Master’s degree in computer engineering

Academic Year 2022/2023

UNIVERSITÀ DI PISA

Giovanni Marrucci

Title:

Cic Filter Interpolator

1 Introduction

The aim of this report is to build and simulate a CIC (Cascade Integrator-Comb) Interpolator. The system will be built with the VHDL programming language and then the behavior will be tested through the simulator ModelSim. After the simulation the filter will be analyzed through the Xilinx Vivado tool to see the maximum frequency (critical path), the elements used and the estimated power consumption.

Before starting all the above we need to understand what a CIC (Cascade Integrator-Comb) is.

The CIC (Cascade Integrator-Comb) is an optimized class of finite impulse response (FIR) digital filter combined with an interpolator or decimator, which was proposed by E. Hogenauer in 1981 as a cheaper alternative to the already existing filters from a decimation and interpolation point of view. An alternative and improved design was proposed in 2006, by R.A. Losada and R. Lyons, reducing the number of adders and delay elements and eliminating the integrators from CIC decimation filters.

2 A general look at the CIC (Cascade Integrator-Comb) Filter

As data converters become faster and faster, the application of narrow-band extraction from wideband sources, and narrow-band construction of wideband signals is becoming more important. These functions require two basic signal processing procedures: decimation (reduction of the sampling frequency) and interpolation (calculation of an approximate value based on values that are already known). In 1981 an efficient way of performing decimation and interpolation was introduced by E. Hogenauer whom devised a flexible, multiplier-free filter suitable for hardware implementation, that can also handle arbitrary and large rate changes.
This is known as CIC (Cascade Integrator-Comb), its structure is based on two sections, one made up by integrators and one made up by combs in cascade. This design gave to the filters some peculiar characteristics:

- Multipliers are not required.
- There are no coefficients to memorize.
- The structure is highly regular, in fact we can build a filter arranging 2 blocks.
- Control units and timings are quite simple to implement.
- The same filter can be used for different sample rates.
- Easy to design since parameters can be obtained from tables and the components can be designed using formulas.

2.1 Basic Components of a CIC filter

The basic components used to make a filter (both Decimator and interpolator) in the E. Hogenauer's architecture are:

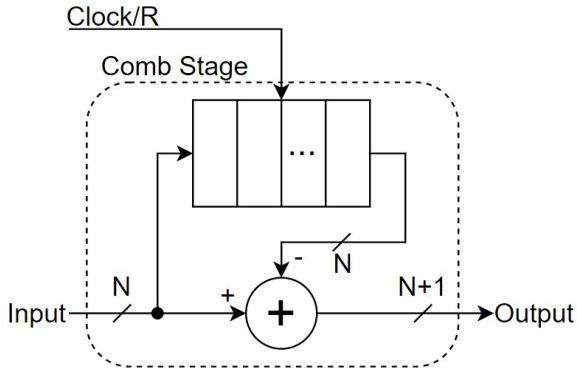
- Comb stage
- Zero Insertion
- Integrator Stage

While in the improved version of the interpolator proposed by R.A. Losada and R. Lyons, we use instead of the Zero Insertion a:

- Hold Insertion

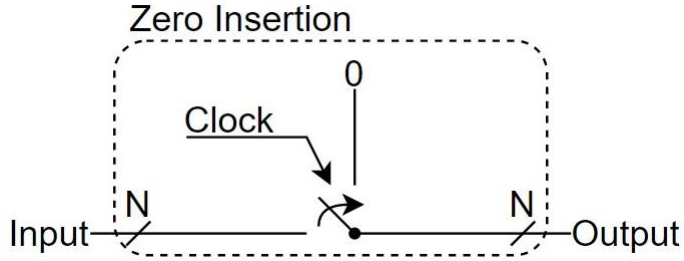
2.1.1 Comb Stage

It's the basic component of the comb section.
It takes a word on N bits, and it outputs a word on N+1 bits. In the Comb phase the data represented by the word on N bits as input is summed to the negated, delayed by a factor of M.



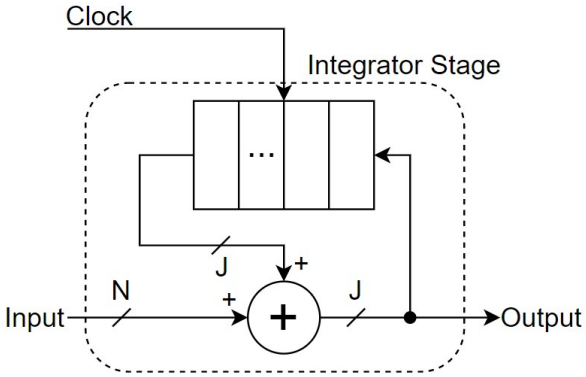
2.1.2 Zero Insertion

It's the component that divides the comb section and the integrator section. It is essentially a switch that causes a rate increase by a factor of R by inserting R - 1 zero valued samples between consecutive samples of the comb section output.



2.1.3 Integrator Stage

It's the basic component of the integrator section.
It takes a word on N bits, and it outputs a word on J bits, where J depends on the parameters of the filter. In the Integrator phase the data represented by the word on N bits as input is summed to the preceding sample. So, the output is going to be the sum of the sample at time t plus the sample taken at t - 1.



2.1.4 Hold Insertion

It's like the Zero insertion and can be defines as a black box, whose job is to repeat each input sample R– 1 times and it retains the value for a certain number of clocks.

2.2 Architectures of a CIC filter

The CIC filter can be both used as an interpolator or decimator, and this depends on the architecture that is chosen.
We will describe the Interpolation configuration with the given parameters:

- Interpolation factor R = 4
- Delay of comb stages M = 1
- Number of filter stages N = 4
- R-1 Zeros inserted
- Input: 16 bits
- Output: 16 bits
- Fs/R frequency for the comb section
- Fs frequency for the zero insertion/hold insertion
- Fs frequency for the integrator section

Before speaking about the architecture, itself we need to make some considerations on the bit growth of blocks.
So, the growth G at the i^{th} stage is:

$$G_i = \begin{cases} 2^i & i = 1, 2, \cdots, N \\ \frac{2^{2N-i}(RM)^{i-N}}{R}, & i = N+1, \cdots, 2N \end{cases}$$

And the register width W at the i^{th} stage is:

$$W_i = \lceil B_{\text{in}} + \log_2 G_i \rceil$$

We have a particular case for M = 1:

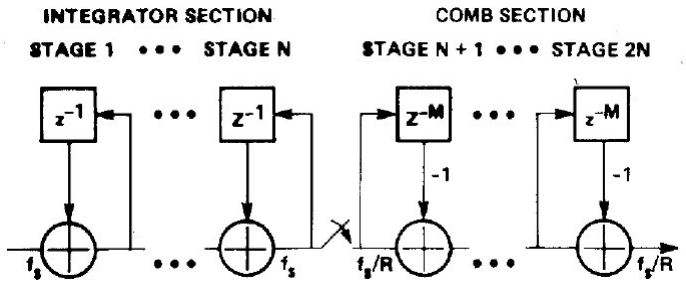
$$W_N = B_{\text{in}} + N - 1$$

These considerations are fundamental for the implementation of the filter. We will be discussing the consequences later.

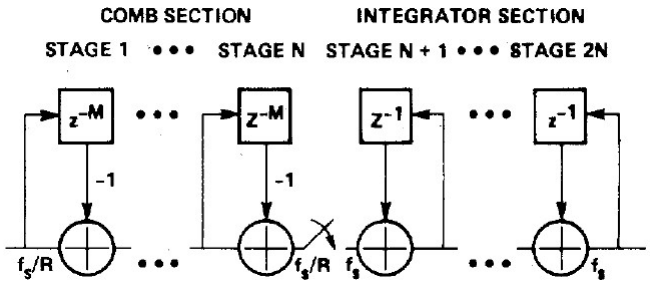
2.2.1 E. Hogenauer's architecture

As previously said, we have 2 possible configurations:

-Decimation configuration: N integrator stages -> Zero insertion -> N comb stages

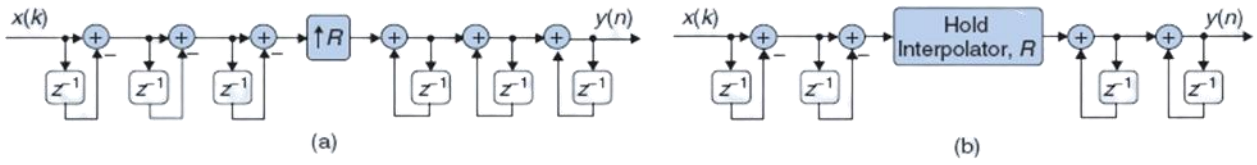


-Interpolation configuration: N comb stages -> Zero insertion -> N integrator stages



2.2.2 R.A. Losada and R. Lyons's architecture

This architecture was made to reduce the complexity of the filter CIC. In fact, as we can see below the architecture b has for both the comb section and the integrator section a stage less than the architecture a (E. Hogenauer's). Another difference, as said before, is the change from the zero insertion to the hold insertion.

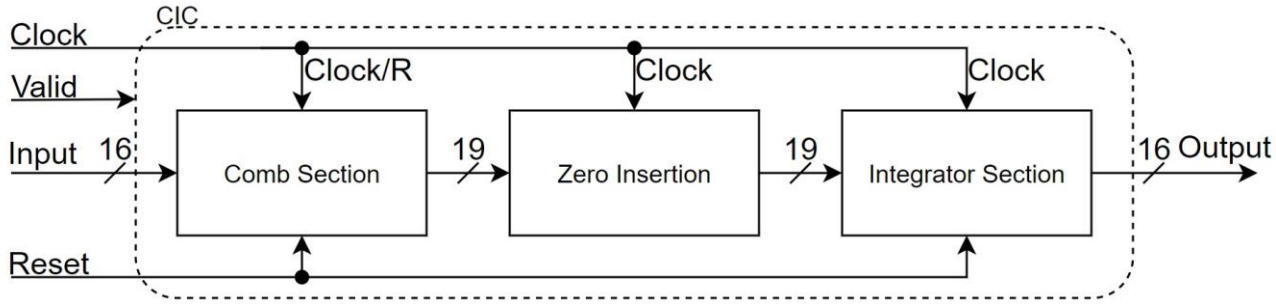


In both architectures we have a growth of bits from the input to the output, we need to either truncate or round the result to make the number of bits even. The truncation can either be done at the MSB (Most Significant Bit) or at the LSB (Least Significant Bit).
In the first case we eliminate the least significant bits, and we focus on the high values of the signal, interpolating them perfectly. We lose precision on low values near the zeros.
In the latter we do the opposite, therefore we will be interpolating perfectly the signal near the zeroes, and we will lose all the high values.

3 Implementation in VHDL

3.1 A general look and considerations

We will implement the interpolator following the E. Hogenauer's architecture.
The block diagram is the following:



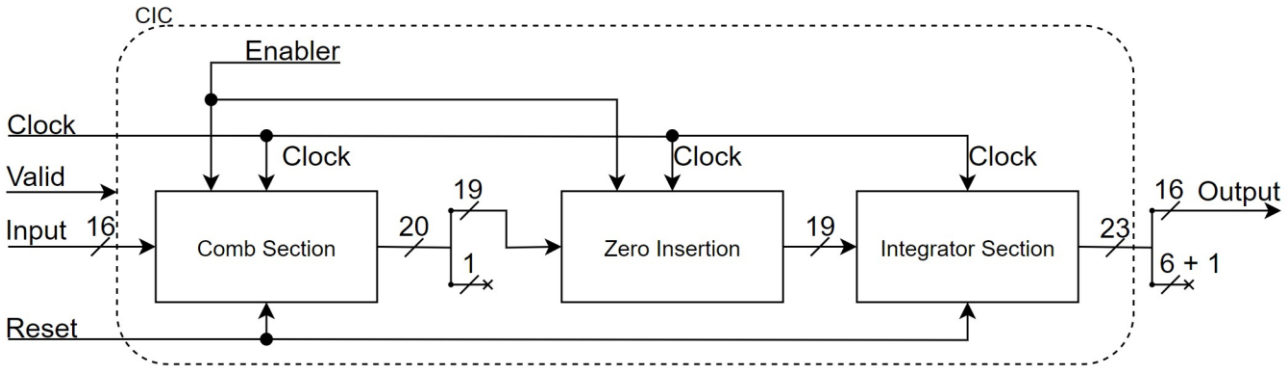
Before analyzing each block, we need to clarify the implementation of:

1. The clock
2. The bit's growth
3. Truncation of the output

1)In the model we require two different clocks, one for the comb section and one for the zero insertion and the integrator section. The easy path would be to create 2 different clocks, but after doing some research and reading some questions on the Xilinx forum, it would seem more efficient to pilot all the blocks with the same clock and implement a new enable signal for the comb section. This also facilitates the implementation of the zero-insertion component. Instead of using the clock for its temporization we can let through the sample when the enable signal is high, while when it's low, we insert a zero. In this case we will have 3 (R -1) zeros for each input.
2)Following the E. Hogenauer's rule we find that each last stage has a Bout = Bin + N – 1.
To put this in practice we see that for the comb section: 16 + 4 - 1 = 19 bits, while for the integrator section: 19 + 4 - 1 = 22 bits. The subtraction of 1 bit is simply a truncation and it can be achieved truncating either the MSB or the LSB.
3)This operation can be done on the 6 MSB or the 6 LSB and in both cases from a VHDL point of view it is sufficient to write:

- MSB selection: output <= out_integr_sec(Nbit + 6 down to 6)
- LSB selection: output <= out_integr_sec(Nbit - 1 down to 0)

Ater these clarifications we a have this block diagram:



Now we can start with the VHDL analysis.

3.2 CIC VHDL code

3.2.1 CIC entity

The entity is characterized by four input signals and one output signal:

```
entity cic is

    generic(
        Nstages : integer := 4; -- N = 4
        Rfactor : integer := 4; -- R = 4
        Nbit    : integer := 16 -- Input and Output bits
    );
    port (
        clk    : in   std_logic;
        valid  : in   std_logic;
        input  : in   std_logic_vector(Nbit - 1 downto 0); -- 16 bit
        res    : in   std_logic;
        output : out  std_logic_vector(Nbit - 1 downto 0)  -- 16 bit
    );

end cic;
```

3.2.2 CIC components

The fundamental components are the comb section, the zero insertion and the integrator section:

```
-- Components declaration
component comb_section is
    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        clk     : in   std_logic;
        enabler : in   std_logic;
        input   : in   std_logic_vector(Nbit - 1 downto 0);                -- 16 bit
        res     : in   std_logic;
        output  : out  std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0) -- 19 bit
    );
end component comb_section;

component zero_insertion is
    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        enabler : in  std_logic;
        input   : in  std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0); -- 19 bit
        output  : out std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0)  -- 19 bit
    );
end component zero_insertion;

component integrator_section is
    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        clk     : in   std_logic;
        enabler : in   std_logic;
        input   : in   std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0);    -- 19 bit
        res     : in   std_logic;
        output  : out  std_logic_vector(Nbit + (2 * Nstages - 2) - 1 downto 0) -- 22 bit
    );
end component integrator_section;
```

3.2.3 CIC Signals

We have five signals, one for each component output plus two to control the circuit:

```
-- Signals declaration
signal out_comb_section       : std_logic_vector (Nbit + (Nstages - 1) - 1 downto 0);     -- 19 bit
signal out_zero_insertion     : std_logic_vector (Nbit + (Nstages - 1) - 1 downto 0);     -- 19 bit
signal out_integrator_section : std_logic_vector (Nbit + (2 * Nstages - 2) - 1 downto 0); -- 22 bit
signal enabler                : std_logic;
signal counter                : integer;
```

3.2.4 CIC components map and process

Each component is mapped, and we have a basic process. Important to notice that we truncate the 6 LSB:

```
begin

    -- Components Map and Processes
    comb_section_map: comb_section
        generic map(
            Nstages => Nstages,
            Rfactor => Rfactor,
            Nbit    => Nbit
        )
        port map(
            clk     => clk,
            enabler => enabler,
            input   => input,
            res     => res,
            output  => out_comb_section
        );

    zero_insertion_map: zero_insertion
        generic map(
            Nstages => Nstages,
            Rfactor => Rfactor,
            Nbit    => Nbit
            )
        port map(
            enabler => enabler,
            input   => out_comb_section,
            output  => out_zero_insertion
        );

    integrator_section_map: integrator_section
        generic map(
            Nstages => Nstages,
            Rfactor => Rfactor,
            Nbit    => Nbit
            )
        port map(
            clk     => clk,
            enabler => enabler,
            input   => out_zero_insertion,
            res     => res,
            output  => out_integrator_section
        );

    cic_process: process(clk, valid, res, enabler, counter)
    begin

        if (valid = '1') then -- The filter remains in steady state if valid = 0
            if (rising_edge(clk)) then -- When we have the rising edge
                if (res = '1') then -- We have a complete reset of the filter
                    enabler <= '0';
                    counter <= 0;
                elsif (counter = 3) then
                    enabler <= '1';
                    counter <= 0;
                else
                    enabler <= '0';
                    counter <= counter + 1;
                end if;
            end if;
        end if;

    end process cic_process;

    output <= out_integrator_section(Nbit + (2 * Nstages - 2) - 1 downto 6); -- MSB 16 bit
```

3.3 Comb section VHDL code

3.3.1 Comb section entity

The entity is characterized by four inputs and one output signal:

```
entity comb_section is

    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        clk     : in   std_logic;
        enabler : in   std_logic;
        input   : in   std_logic_vector(Nbit - 1 downto 0);                -- 16 bit
        res     : in   std_logic;
        output  : out  std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0) -- 19 bit
    );

end comb_section;
```

3.3.2 Comb section components

The components are simply four comb blocks in series:

```
 -- Components declaration
 component comb_block is
     generic(
         Nbit : integer
         );
     port (
         clk     : in   std_logic;
         enabler : in   std_logic;
         input   : in   std_logic_vector(Nbit - 1 downto 0); -- Always a bit less than the output -> if 16 bit in -> 17 bit out
         res     : in   std_logic;
         output  : out  std_logic_vector(Nbit downto 0)    -- Always a bit more than the input -> if 16 bit in -> 17 bit out
     );
 end component comb_block;
```

3.3.3 Comb section signals

Since we have 4 components, we will have 4 outputs:

```
 --Singals declaration
 -- Since we have N = 4 we will be having 4 outsignals, one from each comb_block
 signal comb_block_out_0 : std_logic_vector(Nbit downto 0);     -- 17 bit
 signal comb_block_out_1 : std_logic_vector(Nbit + 1 downto 0); -- 18 bit
 signal comb_block_out_2 : std_logic_vector(Nbit + 2 downto 0); -- 19 bit
 signal comb_block_out_3 : std_logic_vector(Nbit + 3 downto 0); -- 20 bit <-- This must be truncated at the exit of the Comb section
```

3.3.4 Comb section components map and process

We have N = 4 therefore four comb blocks:

```
begin

    -- Components Map
    comb_block_0_map: comb_block
        generic map(
            Nbit => Nbit
        )
        port map(
            clk     => clk ,
            enabler => enabler,
            input   => input,
            res     => res,
            output  => comb_block_out_0
        );

    comb_block_1_map: comb_block
        generic map(
            Nbit => Nbit + 1
            )
        port map(
            clk     => clk,
            enabler => enabler,
            input   => comb_block_out_0,
            res     => res,
            output  => comb_block_out_1
        );

comb_block_2_map: comb_block
    generic map(
        Nbit => Nbit + 2
        )
    port map(
        clk     => clk,
        enabler => enabler,
        input   => comb_block_out_1,
        res     => res,
        output  => comb_block_out_2
    );

comb_block_3_map: comb_block
    generic map(
        Nbit => Nbit + 3
    )
    port map(
        clk     => clk,
        enabler => enabler,
        input   => comb_block_out_2,
        res     => res,
        output  => comb_block_out_3
    );

output <= comb_block_out_3(Nbit + (3 - 1) downto 0); -- Truncation of the MSB -> 19 bit
```

3.4 Zero insertion VHDL code

3.4.1 Zero insertion entity

It is useless to insert a clock or reset input because the entity will be working only through the counter and the values that it inserts are either 0 or the input. Therefore, the entity will have two inputs and one output:

```
entity zero_insertion is

    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        enabler : in  std_logic;
        input   : in  std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0); -- 19 bit
        output  : out std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0)  -- 19 bit
    );

end zero_insertion;
```

3.4.2 Zero insertion process

```
output <= input when enabler = '1' else (others => '0');
```

3.5 Integrator section VHDL code

3.5.1 Integrator Section entity

The entity is characterized by four inputs and one output signal:

```
entity integrator_section is

    generic(
        Nstages : integer;
        Rfactor : integer;
        Nbit    : integer
        );
    port (
        clk     : in   std_logic;
        enabler : in   std_logic;
        input   : in   std_logic_vector(Nbit + (Nstages - 1) - 1 downto 0);    -- 19 bit
        res     : in   std_logic;
        output  : out  std_logic_vector(Nbit + (2 * Nstages - 2) - 1 downto 0) -- 22 bit
    );

end integrator_section;
```

3.5.2 Integrator section components

The components are simply four integrator blocks in series:

```
component integrator_block is
    generic(
        Nbit : integer
    );
    port (
        clk    : in   std_logic;
        input  : in   std_logic_vector(Nbit - 1 downto 0); -- Always a bit less than the output -> if 19 bit in -> 20 bit out
        res    : in   std_logic;
        output : out  std_logic_vector(Nbit downto 0)      -- Always a bit more than the input -> if 19 bit in -> 20 bit out
    );
end component integrator_block;
```

3.5.3 Integrator section signals

Since we have 4 components, we will have 4 outputs:

```
signal integrator_block_out_0 : std_logic_vector(Nbit + 3 downto 0); -- 20 bit
signal integrator_block_out_1 : std_logic_vector(Nbit + 4 downto 0); -- 21 bit
signal integrator_block_out_2 : std_logic_vector(Nbit + 5 downto 0); -- 22 bit
signal integrator_block_out_3 : std_logic_vector(Nbit + 6 downto 0); -- 23 bit
```

3.5.4 Integrator section components map and process

We have N = 4 therefore four integrator blocks:

```
begin

    integrator_block_map_0: integrator_block
        generic map(
            Nbit => Nbit + 3
        )
        port map(
            clk => clk,
            input => input,
            res => res,
            output => integrator_block_out_0
        );

    integrator_block_map_1: integrator_block
        generic map(
            Nbit => Nbit + 4
        )
        port map(
            clk => clk,
            input => integrator_block_out_0,
            res => res,
            output => integrator_block_out_1
        );

    integrator_block_map_2: integrator_block
        generic map(
            Nbit => Nbit + 5
        )
        port map(
            clk => clk,
            input => integrator_block_out_1,
            res => res,
            output => integrator_block_out_2
        );

    integrator_block_map_3: integrator_block
        generic map(
            Nbit => Nbit + 6
        )
        port map(
            clk => clk,
            input => integrator_block_out_2,
            res => res,
            output => integrator_block_out_3
        );

output <= integrator_block_out_3(Nbit + (6 - 1)  downto 0); -- Truncation of the MSB -> 22 bit
```

3.6 Comb block VHDL code

3.6.1 Comb block entity

The entity is characterized by four inputs and one output signal:

```
entity comb_block is

    generic(
        Nbit : integer
        );
    port (
        clk     : in   std_logic;
        enabler : in   std_logic;
        input   : in   std_logic_vector(Nbit - 1 downto 0); -- Always a bit less than the output -> if 16 bit in -> 17 bit out
        res     : in   std_logic;
        output  : out  std_logic_vector(Nbit downto 0)    -- Always a bit more than the input -> if 16 bit in -> 17 bit out
    );

end comb_block;
```

3.6.2 Comb block components

The components should be a full adder and a flip flop, but I decided to synthetize it only with a flip flop since, we can use the '+' as a full adder (including the math library) and then control the output for the overflow:

```
-- Components declaration
component flipflop
    generic(
        Nbit : integer
    );
    port (
        clk         : in  std_logic;
        enabler     : in  std_logic;
        inputData   : in  std_logic_vector(Nbit - 1 downto 0);
        res         : in  std_logic;
        outputData  : out std_logic_vector(Nbit - 1 downto 0)
    );
end component flipflop;
```

3.6.3 Comb block signals

We have only one signal that is the output of the flip flop. This signal is fundamental since it used to calculate the real output of the block:

```
signal data_flow : std_logic_vector(Nbit - 1 downto 0);
```

3.6.4 Comb block components map and process

```
begin

    -- Components Map
    flipflop_map: flipflop
        generic map(
            Nbit => Nbit
        )
        port map(
            clk => clk,
            enabler => enabler,
            inputData => input,
            res => res,
            outputData => data_flow
        );

    -- We need to resize the data at the since we are doing a N - N (since M = 1 it is the previous sample) to get a N + 1 bit signal
    output <= std_logic_vector(resize(signed(input),Nbit + 1) - resize(signed(data_flow),Nbit + 1));
```

3.7 Integrator block VHDL code

3.7.1 Integrator block entity

The entity is characterized by three inputs and one output signal. We don't have the enabler input because each block operates every clock and not once every 4 clocks like the combs:

```
entity integrator_block is

    generic(
        Nbit : integer
    );
    port (
        clk    : in   std_logic;
        input  : in   std_logic_vector(Nbit - 1 downto 0); -- Always a bit less than the output -> if 19 bit in -> 20 bit out
        res    : in   std_logic;
        output : out  std_logic_vector(Nbit downto 0)      -- Always a bit more than the input -> if 19 bit in -> 20 bit out
    );

end integrator_block;
```

3.7.2 Integrator block components

The components should be a full adder and a flip flop, but as I previously mentioned, i decided to synthetize it only with a flip flop since, we can use the '+' as a full adder (including the math library) and then control the output for the overflow:

```
-- Components declaration
component flipflop
    generic(
        Nbit : integer
    );
    port (
        clk         : in  std_logic;
        enabler     : in  std_logic;
        inputData   : in  std_logic_vector(Nbit - 1 downto 0);
        res         : in  std_logic;
        outputData  : out std_logic_vector(Nbit - 1 downto 0)
    );
end component flipflop;
```

3.7.3 Integrator block signals

We have two signals instead of one because the save_data is used to connect the flip flop to the adder, while the data_flow is used to connect the output and as input to the flip flop:

```
signal data_flow : std_logic_vector(Nbit downto 0);
signal save_data : std_logic_vector(Nbit downto 0);
```

3.7.4 Comb block components map and process

```
begin

    -- Components Map
    flipflop_map: flipflop
        generic map(
            Nbit => Nbit + 1
        )
        port map(
            clk => clk,
            enabler => '1', -- Always active since working freq = clock
            inputData => data_flow,
            res => res,
            outputData => save_data
        );

    -- We need to resize the data at the since we are doing a N + J (since M = 1 it is the previous sample) to get a N + 1 bit signal
    data_flow <= std_logic_vector(resize(signed(input), Nbit + 1) + resize(signed(save_data), Nbit + 1));
    output <= data_flow;
```

3.8 Flip flop VHDL code

3.8.1 Flip flop entity

The entity is characterized by four inputs and one output signal:

```
entity flipflop is

    generic(
        Nbit : integer
    );
    port (
        clk         : in  std_logic;
        enabler     : in  std_logic;
        inputData   : in  std_logic_vector(Nbit - 1 downto 0);
        res         : in  std_logic;
        outputData  : out std_logic_vector(Nbit - 1 downto 0)
    );

end flipflop;
```

3.8.2 Flip flop process

```
registerProcess: process(clk, inputData, res) begin
    if (rising_edge(clk)) then
        if (res = '1') then
            outputData <= (others => '0');
        else
            if (enabler = '1') then
                outputData <= inputData;
            end if;
        end if;
    end if;
end process registerProcess;
```

4 Tests and simulations

To be sure about the result of the filter we will be doing a bottom- up simulation. It consists in testing each subcomponent with a testbench. After the examinations, we will be testing the whole architecture with its own testbench.

4.1 Flip flop test

The flip flop has been tested with an array of incremental values. All the inputs have been correctly maintained for a clock and then output.

→ Signals

```
constant T_clk       : time      := 100 ns;
constant Nbit_tb     : integer   := 4;
signal index         : integer := 0;
signal clk_tb        : std_logic := '0';
signal enabler_tb    : std_logic := '1';
signal res_tb        : std_logic := '1';
signal inputData_tb  : std_logic_vector(Nbit_tb-1 downto 0) := "0000";
signal outputData_tb : std_logic_vector(Nbit_tb-1 downto 0);
signal end_sim       : std_logic := '1';
```

→ Input LUT

```
type lut_t is array(natural range <>) of integer;
constant lut : lut_t(0 to 9) := (0, 1, 2 ,4 , 8, 4, 2, 1, 0, 1);
```

→ Process

```
process(clk_tb, inputData_tb) begin
    if (rising_edge(clk_tb)) then
        if (res_tb = '1') then
            index <= 0;
            inputData_tb <= (others => '0');
        else
            if (index < 9) then
                inputData_tb <= std_logic_vector(to_unsigned(lut(index),inputData_tb'length));
                index <= index + 1;
            else
                end_sim <= '0';
            end if;
        end if;
    end if;
end process;
```

4.2 Comb block test

The comb block has been tested with an array of 2 repeated values. These permitted us to test the block both with negative and positive numbers.

→ Signals

```
constant T_clk   : time      := 100 ns;
constant Nbit_tb : integer   := 4;
signal index     : integer   := 0;
signal clk_tb    : std_logic := '0';
signal res_tb    : std_logic := '1';
signal input_tb  : std_logic_vector(Nbit_tb - 1 downto 0) := "0000";
signal output_tb : std_logic_vector(Nbit_tb downto 0);
signal end_sim   : std_logic := '1';
```

→ Input LUT

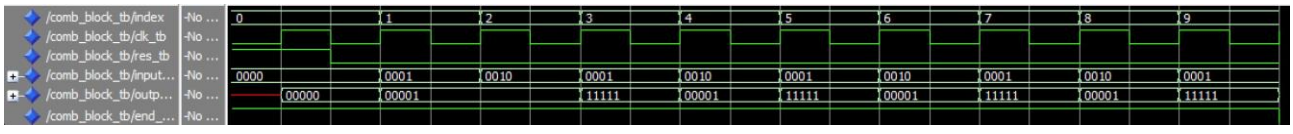
```
type lut_t is array(natural range <>) of integer;
constant lut : lut_t(0 to 9) := (1, 2, 1, 2, 1, 2, 1, 2, 1, 2);
```

→ Process

```
process(clk_tb, input_tb) begin
    if (rising_edge(clk_tb)) then
        if (res_tb = '1') then
            index <= 0;
            input_tb <= (others => '0');
        else
            if (index < 9) then
                input_tb <= std_logic_vector(to_signed(lut(index),input_tb'length));
                index <= index + 1;
            else
                end_sim <= '0';
            end if;
        end if;
    end if;
end process;
```

→ Graphic result

As we can see the output pattern will be characterized like this:
Index 1: 1 – 0 = 1 -> output: 00001; flip flop: 0001
Index 2: 2 – 1 = 1 -> output: 00001; flip flop: 0010
Index 3: 1 – 2 = -1 -> output: 11111; flip flop: 0001
Index 4: 2 – 1 = 1 -> output: 00001; flip flop: 0010
The comb block works as we expected saving the old input and subtracting it to the new one.



4.3 Integrator block test

The integrator block has been tested with an array of 2 repeated values to simulate a situation like the comb block, in fact we need to test the block both with negative and positive numbers.

→ Signals

```
constant T_clk    : time      := 100 ns;
constant Nbit_tb  : integer   := 4;
signal index      : integer   := 0;
signal clk_tb     : std_logic := '0';
signal res_tb     : std_logic := '1';
signal input_tb   : std_logic_vector(Nbit_tb - 1 downto 0) := "0000";
signal output_tb  : std_logic_vector(Nbit_tb downto 0);
signal end_sim    : std_logic := '1';
```

→ Input LUT

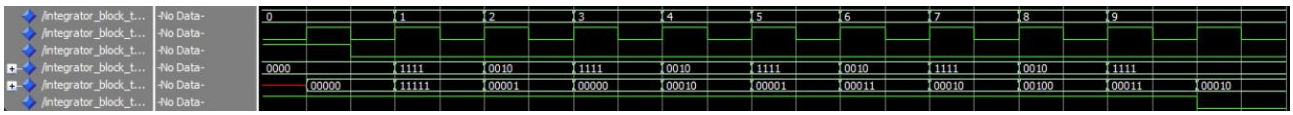
```
type lut_t is array(natural range <>) of integer;
constant lut : lut_t(0 to 9) := (-1, 2, -1, 2, -1, 2, -1, 2, -1, 2);
```

→ Process

```
process(clk_tb, input_tb) begin
    if (rising_edge(clk_tb)) then
        if (res_tb = '1') then
            index <= 0;
            input_tb <= (others => '0');
        else
            if (index < 9) then
                input_tb <= std_logic_vector(to_signed(lut(index),input_tb'length));
                index <= index + 1;
            else
                end_sim <= '0';
            end if;
        end if;
    end if;
end process;
```

→ Graphic result

As we can see the output pattern will be characterized like this:
Index 1: -1 + 0 = 1 -> output: 11111; flip flop: 11111
Index 2: 2 – 1 = 1 -> output: 00001; flip flop: 00001
Index 3: -1 + 1 = 0 -> output: 00000; flip flop: 00000
Index 4: 2 + 0 = 2 -> output: 00010; flip flop: 00010
The comb block works as we expected saving the old output and adding it to the new one.



4.4 Comb section and Integrator section test

To make the process faster I have used the same LUT table and the same process for both the architectures. Signals are different because of the expressions used in the whole architecture.

→ Comb section's signals

```
constant T_clk      : time      := 100 ns;
constant Nbit_tb    : integer   := 4;
constant Rfactor_tb : integer   := 4;
constant Nstages_tb : integer   := 4;
signal index        : integer   := 0;
signal clk_tb       : std_logic := '0';
signal res_tb       : std_logic := '1';
signal input_tb     : std_logic_vector(Nbit_tb - 1 downto 0) := "0000";
signal output_tb    : std_logic_vector(Nbit_tb + (Nstages_tb - 1) - 1 downto 0);
signal end_sim      : std_logic := '1';
```

→ Integrator section's signals

```
constant T_clk      : time      := 100 ns;
constant Nbit_tb    : integer   := 4;
constant Rfactor_tb : integer   := 4;
constant Nstages_tb : integer   := 4;
signal index        : integer   := 0;
signal clk_tb       : std_logic := '0';
signal res_tb       : std_logic := '1';
signal input_tb     : std_logic_vector(Nbit_tb + (Nstages_tb - 1) - 1 downto 0) := "0000000";
signal output_tb    : std_logic_vector(Nbit_tb + (2 * Nstages_tb - 2) - 1 downto 0);
signal end_sim      : std_logic := '1';
```

→ Input LUT

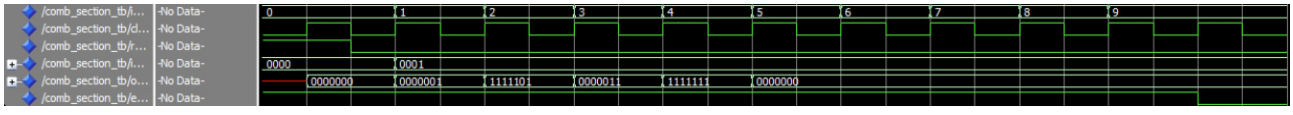
```
type lut_t is array(natural range <>) of integer;
constant lut : lut_t(0 to 9) := (1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
```

→ Process

```
process(clk_tb, input_tb) begin
    if (rising_edge(clk_tb)) then
        if (res_tb = '1') then
            index <= 0;
            input_tb <= (others => '0');
        else
            if (index < 9) then
                input_tb <= std_logic_vector(to_signed(lut(index),input_tb'length));
                index <= index + 1;
            else
                end_sim <= '0';
            end if;
        end if;
    end if;
```

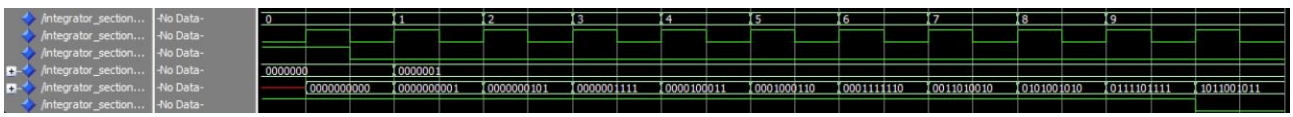
→ Comb section's graphic result

Checking the result after 4 clocks we see that a constant signal goes to 0 as expected.



→ Integrator section's graphic result

Checking the result after 4 clock we see that the output grows very fast as expected.



4.5 Zero insertion test

The zero insertion has been tested with a constant value in unput and with a switch on the enable signal every 3 clocks to test the zeros.

→ Signals

```
constant T_clk      : time      := 100 ns;
constant Nbit_tb    : integer   := 4;
constant Rfactor_tb : integer   := 4;
constant Nstages_tb : integer   := 4;
signal index        : integer   := 0;
signal clk_tb       : std_logic := '0';
signal enabler_tb   : std_logic := '0';
signal res_tb       : std_logic := '1';
signal input_tb     : std_logic_vector(Nbit_tb + (Nstages_tb - 1) - 1 downto 0) := "0000000";
signal output_tb    : std_logic_vector(Nbit_tb + (Nstages_tb - 1) - 1 downto 0);
```

→ Input LUT

```
type lut_t is array(natural range <>) of integer;
constant lut : lut_t(0 to 9) := (1, 1, 1, 1, 1, 1, 1, 1, 1, 1);
```

→ Process

```
process(clk_tb, input_tb) begin
    if (rising_edge(clk_tb)) then
        if (res_tb = '1') then
            index <= 0;
            input_tb <= (others => '0');
        else
            if (index < 9) then
                input_tb <= std_logic_vector(to_signed(lut(index),input_tb'length));
                index <= index + 1;
            else
                end_sim <= '0';
            end if;
        end if;

        if (index = 3 or index = 6) then
            enabler_tb <= '0';
        else
            enabler_tb <= '1';
        end if;
    end if;
end process;
```

→ Graphic result

As we can see the output pattern will be characterized like this:

4.6 CIC filter test

The filter has been tested with a 16-bit sinusoid signal generated with software (50 samples).

→ Signals

```
constant T_clk      : time      := 25 ns;
constant Nstages_tb : integer   := 4;
constant Rfactor_tb : integer   := 4;
constant N_tb       : integer   := 16;
signal clk_tb       : std_logic := '0';
signal valid_tb     : std_logic := '1';
signal input_tb     : std_logic_vector(N_tb - 1 downto 0) := "0000000000000000"; --Input
signal res_tb       : std_logic := '1';
signal output_tb    : std_logic_vector(N_tb - 1 downto 0); -- Output
signal end_sim      : std_logic := '1';  -- Signal to use to stop the simulation when there is nothing else to test
```

→ Input LUT

```
    type lut_t is array(natural range <>) of integer;
    constant lut : lut_t(0 to 49) := (
        0, 7927, 15383, 21925, 27165, 30790, 32587, 32448, 30381, 26509, 21062, 14364, 6812, -1143, -9031, -16383,
        -22761, -27787, -31163, -32687, -32269, -29934, -25820, -20173, -13327, -5689, 2285, 10125, 17363, 23570,
        28377, 31497, 32747, 32050, 29450, 25100, 19259, 12274, 4560, -3425, -11206, -18323, -24350, -28931, -31793,
        -32767, -31793, -28931, -24350, -18323 -- 4pi divide by 49 samples (16 bit in)
    );
```

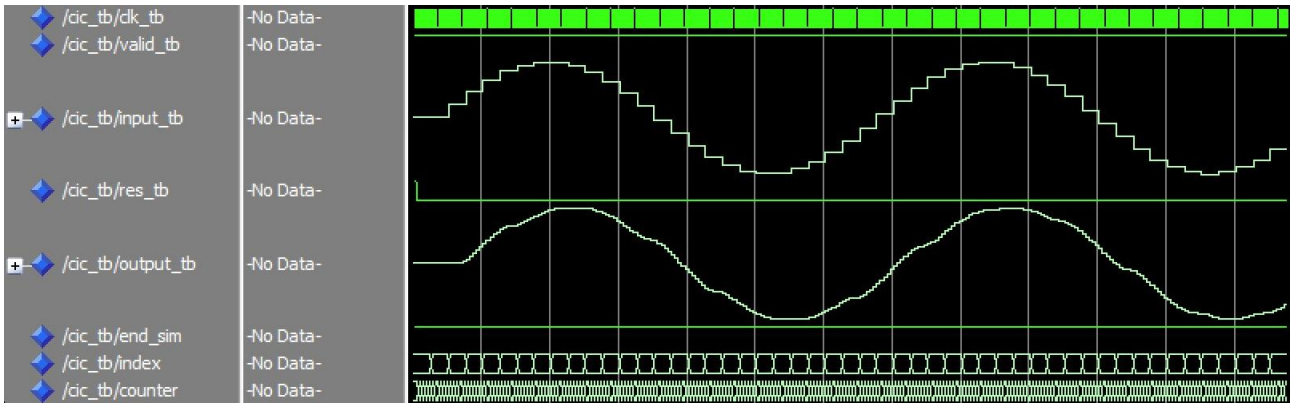
→ Process

```
process(clk_tb, input_tb) begin

    if (rising_edge(clk_tb)) then
        if (valid_tb = '1') then -- The filter remains in steady state if valid = 0
            if (res_tb = '1') then
                index <= 0;
                counter <= 0;
                input_tb <= (others => '0');
            else
                if(counter = 4) then
                    input_tb <= std_logic_vector(to_signed(lut(index),input_tb'length));
                    if(index < 94) then
                        index <= index + 1;
                    else
                        end_sim <= '0';
                    end if;
                    counter <= 0;
                else
                    counter <= counter + 1;
                end if;
            end if;
        end if;
    end if;
```

→ Graphic result

As we can see the output pattern will be characterized like this:



The output signal is exactly as we expected, in fact it is like the input but more defined and with a higher precision.

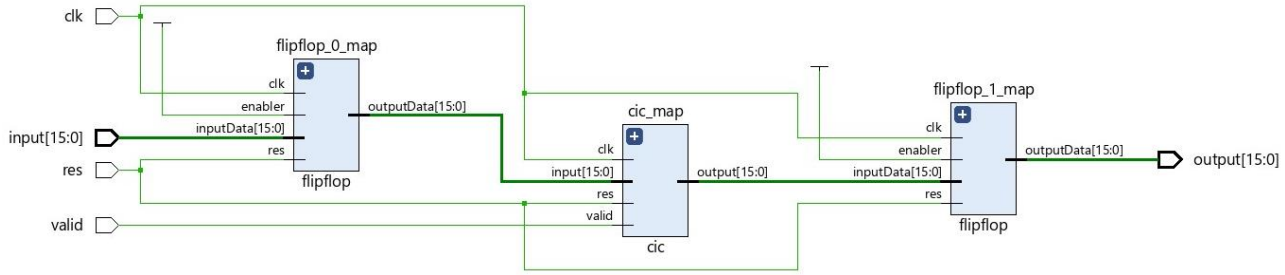
5 Logic synthesis

5.1 Constraints

The only important constraint is the clock frequency. Since filters are often used with analog audio signal it seems reasonable to use one for our implementation. I found out that “CD quality” audio resolution uses a 16-bit word for each sample and the sample rate is 44.1 kHz. So, I will be using this as a reference. Since our interpolator has an interpolation factor of four, we will be needing a clock four times higher than the input frequency signal: 44.1 * 4 = 176.4 kHz.

5.2 Synthesis

The synthesis phase has been done with the Xilinx Vivado 2020.2 software. I used the default parameters given by the application and only changed the “more options” field to set the mode to “out_of_context” as it was suggested during the lectures. The synthesis was carried out without any error or warning, since all the warnings found were caused by the mode set.
The Design:



5.2.1 Critical path

Using a clock set as previously mentioned we can see that the critical path (Worst Negative Slack) is widely above the 0. This result shows us that we can use the filter with higher frequencies signals.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5652.766 ns	Worst Hold Slack (WHS):	0.156 ns	Worst Pulse Width Slack (WPWS):	2834.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	258	Total Number of Endpoints:	258	Total Number of Endpoints:	204

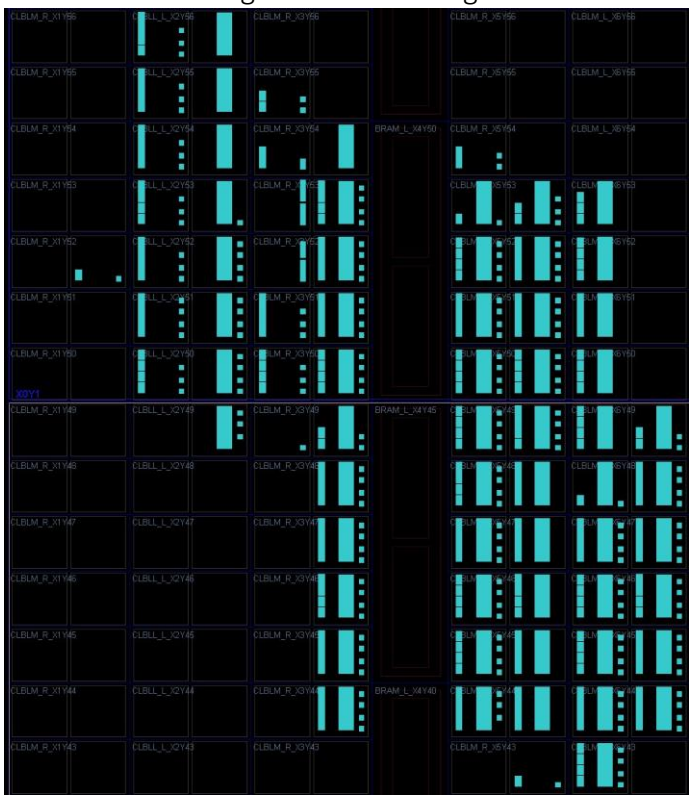
All user specified timing constraints are met.

5.2.3 Elements used

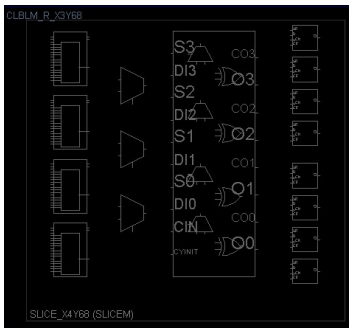
The hardware components used are the following:

Componet	FLOP_LATCH	LUT	CARRY
cic_map	172	205	55
comb_section_map	70	95	19
comb_block_0_map	16	16	4
comb_block_1_map	17	21	5
comb_block_2_map	18	19	5
comb_block_3_map	19	39	5
integrator_section_map	69	69	28
integrator_block_map_0	20	22	10
integrator_block_map_1	21	23	6
integrator_block_map_2	22	24	6
integrator_block_map_3	6		6
flip_flop_map	6		

While the fabric logic is the following:



And the structure of each CLBLM (Slice):



This suggests us that it is possible to implement this kind of architecture on smaller boards.

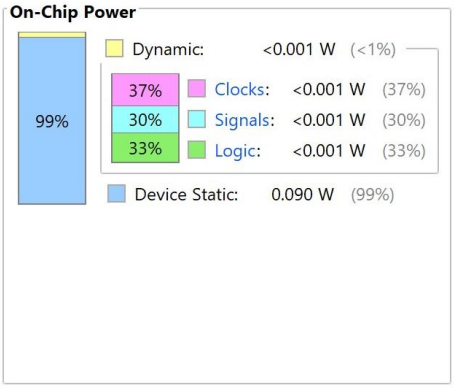
5.2.4 Power consumption

The summary report is the following:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.09 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.0°C**
Thermal Margin:	59.0°C (5.0 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



This divides the consumption under two categories: Dynamic and static power. We can see that most of the power is static and the small part that is dynamic is equally divided between clocks, signals and logic.

6 Conclusion

Cascaded Integrated Comb (CIC) filters are a great solution for operations of both decimation and interpolation. Their structure is made up of basic blocks (adders, registers) inserted in cascade. This makes them easy to build and fast.
Moreover, they can tolerate high frequencies. This was shown during the analysis done in Vivado in which we saw that the critical path was very far from being negative, letting us to use to use higher clocks.
For example, taking in consideration a more modern CD sample rate of 192kHz (we need a clock 4 times the sample rate: 768 kHz), we can see that the filter works fine:

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	1285.766 ns	Worst Hold Slack (WHS):	0.156 ns	Worst Pulse Width Slack (WPWS):	650.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	258	Total Number of Endpoints:	258	Total Number of Endpoints:	204

All user specified timing constraints are met.

Note: the bit depth of newer CDs is 24 bits therefore we should use a filter with greater input and output, but the analysis is done only to show the possibility to use higher clock frequencies.

All the factors mentioned above make the CIC filters a good invention and a reliable tool for interpolation or decimation applications.