Politecnico di Torino

Integrated Systems Architectures

Report for Lab 1

https://github.com/levnikolaevicmiskyn/ISA/tree/main/lab1

Group 12

Antonio Carlucci 276128
Gabriele Perrone 269089
Alessandro Scisca 276032

Contents

Chapter 1

Reference Model

1.1 Project Specifications

The aim of this project is to develop both the software model and a VLSI implementation of an **IIR digital filter**. According to the provided instructions, the project specifications are listed in Table 1.1.

Parameter	Symbol	Value
Cutoff frequency	f_c	2 kHz
Sampling frequency	f_s	10 kHz
Filter order	N	1
Data width (number of bits)	n_b	8

Table 1.1: Project specifications

1.2 Filter Parameters

Given the project specifications, it is possible to draw the **Data Flow Diagram** as shown in Figure 1.1.

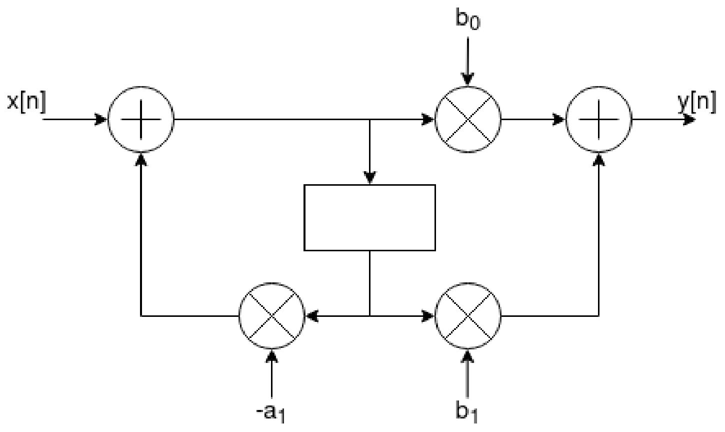

Figure 1.1: Target filter data flow diagram

This DFD implements the Direct Form II of an IIR filter. In this specific case, it implements the following equations:

$$\begin{cases} w[n] &= x[n] - a_1 w[n-1] \\ y[n] &= b_0 w[n] + b_1 w[n-1] \end{cases} \tag{1.1}$$

It is thus necessary to determine the values of the a and b parameters. Thanks to the provided Matlab script `myiir_design.m`, the derived results are:

$$a = [-0.1641] \tag{1.2}$$
$$b = [0.4141,\ 0.4141] \tag{1.3}$$

Following the project specifications, the filter is going to be implemented in hardware and will use an 8 bit **fixed point** representation: this simplifies the structure of the filter, but limits its precision. Moreover, the internal structure of the filter will be developed to prevent overflows by choosing a wide-enough parallelism in order to maximize the effectiveness of the filter.

1.3 Reference Model

To properly verify the functionality of the filter, it is necessary to have a reference model that also works with 8 bit fixed point numbers. In addition, this allows for an insightful comparison of the filter's behavior when it is working in floating point (in Matlab) versus fixed point.

Truly fixed point operations are modeled in a C program where coefficients and data are stored using the standard `int` data type. A scale factor equal to 2^{n_b-1} is implied throughout the execution. The loss in precision arising from truncating the least significant bits at the output of every hardware multiplier is simulated in C using the right-shift operator `>>`.

As shown in Equation 1.1, there is a subtraction among the operations. To simplify the hardware, it is easier to transform the operation in an addition by changing the sign of a_1, as in

$$w[n] = x[n] + (-a_1)w[n-1] \tag{1.4}$$

To better mirror this behavior, some small changes were necessary in the source files: first of all, a_1 had to be sign-flipped, then, inside the `myfilter` function, this section

```c
for (i=0; i<N; i++) {
  fb -= (sw[i]*a[i]) >> (NB-1);     // Subtraction here
  ff += (sw[i]*b[i]) >> (NB-1);
}
```

was changed into this

```c
for (i=0; i<N; i++) {
  fb += (sw[i]*a[i]) >> (NB-1);     // Addition here (a[i] changed sign)
  ff += (sw[i]*b[i]) >> (NB-1);
}
```

To clarify the difference, suppose to have:

$$\texttt{NB} = 8 \qquad \texttt{a[i]} = \pm 21$$
$$\texttt{fb} = 0 \qquad \texttt{sw[i]} = 1$$

and notice how the results differ depending on the chosen operation and the actual sign of `a[i]`:

$$\begin{array}{ll|ll}
0 - (1 \times -21)\texttt{>>}(8-1) & = & 0 + (1 \times -(-21))\texttt{>>}(8-1) & = \\
0 - 11101011_2\texttt{>>}7 & = & 0 + 00010101_2\texttt{>>}7 & = \\
0 - 11111111_2 & = & 0 + 00000000_2 & = \\
1 & & 0 &
\end{array}$$

1.4 THD

Rounding the input samples to the closest representable value according to the adopted bit width is a nonlinear operation that introduces distortion in the signal, producing spurious harmonics on top of the pure sinusoidal wave expected at the output of a linear filter. The relevant parameter to characterize this effect is the *total harmonic distortion* (THD). The dependence of the THD on the internal bit width has been calculated from simulations, with the results reported in Figure 2.2.

From these results we conclude that the smallest implementation in terms of area occupied by arithmetic operators requires $n_b = 7$ in order to keep the total distortion below $-30\,\text{dBc}$, as requested by the specifications.

Chapter 2

Standard Architecture

2.1 Datapath

The datapath has been derived directly from the DFG from Figure 1.1, obtaining the architecture shown in Figure 2.1.

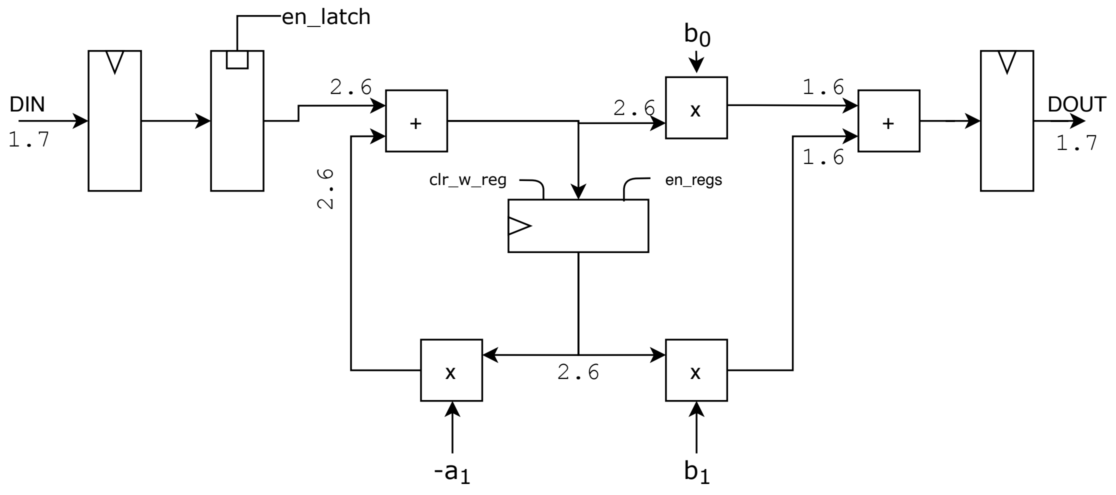

Figure 2.1: IIR filter datapath

The specified timing requires input data to be sampled on the same clock cycles where `VIN` is asserted. For this reason, the input register `R1` is always enabled. The most convenient solution to carry on operations only when data is actually valid is to place a latch at the output of `R1`.
This is because appending another register instead of a latch will increase the overall latency; while selectively enabling the internal registers complicates their internal structures. Furthermore, the input latch will also perform a gating function (**guarded evaluation**), preventing the waste of power in the internal combinational nodes when the input samples are not valid. The only internal register that actually needs to be disabled is the one storing w_{n-1} in the feedback path, controlled by the `en_regs` signal.

Overall, the only real commands needed from the control unit are `en_latch`, `en_regs` and `clr_w_reg`, since the `W` register should still be initialized.

2.1.1 Accuracy

From the preliminary investigation in section 1.4, it is possible to see that $n_b = 7$ is enough to meet the requirements. This means that dropping the least significant bit from the input samples (represented on 8 bits) can simplify the internal computations with an acceptable degradation in performance.

The analysis in section 1.4 has been carried using a model where numbers include $n_b - 1$ fractional bits (the binary digits with weights 2^{-1}, 2^{-2}, ..., 2^{n_b-2}), therefore it is expectable that allocating 6 fractional bits for all the internal variables in the VHDL implementation will provide enough accuracy.

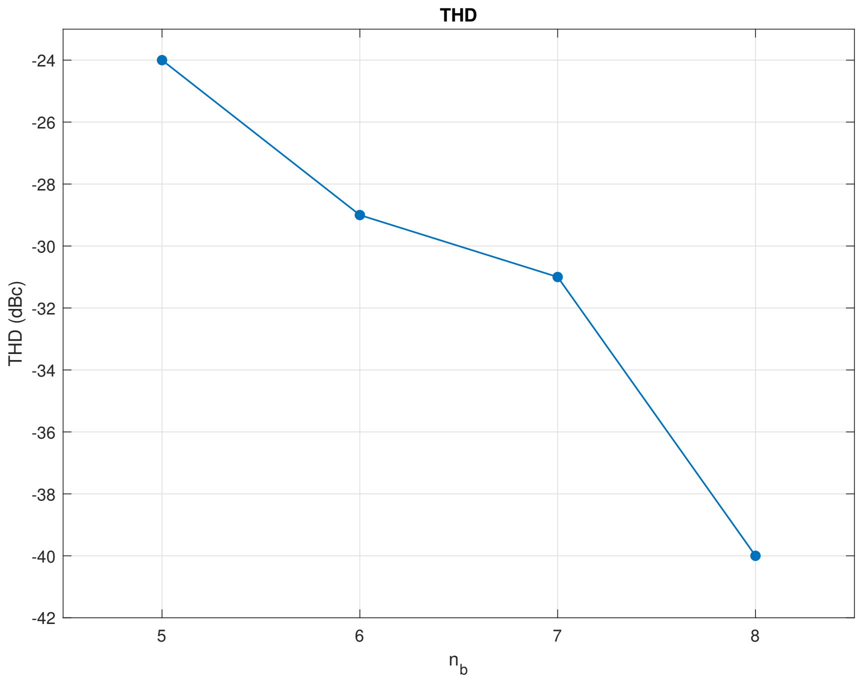

Figure 2.2: THD as a function of n_b

2.1.2 Avoiding overflow

The C reference model is accurate in predicting the error introduced by truncation, which affects the output of every multiplier block, but it neglects the inability to represent a number larger or equal to 1 in absolute value with the single integer bit available in the standard fixed-point format. An overflow condition may occur in intermediate steps of the computation where the result would require more than one integer bit. In order to determine the right sizing that guarantees the absence of overflow, the maximum value of the intermediate variable $w[n]$ must be determined. Using Equation 1.1,

$$w[n] = \sum_{i=0}^{n} (-a_1)^i x[n-i]$$

Hence, assuming $|x[n]| \leq 1$,

$$|w[n]| \leq \sum_{i=0}^{n} |(-a_1)^i x[n-i]| \leq \sum_{i=0}^{n} |a_1|^i \leq \frac{1}{1-|a_1|} \approx 1.2$$

According to this computation, two integer bits are enough to avoid overflow at the nodes where $w[n]$ is processed within the DFG. As a consequence, the adder and multiplier in the feedback loop in Figure 1.1 will operate on inputs with 2 integer bits and 6 fractional bits, as already determined by the previous reasoning on the internal accuracy.
As for the operators in the feedforward part, multiplying $w[n]$ and $w[n-1]$ by b_1 and b_2 will always produce a result lower than 1 in magnitude, for which a single digit to the left of the radix point suffices. Therefore the output of those multipliers is resized to match the same format used by the

final adder consisting of 1 integer and 6 fractional bits. The eighth digit, which is always zero, is appended to the final result, thus becoming its LSB, to comply with the specified interface format. A summary of the internal parallelism used throughout the filter is reported in Figure 2.1, where the format $a.b$ stands for a integer bits and b fractional bits. All operators are assumed to accept operands and to provide the result in the same format: a resizing block is implied on edges where different format labels are specified at the endpoints.

Transfer function A comparison of transfer functions obtained in floating point arithmetic (using Matlab) and fixed point with varying bit width is shown in Figure 2.3. The DC gain of the filter decreases as n_b is decreased, this is due to the truncation of results done after several of the operations involved in the actual implementation. Integrating the losses incurred in while removing the least significant bits results in a gain loss. The transfer function was obtained by computing the FFT of the impulse response (discrete).

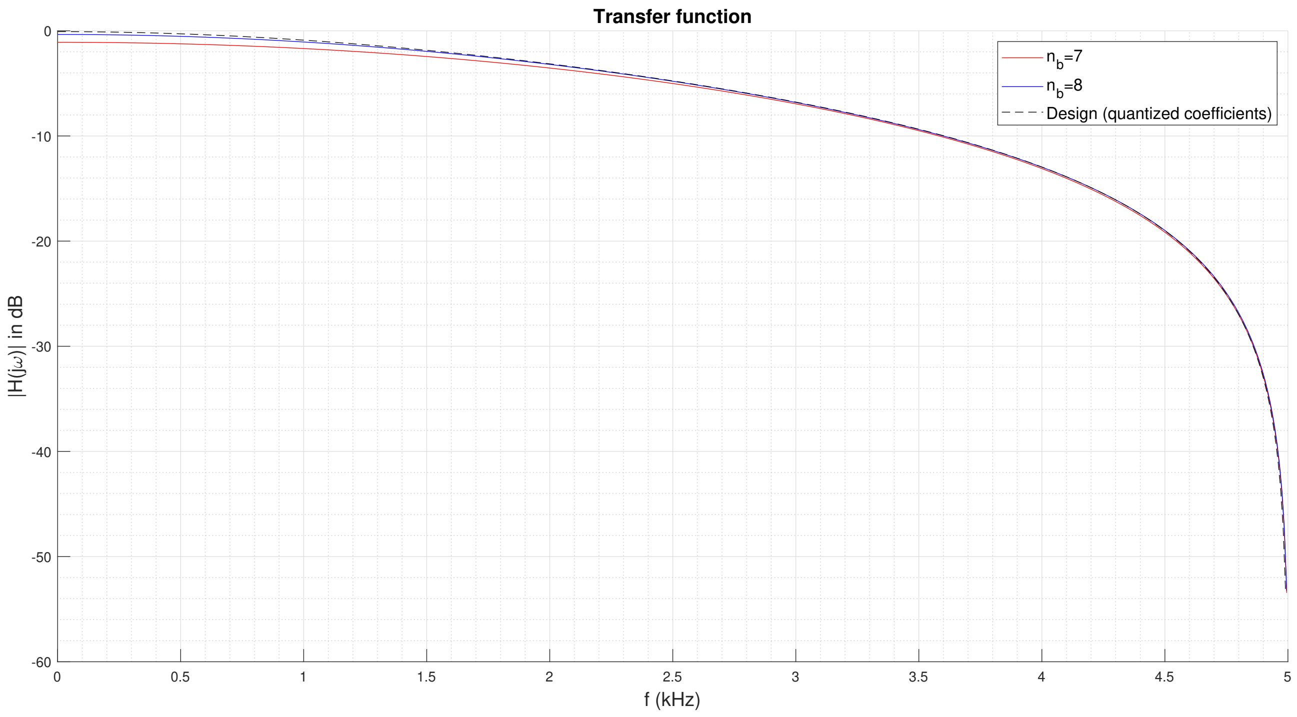

Figure 2.3: Transfer function for a few values of n_b

2.2 Control Unit

The datapath derived from figure Figure 1.1 requires two control signals to clear all registers and to enable the latch. A single status signal `VOUT` must be provided, as requested. The resulting ASM chart describing the state machine is in figure Figure 2.4.

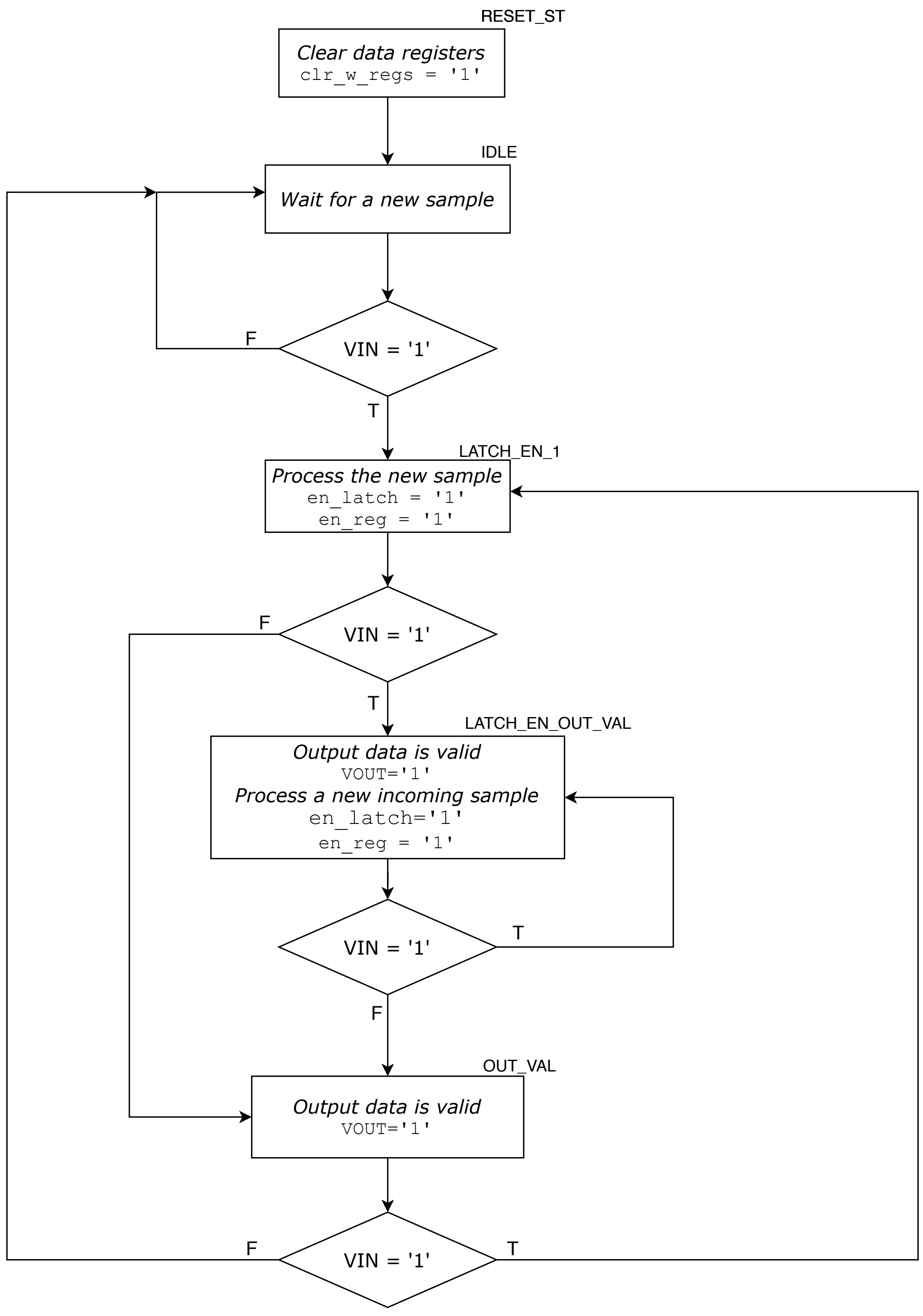

Figure 2.4: ASM chart representing the state machine for the standard architecture. The values of the control signals are indicated along with comments describing the purpose of each state

Chapter 3

Verification

3.1 Methodology

The task of verifying a design using a logical simulator such as ModelSim can be broken down into four distinct phases:

1. Generation of suitable input vectors
2. A number of logical simulations where the *device-under-test* is operated within a testbench environment, providing it with the generated inputs and recording its outputs
3. A number of reference software runs, where the ideal behavior and the reference results are generated
4. Analysis and comparison of the output data against the reference results

Targets In order to plan the verification flow, all aspects of the design that need to be tested must be identified. In the case of a data processing unit like the one we are considering, the correctness of the **output data** stream must be checked along with the **timing** of every control and status signal.

VHDL testbench structure The testbench is made up of the following non-synthesizable components:

- Clock generator (`clkGen.vhd`): it generates a clock signal whose period is parametrized by a constant. If vertools is used, this constant can be written in vertools configuration file (`vertools.config`).
- Data generator (`dataGen.vhd`): it fetches the input samples from a text file to feed the DUT at regular intervals, with a frequency submultiple of f_{clk}.
- Monitor (`dataSink.vhd`): it collects output samples in a text file for further analysis and verifies that the status signal `VOUT` is asserted with the right timing. This is done by probing the control signal `VIN` and ensuring that `VOUT` is issued after a latency period (in clock cycles) given as a constant and dependent on the internal architecture. If any unexpected behavior is detected, a warning message appears on the output log.

The described verification steps are executed with an automated Python script, discussed in section 3.2.

3.2 Vertools

Vertools is a simple verification tool written in **Python** used to aid the verification process. It automatically calls the necessary commands, checks the outputs, manages logs and does general cleanup operations before and after each run.

Vertools is an high level utility, meaning that it is not intrusive: it avoids, whenever possible, to edit any of the files used by the other programs or to perform similar operations. The only instance where Vertools edits the content of a VHDL file is during the simulation phase, where it sets the clock period in a specified clock generator.
This is not only because the verification process is logically supposed to not change the sources, but also because letting a (newly written) software fiddle with such sensitive files is dangerous. This leaves some operations which could, in theory, be implemented with an additional parameter to the hands of the user. For example, if multiple architectures are defined for the same component, it is up to the user to compile the right file or to manually edit the compilation command called by Vertools.

Before going into the details of its usage, it is necessary to correctly prepare the environment for its usage.

3.2.1 Setting up the environment

Vertools uses external Python packages which can be installed through Python's package manager, **pip**. To install them,

```
$ cd /path/to/vertools/
$ python3 -m pip install -r requirements.txt
```

In shared machines, it could be better to install those packages for the current user only by adding the `--user` flag in the `python3 -m pip` command.

For Vertools to be able to run, it is necessary to add the path of the package folder to the `PYTHONPATH`.

```
$ export PYTHONPATH=${PYTHONPATH}:/path/to/vertools/
```

Finally, Vertools is a Python package, so it should be called with `python3 /path/to/vertools/`. It can be convenient to create an alias:

```
$ alias vertools="python3 /path/to/vertools/"
```

For simplicity, from now on it is assumed that the alias has been created and `vertools` will be treated as a command of its own.

3.2.2 Subcommands

To improve the granularity of the operations, the tool's functionality is divided in various levels of subcommands. Beside the main `vertools verify` command, which runs the whole verification process (simulation, reference, comparison), the user can call:

- `vertools generate-inputs`: generate the input vectors choosing among various customizable waveforms.
- `vertools simulate`: set the simulation parameters (like clock period), call the provided simulation command (usually something like `source simulate.do`) save the logs and check if the expected output was generated.
- `vertools reference`: call the provided reference command (in this case, the C executable), store logs and check if the expected output was generated.
- `vertools compare`: compare the simulation and reference outputs, checking first and foremost if the lengths of the produced files are compatible, and then comparing them line by line.
- Other more specific commands which are not worth exploring here, but they can all be reviewed with `vertools --help`.

Each subcommand can have its own flags, parameters and even another level of subcommands. For instance, `generate-inputs` has a subcommand to specify which waveform should be created. Each waveform has a different number of parameters with different meanings. A specific help for each level of subcommand can be consulted by calling it with the `--help` flag, as in `vertools generate-inputs sine --help`, which will show the order in which to specify the sine wave parameters.

3.2.3 Configuration files

To grant some level of customizability and reusability, Vertools' operations are parametrized. It is possible to store those verification parameters in a configuration file: this way, it is not necessary to specify them by hand at each run. Because Vertools works with the concept of *scoping*, the user can override specific parameters through the appropriate command line options without actually editing the configuration file.

By default, Vertools looks for a file named `vertools.config` in the directory where the command is called. It is also possible to name the configuration file in other ways: this allows to choose between multiple configurations. In that case, the file should be specified with

```
    $ vertools --config filename.config COMMAND [ARGUMENTS ...]
```

Configuration files support different **sections**, so that the same variable can have different values in different context (for example, `tstep` might be different between the input generation and the actual simulation step) and **interpolation**, meaning that it is possible to expand a variable in the assignment of another variable, bash-style.

3.2.4 Program structure

The program is composed of multiple modules, responsible to implement its various parts. Some of the modules can be customized to provide additional or ad-hoc functionalities.

cli.py This is the module setting and managing the Command Line Interface. It uses the default `argparse` library which offers a helpful interface to handle command line arguments and flags and to generate their help menus without the need of manually implementing a custom basic interpreter. As an example, the following lines of code generate the top level command (`vertools`) and its first subcommand (`generate-inputs`/`simulate`/...).

```
vertools = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    description='High level simulation tool for digital circuit verification'
)
vertools.add_argument(
    '--config',
    help='configuration file',
    metavar='FILE',
    dest='local_config',
    default=None
)
subparsers = vertools.add_subparsers(
    title='command',
    description='vertools command'
)
```

commands.py The actual command procedures are implemented here. All commands are implemented as classes inheriting from the custom `CommandAPI` class, which exposes three methods - `setup`, `run`, `exit`. These methods implement the respective steps of the execution of a command and, by default, are all called in sequence when the command is requested. This division can be helpful, for instance, when multiple commands share the same setup steps, allowing the user to define them only once. Furthermore, the setup function can also halt the execution of the command by returning `False` when something goes wrong.

```
class CommandAPI:
"""Program command base class
Attributes:
    args (List): command arguments
    context (vertools.Context): contextualized parameters
    verbose (bool): flag to allow or block the command's output
    data (dict): custom data shared between the command phases
"""
def __init__(self, args, context, verbose=True):
    self.args = args
    self.context = context
    self.verbose = verbose
    self.data = {}

def setup(self):
    """Initialize files, context, variables, ...
    Returns:
        bool: True if command needs to continue, False to abort
    """
    return True

def run(self):
    """Run core actions"""
    pass

def exit(self):
    """Post execution actions"""
    pass

def output(self, output_func=output.update, *args, **kwargs):
    """Generate an output through a generic function only if command is set to
        verbose
    Args:
        output_func (function): output function (usually print or output.update)
        *args: output_func arguments
        **kwargs: output_func keyword arguments
    """
    if self.verbose is True:
        return output_func(*args, **kwargs)

def __call__(self):
    """Run the command by calling the setup, run and exit methods in order.
    If the setup method returns false, the command aborts.
    """
    setup_status = self.setup()
    if setup_status is False:
        return
    self.run()
    self.exit()
```

system.py This module provides some wrapper functions to build an interface with the system. For example, the `run_bash` function (which is the one used to call the simulation and reference commands) handles all the necessary steps to run an external bash command in a subshell and to redirect its output either to logs or `/dev/null`, when they are disabled.
It can receive a list of strings containing the bash commands which will all be executed in the same shell, which is useful for when some environment variables are shared.

```
def run_bash(commands, **kwargs):
    """Run a bash command
    Args:
        commands (Union[str, List[str]]): command (or list of commands) to be
            executed in the same shell
        **kwargs: arbitrary keyword arguments
    Returns:
       subprocess.CompletedProcess
    """
    stdout = kwargs.get('stdout', None)
    stderr = kwargs.get('stderr', None)
    if stdout is False:
        stdout = subprocess.DEVNULL
    if stderr is False:
        stderr = subprocess.DEVNULL
    if isinstance(commands, list):
        command = ' && '.join(commands)
    else:
        command = commands
    status = subprocess.run(command, shell=True, stdout=stdout, stderr=stderr)
    return status
```

context.py Module responsible for managing the scoping and context functionalities, which allow the user to work with configuration files and override some of the stored parameters through the command line. There are two fundamental classes, `Scope` and `Context`, which manage an arbitrarily deep parameter lookup hierarchy, maintaining the useful configuration files concept of sections.

waveforms.py Module to generate the input data with various waveforms through the `scipy` and `numpy` libraries.

Chapter 4

Synthesis

The minimum clock period is found to be $T_{min} = 2.74\,\text{ns}$ by doing several synthesis steps after the maximum frequency analysis in order to be able to meet the constraint. The timing report derived after synthesizing with $T_{clk} = 4T_{min} = 11\,\text{ns}$ is reported in Table 4.2. The command `report_area` returns the data shown in Table 4.1.

Combinational area	795.606
Buf/Inv area	34.314
Noncombinational area	242.857
Macro/Black Box area	0.000
Total cell area	1038.463

Table 4.1: Area report summary (unit is µm^2)), $T_{clk} = 11\,\text{ns}$

Point	**Incr**	**Path**
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
comp_dp/w1_reg[3]/CK	0.00	0.00
comp_dp/w1_reg[3]/Q	0.19	0.19
comp_dp/comp_mul_a1/a[3]	0.00	0.19
...		
comp_dp/comp_sum2/add_1_root_add_27_2/SUM[6]	0.00	2.90
comp_dp/comp_sum2/sum[6]	0.00	2.90
comp_dp/DOUT_reg[7]/D	0.01	2.91
data arrival time		2.91
clock CLK (rise edge)	11.00	11.00
clock network delay (ideal)	0.00	11.00
clock uncertainty	-0.07	10.93
comp_dp/DOUT_reg[7]/CK	0.00	10.93
library setup time	-0.03	10.90
data required time		10.90
data required time		10.90
data arrival time		-2.91
slack (MET)		7.99

Table 4.2: Timing report at $T_{clk} = 11\,\text{ns}$

4.1 Post-synth simulation

A simulation of the synthesized netlist accurately provides the switching activity for every internal net. The resulting power report is in Table 4.3. To observe how power consumption is

Power Group	Internal Power	Switching Power	Leakage Power	Total Power
io_pad	0.0000	0.0000	0.0000	0.0000 (0.00%)
memory	0.0000	0.0000	0.0000	0.0000 (0.00%)
black_box	0.0000	0.0000	0.0000	0.0000 (0.00%)
clock_network	0.0000	0.0000	0.0000	0.0000 (0.00%)
register	29.5835	1.5295	3.8171e+03	34.9300 (24.07%)
sequential	0.3539	0.2502	329.9172	0.9340 (0.64%)
combinational	49.0587	43.6288	1.6596e+04	109.2838 (75.29%)
Total	78.9960 uW	45.4085 uW	2.0743e+04 nW	145.1478 uW

Table 4.3: Power report (Synopsys) at $T_{ck} = 4T_{min}$

Total internal power	52.6260 uW
Total switching power	24.5493 uW
Total leakage power	2.0762e+04 nW
Total power	97.9375 uW

Table 4.4: Total dissipated power (bottom line in Synopsys power report) when `SAMPLING_FACTOR`=4

influenced by the data rate, it could be interesting to perform a power analysis corresponding to `SAMPLING_FACTOR=4`, that is when new data is available once every four clock cycles. The bottom line of the corresponding power report is in Table 4.4. The overall power is halved with respect to continuous operation.

4.2 Place & Route

Timing analysis The worst slack is 8.56 ns, slightly larger than the one found in Synopsys (8 ns). The same situation occurs with the look-ahead architecture.

Connectivity verification The are 0 warning or violations for this step.

Gate count Total area and cell/gate count is:

Gates	1143
Cells	459
Area	912.4 um^2

Power analysis Results are reported in Table 4.5. The total power is slightly larger than the one computed by Synopsys (Table 4.3), this could be expected since Innovus takes into account resistive and capacitive parasitics that will only increase power dissipation.

Group	**Internal Power**	**Switching Power**	**Leakage Power**	**Total Power**
Sequential	0.03623	0.001813	0.004147	0.04219 (18.21%)
Macro	0	0	0	0
IO	0	0	0	0
Combinational	0.09489	0.07816	0.01643	0.1895 (81.79%)
Clock (Combinational)	0	0	0	0
Clock (Sequential)	0	0	0	0
Total	0.1311	0.07997	0.02057	0.2317 (100%)
CLK	0.03587	0.001562	0.003817	0.04125 (17.81%)

Table 4.5: Power report after place & route: Group Power for Rail VDD and CLK

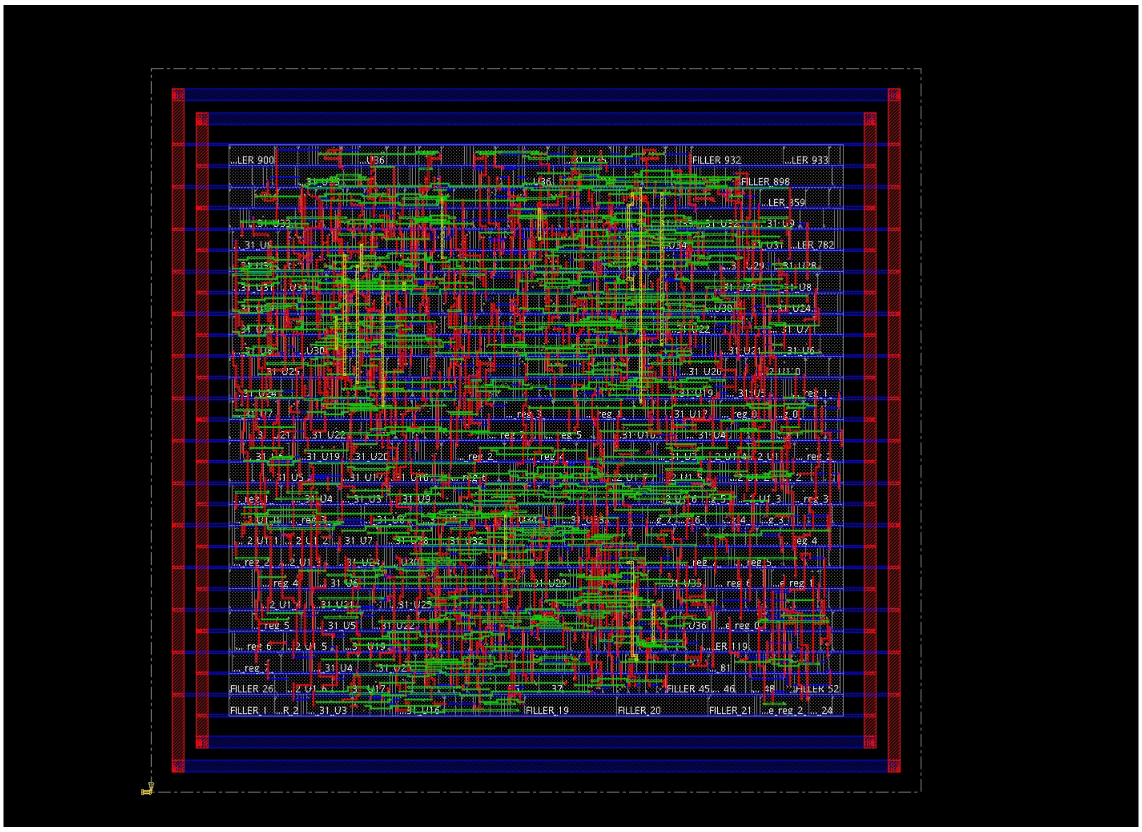

Figure 4.1: Snapshot after routing

Chapter 5

J-Look Ahead

5.1 Derivation

It is possible to further optimize the filter using the look-ahead technique, which simply consists in unrolling a recursive relation to obtain another equivalent one that depends explicitly on instants further back in time. The aim is to introduce a larger delay in the feedback loop so as to reduce T_∞, paving the way for universal techniques such as retiming.
Considering Equation 1.1, notice that $w[n-1]$ appears in the expressions for $w[n]$ and $y[n]$.

$$w[n-1] = x[n-1] - a_1 w[n-2] \tag{5.1}$$

Substituting Equation 5.1 back into both the equations that define direct form II (Equation 1.1):

$$\begin{cases} w[n] &= x[n] - a_1 x[n-1] + a_1^2 w[n-2] \\ y[n] &= b_1 x[n-1] + b_0 w[n] - a_1 b_1 w[n-2] \end{cases} \tag{5.2}$$

Equation 5.2 shows the result obtained by performing the same substitution in the output equation as well. It is clear that this is not convenient because the number of operators required in the feedforward part would be increased unnecessarily. In fact, the look-ahead technique is very useful in tackling loops that cannot be sped up using universal techniques either because there are too few registers involved or they might have an unacceptably large T_∞. It is not really well suited for purely feedforward structures within the DFG, where standard pipelining can effectively cut critical paths. In conclusion, the solution is to use the new expression for $w[n]$ (which includes a feedback) while retaining the unmodified $y[n]$. In this way, $b_1 x[n-1]$ does not appear in the equation for $y[n]$, saving one multiplier and one adder, while still allowing a complete optimization that, as will be discussed, brings the critical delay down to $T_{CP} = 1 \times T_{\text{mul}}$.
The **final set of equations** used in the look-ahead architecture is the following:

$$\begin{cases} w[n] &= x[n] - a_1 x[n-1] + a_1^2 w[n-2] \\ y[n] &= b_0 w[n] + b_1 w[n-1] \end{cases} \tag{5.3}$$

Its implementation and optimizations are explored in the following sections.

5.1.1 Retiming and pipelining

The preliminary result of a direct mapping of these equations into a DFG is in Figure 5.2. It is clear that pipeline registers can be placed on feedforward cut-sets in order to break critical paths. Since the feedback loop sets a lower bound on T_{cp}, retiming will be applied first in order to leverage the possibilities opened by the look-ahead transformation, and only then the correct distribution of pipeline stages will be determined accordingly.

Retiming As for the feedback loop, it is clear that look-ahead has changed T_∞ that has now decreased to $\frac{T_m+T_a}{2}$. Moreover, the introduction of a second register enables the use of retiming to bring the critical path closer the its lower bound (loop bound). By moving the pink register in the position indicated by the dashed arrow, thus separating the multiplier from the adder, T_{CP} is reduced to T_m (assuming $T_m > T_a$). Another slight optimization consists in replacing the two orange registers by a single one positioned at the output of the adder, as can be seen in figure Figure 5.3.

Pipeline stages The loop contains a critical path equal to a single multiplier delay, therefore the correct number and placement of pipeline register is to be determined so as to remove any feedforward path with a delay larger than T_m. It follows that there must be a delay element on every edge joining an adder and a multiplier ($T_m + T_a > T_m$). The stages indicated with the blue and brown dashed lines in figure Figure 5.2 serve exactly this purpose. On the other hand, the yellow line corresponds to a pipeline stage separating two adders: this is actually useful only if $2T_a > T_m$. According to timing reports collected during the synthesis of the standard architecture, DesignWare can provide operators with delays as low as $T_m = 0.82\,\text{ns}$ and $T_a = 0.32\,\text{ns}$ when constrained for maximum speed. These numbers suggest that the path formed by `A1` and `A2` without the interposed yellow register is not the critical one since the multiplier has still a slightly larger delay ($T_m = 0.82 > 2T_a = 0.64$). However, the implementation of fast operators might involve larger structures in terms of area (complexity). In short, there is a trade off and the optimal solution depends on the actual parameters of the available library cells and the specific design goals. In order to assess the impact of adding the yellow pipeline stage, the two cases have been synthesized separately. Since the aim is to discover the impact on performance (speed), these synthesis are run with a constraint on T_{clk} set to zero.

Parameter	**2 pipeline stages**	**3 pipeline stages**
Minimum clock period	1.05 ns	1.02 ns
Adder Area	51.9 (Buf/Inv 6.9)	48.4 (Buf/Inv 6.9)
Combinational Area	1389	1437
Buf/Inv Area	163	168
Noncombinational Area	466	505
Overall area	1856	1942
Dynamic power ($T_{ck} = 4.08\,\text{ns}$)	252 μW	260 μW

Table 5.1: Comparison showing the impact of the yellow register. Synthesis is performed for maximum speed for all parameters except for dynamic power, which has been derived at fixed f_{clk} for a fair assessment

Table 5.1 shows that the compiler finds a slightly faster solution (3%) when synthesizing the design with the additional pipeline stage. This is not due to a different critical path, because both timing reports list the path through a multiplier. It is more likely a consequence of the heuristic nature of the algorithm used to search the design space, which targets a global optimization with a complex cost function that accounts for several factors beside speed.
As for the area used up by adders, it becomes 7% larger if the yellow stage is removed, with buffers taking up more space. In fact, breaking the path `A1-A2` could also translate into a more relaxed constraint on the delay of each one of them, perhaps attainable with simpler and more compact architectures. However, the area used up by the register itself largely outweighs this reduction (see *Noncombinatinal area*), resulting in a larger overall area. Dynamic power consumption is 4% higher if the pipeline stage is added.
In conclusion, this comparison shows that an higher speed is obtained when all operators are isolated by pipeline registers, but this results in a slight increase in area and power dissipation. If the aim is maximum performance the alternative with $T_{min} \approx 1.02\,\text{ns}$ is the best choice, whereas a strategy that prioritizes area or power would lead to the opposite conclusion. The final DFG that corresponds to the actual VHDL implementation is in figure 5.3.

A formal approach to retiming Retiming can be easily done by inspection in this particular case, as shown in Figure 5.2. However, for larger designs this task cannot be handled without a more formal and algorithmic approach. The problem can be cast as a system of inequalities expressing the feasibility constraint for retiming, which states that the final number of delays on each edge must be nonnegative. Moreover, there has to be at least one register on paths joining

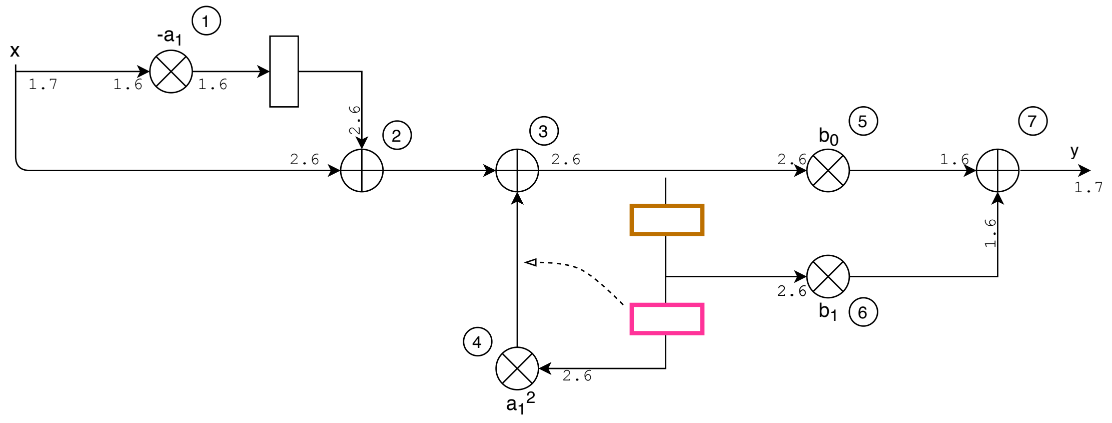

Figure 5.1: Numbering of operators for the retiming algorithm (system on inequalities)

operators whose combined delay is larger than the critical path. The introduction of pipeline stages can be equivalently thought of as inserting a number of input registers and the performing retiming to transfer them to the desired cut-sets. In this case this is convenient because both problems can be addressed using this formal approach. The system of inequalities is the following

$$
\begin{aligned}
&r(1) + \Delta \geq 0\\
&r(2) + \Delta \geq 0\\
&r(2) - r(1) + 1 \geq 1\\
&r(3) - r(2) \geq \epsilon\\
&r(3) - r(4) \geq 1\\
&r(4) - r(3) + 2 \geq 1\\
&r(5) - r(3) \geq 1\\
&r(6) - r(3) + 1 \geq 1\\
&r(7) - r(5) \geq 1\\
&r(7) - r(6) \geq 1
\end{aligned}
$$

Where ϵ can be set to 1 if a register should be in between operators 2 and 3 or to 0 otherwise. Δ is the (maximum) number of pipeline stages (input registers) to be added. A solution can be found by building the so-called *constraint graph*, where for each inequality $r_i - r_j < c$ a directed edge is drawn from j to i with a weight equal to c. An additional virtual node is connected to all the other nodes by edges of zero length. Finally, an all-point shortest path algorithm is executed on this graph and the result r_i can be found as the shortest path from the virtual node to node i. This algorithm has been implemented in the script `retiming_algorithm.m` which for this instance returns ($\Delta = 3$, $\epsilon = 1$):

$$r = [-3, -3, -2, -3, -1, -1, 0]$$

which is equivalent to the solution found by inspection (Figure 5.3).

5.1.2 Accuracy

This new computational structure has also been modeled with a C program in order to perform fast design analyses such as finding of the minimum number of fractional bits that provide the specified accuracy. Although in this case the total harmonic distortion has different values than those found previously, the simulation points towards the same conclusion that a minimum number of $n = 7$ bits is necessary.

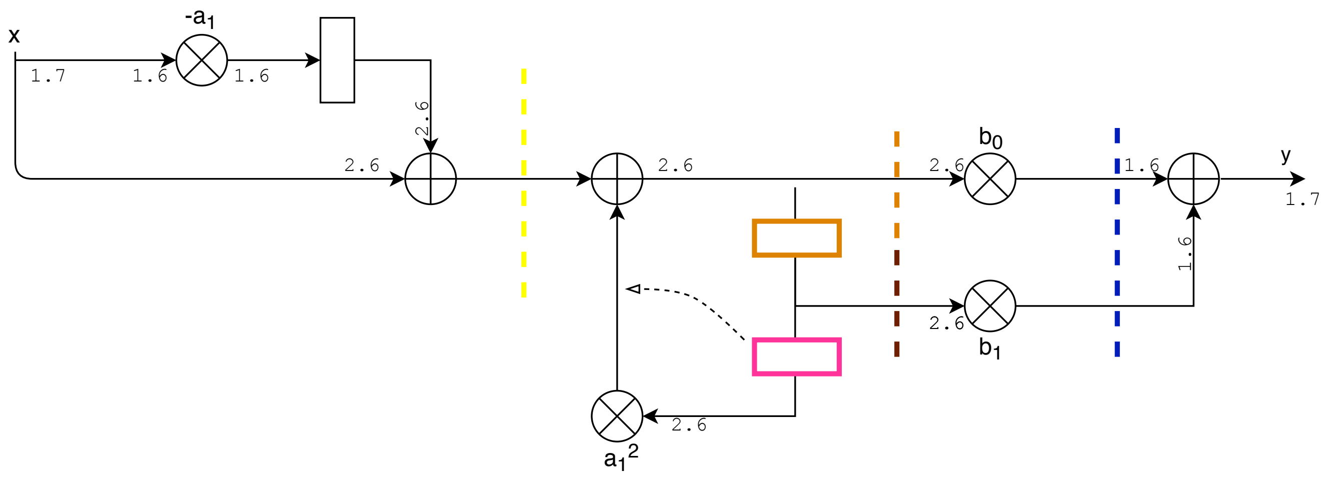

Figure 5.2: Initial DFG with annotated parallelism. Dashed vertical lines indicate cut-sets where pipeline stages will be inserted. The pink register can be moved to the position indicated by the arrow by means of retiming

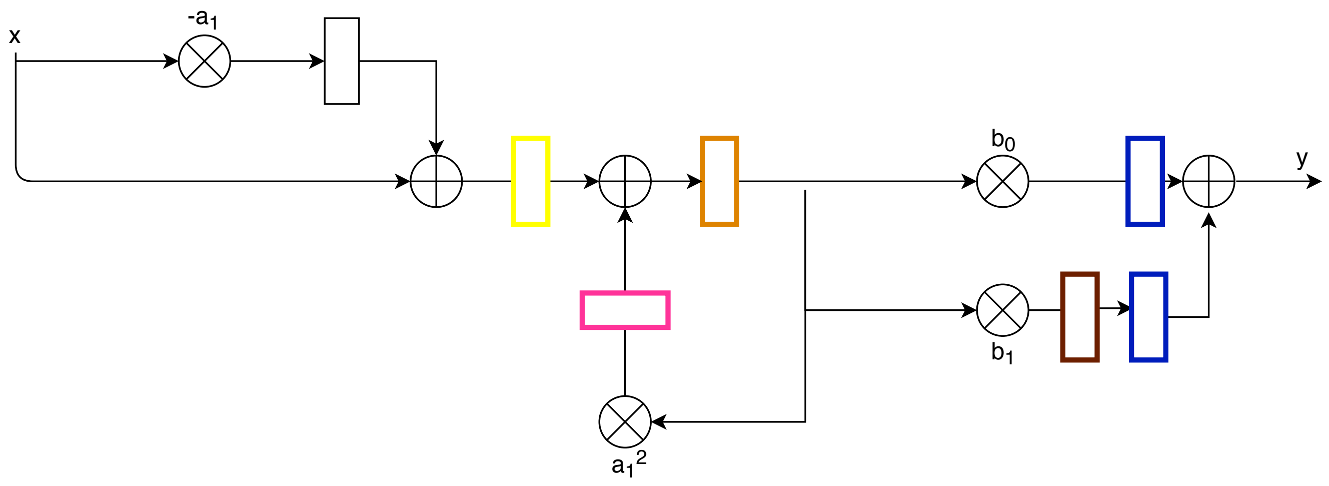

Figure 5.3: Final DFG. Colors identify registers corresponding to the DFG in figure Figure 5.2

5.1.3 Parallelism

Similarly to the first architecture, the sequence $w[n]$ can take on values slightly greater than one in magnitude, for example when the input signal is constant and equal to an extremum of the representable range ($2^7 - 1$ or -2^7). In order to avoid overflow, two integer bits are allocated wherever the intermediate variable w is processed. Slight savings in area can be achieved by allocating only one integer bit for the final adder, since the output of the feedforward multipliers (b_0, b_1) and the output variable y are necessarily lower than one in magnitude. The first multiplier ($-a_1$) can also operate on a resized version of x with only 6 fractional bits (as determined in the previous section) and 1 integer bit. In figure page 19 the final parallelism is annotated on every signal. All operators are assumed to operate with the same bit width for both the input operands and for the output. A different labeling on opposite sides of the same line implies that a resizing occurs in between and the same numerical value is represented in different formats by means of sign extension when more integer bits are added, truncation when the number of fractional bit is reduced, removal of the leftmost bits when the value is expected to be small enough to allow the allocation of less integer bits without running into overflow.

Point	Incr	Path
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
comp_dp/w1_reg[0]/CK	0.00	0.00
comp_dp/w1_reg[0]/Q	0.10	0.10
comp_dp/comp_m2/a[0]	0.00	0.10
comp_dp/comp_m2/mult_31/a[0]	0.00	0.10
...		
comp_dp/m2out_del_reg[6]/D	0.01	0.92
data arrival time		0.92
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
clock uncertainty	-0.07	-0.07
comp_dp/m2out_del_reg[6]/CK	0.00	-0.07
library setup time	-0.03	-0.10
data required time		-0.10
data required time		-0.10
data arrival time		-0.92
slack (VIOLATED)		-1.02

Table 5.2: Excerpt of the timing report with constraint $T_{ck} = 0$ (yellow stage included)

Point	Incr	Path
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
comp_dp/w1_reg[3]/CK	0.00	0.00
comp_dp/w1_reg[3]/Q	0.20	0.20
comp_dp/comp_m3/a[3]	0.00	0.20
comp_dp/comp_m3/mult_31/a[3]	0.00	0.20
comp_dp/comp_m3/mult_31/U170/ZN	0.08	0.28
comp_dp/comp_m3/mult_31/U169/ZN	0.09	0.36
comp_dp/comp_m3/mult_31/U257/ZN	0.13	0.49
comp_dp/comp_m3/mult_31/U255/ZN	0.10	0.59
...		
comp_dp/a3a_reg[6]/D	0.01	1.68
data arrival time		1.68
clock CLK (rise edge)	4.08	4.08
clock network delay (ideal)	0.00	4.08
clock uncertainty	-0.07	4.01
comp_dp/a3a_reg[6]/CK	0.00	4.01
library setup time	-0.03	3.98
data required time		3.98
data required time		3.98
data arrival time		-1.68
slack (MET)		2.30

Table 5.3: Excerpt of the timing report with constraint $T_{ck} = 4T_{min}$ (yellow stage included)

5.2 Control Unit

The addition of pipeline stages increases dramatically the number of possible states in our FSM, making it impractical to describe the control unit by enumerating all of them. We resorted to a structural description consisting of a delay line where a number of flip-flops are connected in cascade. The delay line contains two flip flops plus one additional flip flop for each pipeline stage, which amounts to a total of five delay units. One additional register samples the reset signal and controls `clear_w_regs`, a signal that causes the unit to enter the reset state for a single clock cycle, where both the datapath registers and the delay line are cleared. The latch enable signal is found at the output of the first flip flop, whereas `en_regs` corresponds to the `Q` output of the second flip-flop, which controls the registers in the feedback path (brown and pink) enabling them when a new data is available in the yellow pipeline register. Finally, VOUT corresponds to the end of the delay line.

5.3 Synthesis

The final design has been synthesized. The maximum frequency analysis is reported in Table 5.2. In a second run, the clock period has been set to $4T_{min} = 4.8\,\text{ns}$, with the data reported in Table 5.3.

5.3.1 Post-synth simulation

Synopsys can produce a Verilog description of the netlist generated by the compiler. This allows to verify the correctness of the outcome using ModelSim. Moreover, by running a second simulation after synthesis, reliable data regarding the switching activity associated to every internal node can be obtained and exported to enable an accurate power consumption estimation using the `report_power` command provided by Synopsys. The results regarding power consumption are reported in Table 5.4, where the simulation has been performed with a continuous stream of data (`VIN` always equal to one).

Power group	**Internal Power**	**Switching Power**	**Leakage Power**	**Total Power**
io_pad	0.00	0.00	0.0000	0.00 (0.00%)
memory	0.00	0.00	0.0000	0.00 (0.00%)
black_box	0.00	0.00	0.0000	0.00 (0.00%)
clock_network	0.00	0.00	0.0000	0.00 (0.00%)
register	162.68	2.45	8.11e+03	173.14 (66.48%)
sequential	0.37	0.516	323.25	1.20 (0.46%)
combinational	34.04	29.11	2.24e+04	86.09 (33.06%)
Total	197.08 uW	31.98 uW	3.1387e+04 nW	260.45 uW

Table 5.4: Power report with constraint $T_{ck} = 4T_{min}$ (yellow stage included)

Gates	1962
Cells	709
Area	1565.7 μm^2

Table 5.5: Gate count summary

5.3.2 Place and Route

This process was carried out by using an automated script derived while routing the standard architecture with the Innovus GUI. The commands issued by the interface could be found in a `.cmd` file. The results for this step are available in the directory named `innovus_fast` and can be fully replicated by running `source place_and_route.do` in Innovus.

Global routing The global routing phase issued a few warnings claiming that a few pins do not have a physical counterpart and thus cannot be routed: they correspond to the MSB and LSB of the coefficients which are resized to match the internal representation. Synopsys had already detected that those interface pins are not used internally and optimized them out with a warning. Since this mismatch between interface and internal representation is wanted in the design and given that the reports resulting from this step are acceptable, these warnings can be safely ignored. Global routing phase returns the following information regarding the total length routed on each layer (upper layer with 0 length are omitted).

```
#Total wire length = 5618 um.
#Total half perimeter of net bounding box = 6170 um.
#Total wire length on LAYER metal1 = 6 um.
#Total wire length on LAYER metal2 = 2967 um.
#Total wire length on LAYER metal3 = 2556 um.
#Total wire length on LAYER metal4 = 88 um.
#Total wire length on LAYER metal5 = 0 um.
...
#Total number of vias = 3817
```

Timing analysis An inspection of the slacks reported in `IIRFilter_postRoute.slk` shows that they are all positive and the lowest one is 2.53 ns. The summary shows that there are no violations.

Connectivity verification This check confirms that there are no violations or warnings.

Gate count The summary of total area and gate/cell count is summarized in Table 5.5

Power analysis Results are reported in Table 5.6, where power unit is mW.

Group	**Internal Power**	**Switching Power**	**Leakage Power**	**Total Power**
Sequential	0.1723	0.003196	0.008441	0.1839 (64.39%)
Macro	0	0	0	0
IO	0	0	0	0
Combinational	0.04565	0.03347	0.02262	0.101 (35.61%)
Clock (Combinational)	0	0	0	0
Clock (Sequential)	0	0	0	0
Total	0.218	0.03667	0.03106	0.2857 (100%)
CLK	0.1719	0.002668	0.008118	0.1827 (63.96%)

Table 5.6: Power report after place & route: Group Power for Rail VDD and CLK

Chapter 6

Conclusion

The result of this development process is a small, fast and power efficient first order IIR filter. Both of its architectures were obtained after a series of theoretical studies and tweaks through simulations, which allowed to develop the best possible structures. As few operators as possible were instantiated to minimize the area and power consumption, and various optimization techniques such as pipelining, retiming and guarded evaluation were applied to further improve the performance.

The filter's basic architecture can work with a clock frequency equal to 360 MHz, consume a total power of 230 µW whilst occupying an area of 910 µm^2. Most importantly, it filters with an almost ideal profile, as seen in Figure 2.3. The internal parallelism makes it so that there are no overflows and the minimal parallelism is chosen so that THD is better than -30 dBc.

Its more advanced version, implemented with the J-Look Ahead technique, can bring the clock frequency to almost 1 GHz, with a consumption of 285 µW and an area of 1565 µm^2.

Further optimizations or adaptations to particular needs favouring higher speeds, smaller areas or power consumptions, can be implemented by changing the internal architectures of the most complex components (the adders and multipliers), or by changing the structure of the filter altogether.