and route are stored in the folder 'innovus'. The design has been saved under the name *IIR_FILTER*.

Figure 4.7 presents an screenshot with the result obtained by Cadence Innovus after following the place and route process. The initial netlist for performing the routing corresponds to the architecture synthesized with a constraint about minimum period equal to 8.88 ns, which corresponds to four times the absolute minimum period shown in table 4.1. Then, the area obtained using this constraint corresponds to 3541 units, much lower than the case with the more rigorous timing constraint.

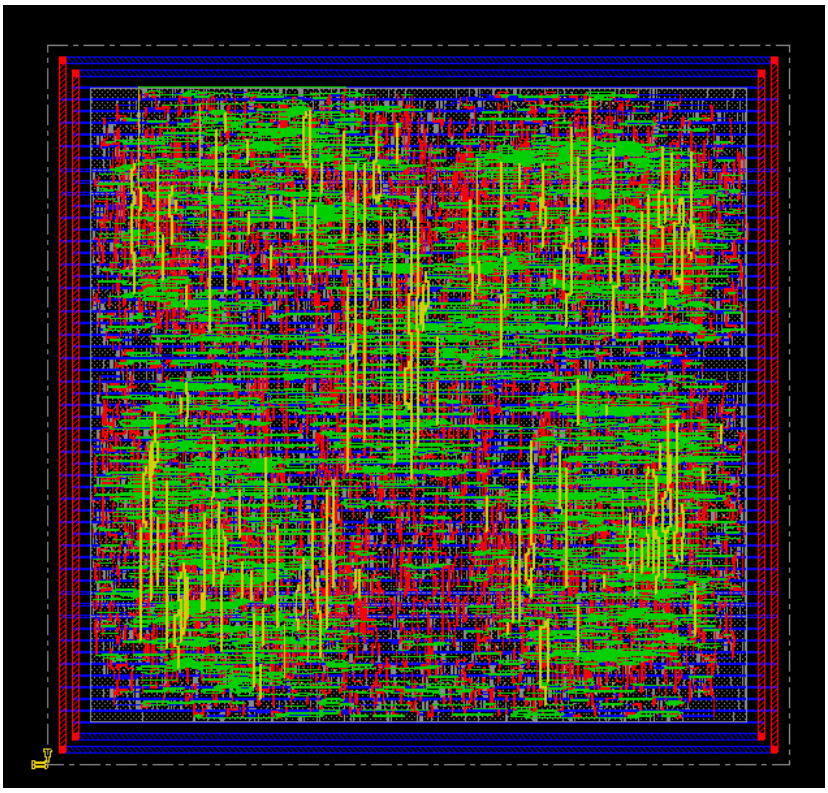

Figure 4.7: Place and Route result of the IIR Filter with look-ahead.

The timing reports of this architecture are stored in the folder 'timingReports' and they show no timing violation after performing the place and route. The check for violations was performed after the Clock-Tree Synthesis (CTS) and also for the post-route stage, both for the setup and hold time. This explains why the folder contains several files corresponding to the reports. From reading the files with the extension *.summary.gz* it is possible to recognize that there are no violations to the timing constraints that have been set.

Regarding the area results, the place and route report shows the following counts:

- **Gates** = 4427
- **Cells** = 1769

- **Area** = 3533.3

Observe that the resulting area is very close to the estimation done during the synthesis step.

The post-place-and-route netlist is exported into a verilog file under the name *IIR_FILTER.v*, this file is exported after the RC extraction. Exporting the netlist allowed to perform a check of the behaviour of the filter using the previously described testbench. Also, during the simulation, the information about the net activity was annotated, the switching activity allows the possibility to perform a better analysis of the power estimation of the circuit. These are the results of the power estimation of the IIR Filter for the total number of cells included in the design:

- **Internal Power** = 0.7 mW
- **Switching Power** = 0.53 mW
- **Leakage Power** = 0.07 mW
- **Total Power** = 1.3 mW

  Other details about the power report can be found in the file *IIR_FILTER.rpt*.

Notice that the scripts and bash files used for automatizing some steps of the process of synthesis and place and route are found in the folder. Usually the scripts have the extension .scr.