UNIVERSITÀ DI PISA

Computer Engineering

Electronics and Communications Systems

Turbo coding interleaver

Project Documentation

Clarissa Polidori

Academic Year: 2020/2021

Table of Contents

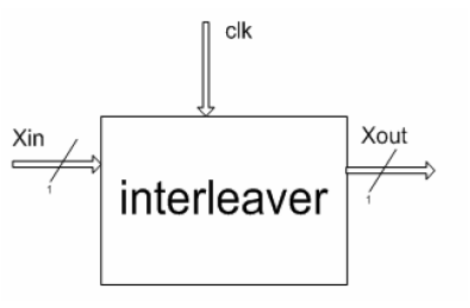
1 Introduction

1.1 Requirements

The objective is to design an RP (relative prime) interleaver for turbo codes compatible with the following specifications:

- Interleaver length = 1024.
- Implements the relation Xout(i) = Xin($|45 + i*3|_{1024}$)

Block diagram:



In the second chapter the concepts of channel coding and turbo codes will be presented in a simplified way, in order to introduce the device in its possible applications. It will be then shown the interleaver, in particular our model, its development in VHDL language and the tests performed.

1.2 Channel encoding

In a digital transmission the use of noisy channels, which introduce errors in data communication, requires information signal processing techniques to ensure data integrity for the best possible transmission of the message. The prerogative of representing information in digital format brings,compared to analogue transmission, the advantages of channel coding (data integrity) and source coding (information compression). The process of channel coding lends itself to encode the information sent in a communication channel in such a way that, in the presence of noise, errors can be detected and/or corrected. Two coding methods are distinguished:

- Backward Error Correction (BEC)
- Forward Error Correction (FEC)

The BEC requires only error detection, whereas in the FEC the decoder must also have the ability to correct a number of errors and to locate the location where they occurred. In the rest of the report we will consider the latter.

Among the types of codes for correction we briefly recall the *convolutional codes*. In convolutional codes each m-bit information symbol to be encoded is transformed into an n-bit symbol (with n > m), and m/n is the ratio (rate). The rate expresses the amount of redundancy in the code: the lower the rate, the more redundant the code.

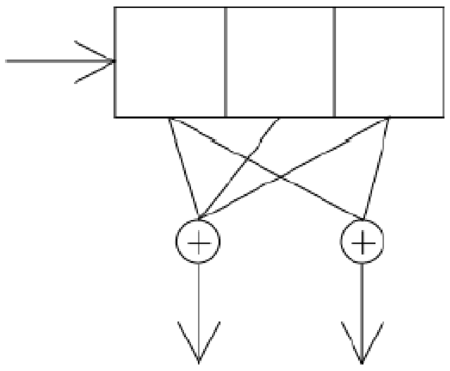


Figure 1.1: Figura 2.1: Convolutional encoder

The structure of a simple convolutional encoder is shown in Figure 2.1. The length k of the sliding register is called the constraint length, and the transformation is a function of the last k information symbols (in the figure k = 3). The encoder has n summers and typically the k memory registers are initialized to the value 0, have one input bit and two output bits, linear combinations of the current bit and some previous (two, in the figure). An input bit is entered into the leftmost register. The encoder gives the result and moves the bits of all register values to the right and waits for the next bit. If there are no input bits remaining, the encoder continues to provide results until all registers have returned to the zero state. The encoded bits are convolutions of the input sequence.

1.3 Turbo codes

Turbo codes are a class of high-performance error correction codes introduced in 1993.

They are a new class of convolutional codes that closely approximate the theoretical bounds imposed by Shannon's (second) theorem. The capacity Shannon capacity, also known as the Shannon limit, of a communication channel is the maximum data transfer rate that the channel can provide per a given level of signal-to-noise ratio, with an error rate as small as pleasure.
If an error probability per bit p_b is acceptable, then the following are achievable rates up to $R(p_b)$:

$$R(p_b) = \frac{C}{1 - H_2(p_b)}$$

$$H_2(p_b) = -[p_b \cdot log(p_b) + (1 - p_b) \cdot log(1 - p_b)]$$

Turbo codes have performance that approximates the theoretical Shannon results to less than 1 dB.

The basic idea is the concatenation of two codes: the information is encoded, all the obtained bits are permuted by an interleaver and again encoded by a second encoder (figure 1.2).

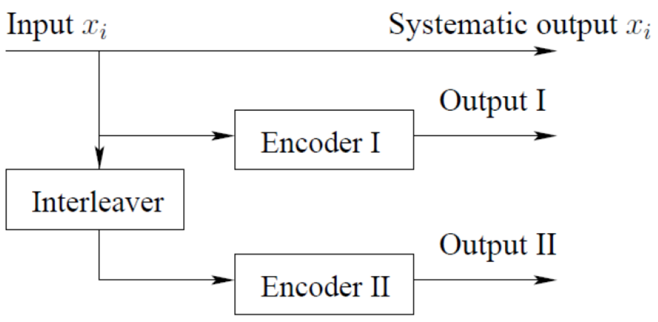


Figure 1.2: Figura 2.2: Generic turbo encoder

The novelty with respect to the traditional techniques of decoding of concatenated codes is the iterative decoding: in reception the decoders of the two component codes are activated alternatively, so as to use at each step the information produced by the other decoder at the previous step. This offers much less decoding complexity than previous

algorithms.
In practice, the same data is encoded with two disjoint encoders and the decoding is done using for each bit to be decoded the reliability of that bit provided by the other encoder. In other words, first is done a decoding with encoder 1; the verisimilitude is also stored of each bit. Then, you decode the same data with encoder 2, and use the previous verisimilitudes to aid in decision making.

Interleaving is the basis for turbo codes and the choice of interleaver, having a significant effect on the performance of the codes, is a crucial part in their their design.

1.4 Interleaving

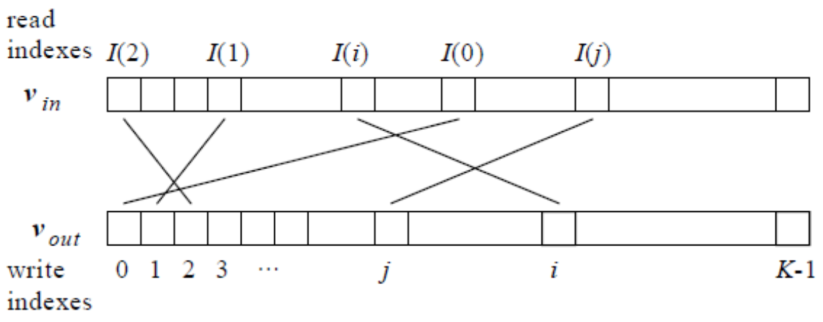
Interleaving is a digital signal processing technique used in digital broadcasting to arrange data in a manner used in digital broadcasting to arrange data in a non-contiguous manner to improve performance in order to improve performance in terms of detection and correction of errors in reception. correction of errors in reception. As shown above, it is a fundamental part fundamental part of turbo codes.

To show how the permutation of the data happens and the effectiveness of the method we use an example: we apply a very simple error correction code (repetition code) and carry out the interleaving process to a message, which will be sent over a noisy channel that causes packet errors, i.e. consecutive bits.

```
messaggio                      ==>  a  b  c  d  e  f

messaggio codificato           ==>  aaabbbcccdddeeefff

messaggio con interleaving     ==>  abcdefabcdefabcdef

messaggio ricevuto con errori  ==>  abcdefabcdXXXbcdef

messaggio dopo deinterleaving  ==>  aaXbbbcccdddXeefXf

messaggio decodificato         ==>  a  b  c  d  e  f
```

You can see that, despite the presence of errors (marked by "X"), the particularity of having permuted them allows us to reconstruct the message, which is not possible without the interleaving-deinterleaving technique.

The interleaver can be defined and implemented as shown in figure 1.3. The device reads from a vector of input bits v_{in} and writes to a v_{out} output vector the interleaved, i.e., permuted, bits.



The output data is written considering the indices i = 0...K-1, where K is the length of the interleaver. The vector I defines the order in which the input bits are read, so the i-th output bit, written to location i of the output vector, is read from location I(i) of the input vector.

A RP (relative prime) interleaver of length K is defined by

$$I(i) = |s + i \cdot p|_K, \qquad i = 0 \ldots K - 1$$

where p and K are relatively prime numbers (their greatest common divisor is 1) and s is the beginning index. The characteristic of being relatively prime ensures that each element is read only once.

Interleaving is primarily used in data communication, multimedia files, radio transmission multimedia files, in radio transmission (for example, over satellite or digital TV or ADSL). digital TV or ADSL). Interleaving is also used in optical media optical media (CD-ROMs, DVDs, etc.) to protect data from scratches and damage to the storage media. storage media.

1.5 Project structure

The project folder has the following structure:

```
turbo_coding_interleaver
├── create_testbench
├── design_logism
├── hdl
│   ├── src
│   └── tb
├── images
└── vivado_project_turbo_coding_interleaver
```

2 | Design

Starting from the specifications of an interleaver that must work with 1024 bits taking in input one bit at a time I decided to structure it in the following way. This is a simplified scheme built with logism, to give an idea of the general structure.

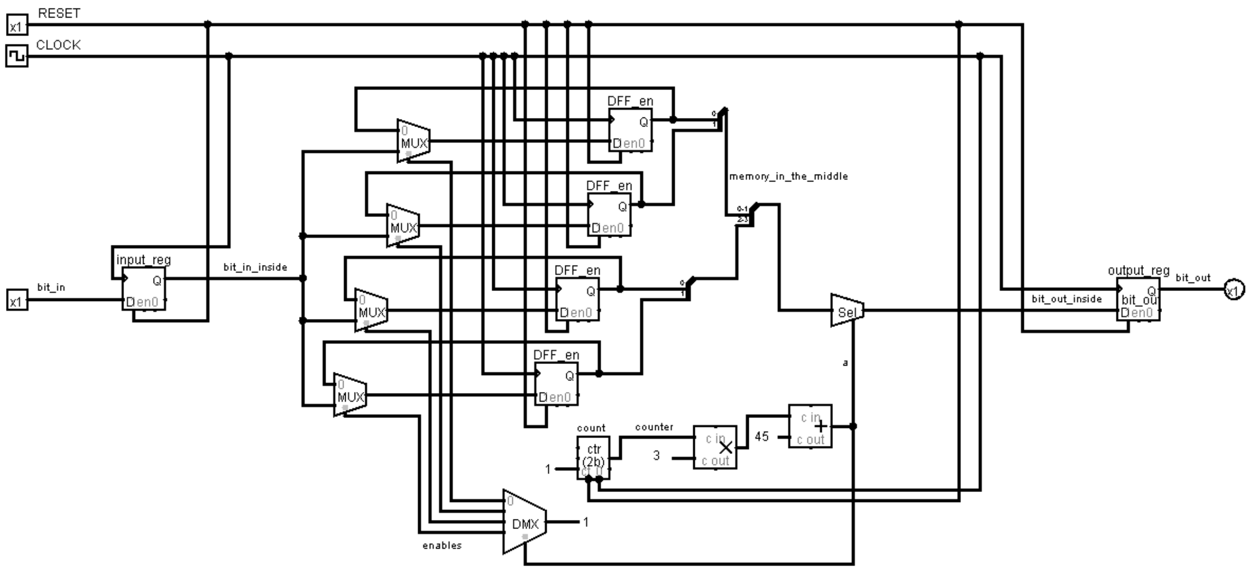


Figure 2.1: Schematic representation of the Interleaver architecture

The idea is to use 1024 registers (in this case 4 for simplicity) with enable, i.e. preceded by a multiplexer with which I attach to the input of the register or the input of the interleaver, or the output of the register itself then keeping the bit stored. I need a counter to keep track of the bits taken in, and then attach registers to the system input one at a time. The *enables* signal is used to enable the registers. The one described above is the input phase.
In the second phase, I use the counter to calculate the order in which to output bits, i.e. apply the interleaver formula. For this purpose I provide an adder and a multiplier. After computation, I can use this result represented by signal a to select the bits to be output one at a time. These bits are stored in registers throughout the output phase and are assigned to the *memory_in_the_middle* signal
Finally we see two additional registers, one for the input bit and one for the output bit, necessary to implement the interleaver on vivado, which can then correctly calculate all paths.

3 Implementation

3.1 Interleaver

The implementation of the interleaver is located within the *interleaver.vhd* file. It was built using 1024 registers with enable and a counter. The definition of the entity is show in Figure 3.1. It has been implemented using a generic dimension for more flexibility, even if in this project it was requested to work in 1024 bits values.

```
---------------------------- Entity declaration  ------------------------------
entity interleaver is
    generic( Nbit : POSITIVE := 1024);
        port(
                clock   : in std_logic;
               reset   : in  std_logic;
               bit_in  : in std_logic;           --  input bit
               bit_out : out std_logic          --  output bit
    );

end interleaver;
```

In figure 3.2 we can see the declaration of the components.

```
----------------------- Components declaration  -------------------------------------

    -- D-flip-flop with enable
    component DFF_en is
    port(
        resetn : in std_logic ;
        clock : in std_logic ;
        en : in std_logic;
        di : in std_logic;
        do : out std_logic
    );
    end component DFF_en;
    -- Counter
    component counter is
    generic( N : NATURAL := 8);
    port(
        clk       : in  std_logic;
        a_rst_h   : in  std_logic;
        increment : in  std_logic_vector(N - 1 downto 0);
            en        : in  std_logic;
        cntr_out  : out std_logic_vector(N - 1 downto 0)
    );
    end component counter;
```

For simplicity we do not report the components of the counter, which is the same that was implemented during the lessons composed of register and ripple carry adder.
Below instead we can see the internal signals that I used with the relative explanation.

```
----------------------- signals declaration  ---------------------------------------
-- Used to store input bits
signal memory_in_the_middle: std_logic_vector(Nbit-1 downto 0);
-- Signal connected to the counter output bits
signal counter_i: std_logic_vector(11 downto 0);
--Integer signal connected to counter output to do computations
signal counter_int: integer;
-- Index to access to the correct element of memory_in_the_middle
--to provide the correct output bit
signal a: integer;
-- Signal to enable one at time the registers to store input bits
signal enables: std_logic_vector(Nbit-1 downto 0);
-- Signal connected to counter enable
signal count_en: std_logic;
```

Below instead is the VHDL code of the interleaver behavior. For a better readability I don't report here the comments of the code, the complete comments can be found in the source file

```
proc: process(clock, reset)

    begin
```

```vhdl
    -- Initialize all signals
        if reset = '1' then

            a<=0;
            bit_out_inside<='Z';
            counter_int <= -1;
            count_en <= '1';
            for i in 0 to Nbit-1 loop
                enables(i) <='0';
            end loop;

        elsif rising_edge(clock) then

            counter_int <= ieee.NUMERIC_STD.TO_INTEGER(unsigned(counter_i));

            if counter_int < Nbit and counter_int > -1 then
                if(counter_int /= 0) then
                    enables(counter_int-1) <= '0';
                end if;

                enables(counter_int)<= '1';

            elsif counter_int <= Nbit*2 and counter_int >= Nbit then

                a <= ((45+(((counter_int) mod Nbit)*3)) mod Nbit);

                if counter_int = Nbit then
                    enables(counter_int -1) <= '0';
                    bit_out_inside <= 'Z';
                else
                    bit_out_inside <= memory_in_the_middle(a);
                end if;

                if counter_int = Nbit*2 then
                    count_en <= '0';
                end if;

                elsif counter_int  > Nbit*2 then
                    a<=0;
                    bit_out_inside<='Z';
                    count_en <= '1';
                end if;
            end if;

    end process proc;
```

4 | Testing

Regarding the testbench I thought that the best way to verify the correct operation of the system was to set all bits to 0 and only one at a time set it to 1 so as to go to check if it is positioned in the correct output bit. In the first testbench I decided to set to 1 bit 45, that according to the formula of our interleaver should represent bit 0 in output, and bit 42 that represents bit 1023 in output. This way I can check if the system behaves as we expect at the extremes. I also did the test with all bits at 0 and all bits at 1 and everything seems to work correctly. In conclusion, the only way we can verify that the system is shuffling according to the specifications is to set only one or two bits to 1 so we can easily evaluate which bit set to 1 has been moved to which position of the output, otherwise it is difficult to evaluate which bit has gone where.
Below is the testbech code in which only input bits 42 and 45 are set to 1:

```
  -- Librerie utilizzate

library IEEE;
use IEEE.std_logic_1164.all;

entity interleaver_tb is
end interleaver_tb;

-- Dichiarazione dell'entità

architecture interleaver_test of interleaver_tb is
        component interleaver
        generic (Nbit : POSITIVE := 1024);
                port (
                        clock            : in std_logic;
                        reset            : in std_logic;
                        bit_in            : in std_logic;
                        bit_out          : out std_logic
                        );
        end component;
```

```
-----------------------------------------------------

    --CONSTANT
        CONSTANT clock_period : TIME := 100 ns;
        CONSTANT len : INTEGER := 2051;

        --INPUT SIGNALS
        SIGNAL clock_tb : std_logic := '0';
        SIGNAL reset_tb : std_logic := '1';
        SIGNAL bit_in_tb : std_logic := '0';


        --OUTPUT SIGNALS
        SIGNAL bit_out_tb : std_logic;

        SIGNAL testing: Boolean :=True;
        SIGNAL count: INTEGER:= 0;
        SIGNAL count_reset: INTEGER:= 0;

        BEGIN
                I: interleaver
                generic map(Nbit => 1024)
                PORT MAP(
                        clock => clock_tb,
                        reset => reset_tb,
                        bit_in => bit_in_tb,
                        bit_out =>bit_out_tb
                        );

    --Generates clk
        clock_tb <=NOT clock_tb AFTER clock_period/2 WHEN testing ELSE '0';
    --reset_tb <= '0' after 10*clock_Period;

    --Runs simulation for len cycles
    proc_test: process(clock_tb, reset_tb)

    begin

        if(reset_tb = '1') then
                            bit_in_tb <= 'Z';
                count <= 0;
                        if rising_edge(clock_tb) and (count_reset < 3) then
                count_reset <= count_reset +1;
            elsif rising_edge(clock_tb) and (count_reset >= 3) then
                reset_tb <= '0';
            end if;
        elsif rising_edge(clock_tb) then
                        if count < 1024 and count /= 45 and count /= 42 then
                                bit_in_tb <= '0';

                        elsif count < 1024 and (count = 45 or count = 42) then
                                bit_in_tb <= '1';

                        elsif count >= 1024 then
                                bit_in_tb <= 'Z';
                        end if;
                        if(count > len) then
                                reset_tb <= '1';
                                testing <= false;          -- fine test
                        end if;
                        count <= count + 1;

                end if;

        end process proc_test;

end interleaver_test;
```

The behavior of the system with the testbench described above is shown below. I have reported the clock, the reset, the input bit, the input_inside bit, the output bit, the signal a (used for the output bits position) and the counter (used to distinguish between input or output phases). In the first image we see the initial setting of the signals before the reset signal goes to 0. In the second image we see bit 42 and bit 45 set to 1 while all others are 0. I chose these two bits because only with few bits set to 1 we can detect the system is working properly. Specifically I chose the bits that output after shuffling correspond to the output bit 0 and 1023 so as to analyze the operation at the extremes of the output phase. Applying the formula we get:

$$Xout(i) = Xin(|45 + i * 3|_{1024})$$

Replacing i with 45 and 42 which are the output bits we want at 1:

$$Xout(0) = Xin(|45|_{1024}) = Xin(45)$$

$$Xout(1023) = Xin(|45 + 1023 * 3|_{1024}) = Xin(|45 + 3069|_{1024}) = Xin(|3114|_{1024}) = Xin(42)$$

Initially there was only one clock cycle between the end of the input phase and the beginning of the output phase, now instead The reason why we lose 3 clock cycles is that I have inserted an input register and an output register to the interleaver for the calculation of the paths in vivado.

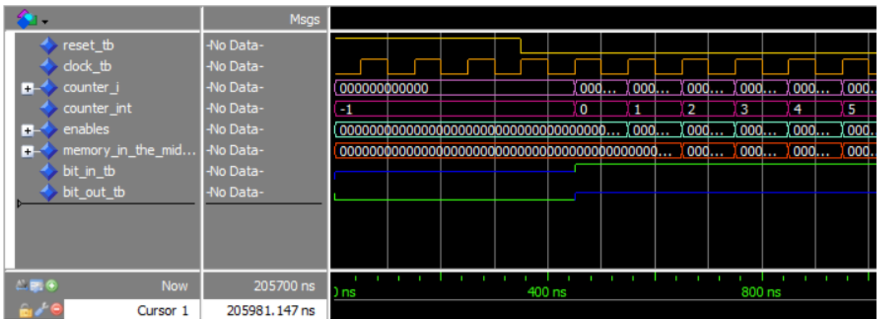


Figure 4.1: Initial signal setting

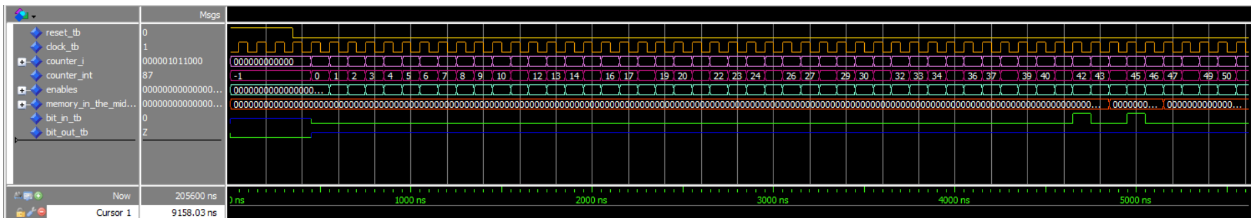


Figure 4.2: Input phase, 42nd and 45th bits set to 1

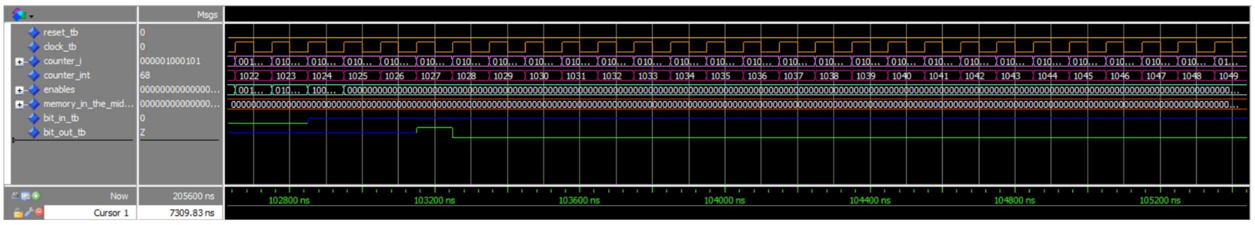


Figure 4.3: Output phase, 1st output bit is 1

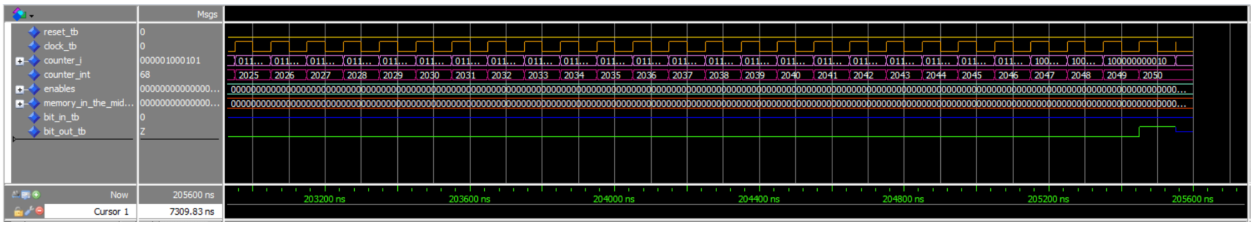


Figure 4.4: Output phase, 1023rd output bit is 1

For simplicity's sake I have not shown the testbench images with all 1024 input bits set to 1 or 0. The result is as expected, in the output in the two cases respectively all bits are at 1 and 0.
Moreover I decided to build a little program in python to build the testbench, that takes as input a text file in which can be inserted the inputs in binary or hexadecimal format and builds the testbench in vhd format.

5 | Synthesis and Implementation

Before proceeding with the synthesis, a wrapper for the interleaver is needed, since it has been designed using a generic dimension for more exibility: because of that, the *interleaver_1024.vhd* does nothing more than setting the dimension of the interleaver to be a 1024-bit interleaver. Then, the syntesis has been performed using the Vivado software by Xilinx and addressing the xc7z010clg400-1, which is the FPGA on the ZyBo board.

5.1 RTL Analysis

Before heading with the Synthesis a preliminary double-check of the correctness of the system has been made by simply opening the schema obtained by the Elaborated Design. No problem has been found at this stage.

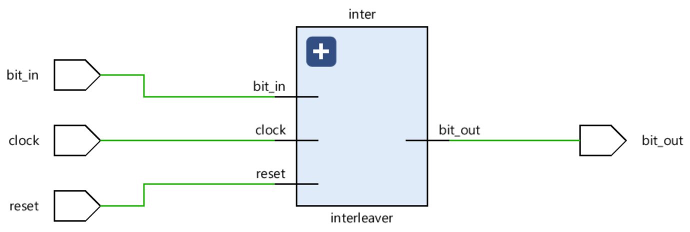
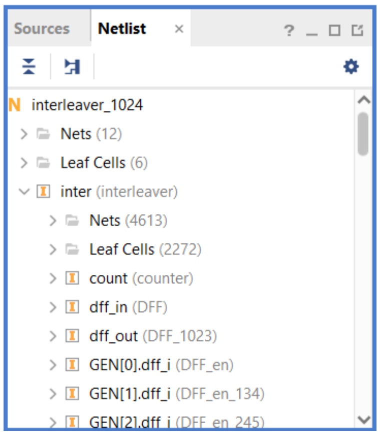


Figure 5.1: Interleaver Wrapper



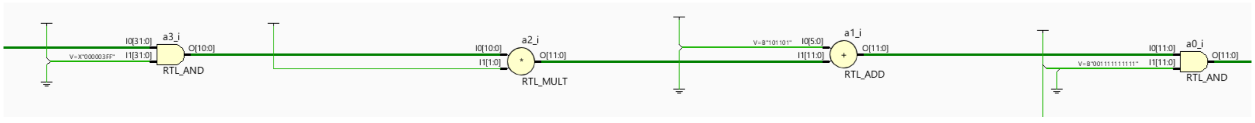
interleaver_1024
Nets (12)
Leaf Cells (6)
inter (interleaver)
Nets (4613)
Leaf Cells (2272)
count (counter)
dff_in (DFF)
dff_out (DFF_1023)
GEN[0].dff_i (DFF_en)
GEN[1].dff_i (DFF_en_134)
GEN[2].dff_i (DFF_en_245)

Figure 5.2: RTL Netlist

5.2 Synthesis

5.2.1 Vivado elaborated design

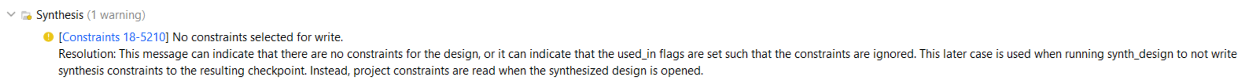
The design developed by vivado is too complicated to be able to report here an exhaustive image, because of the large amount of registers and signals present in the system, I will only report the implementation of the adder for the constant 45 and the multiplier for the constant 3.



5.2.2 Timing constraints

- Clock @ 125 MHz (period: 8 ns, rise at: 0 ns, fall at: 4 ns)

5.2.3 Synthesis warnings



Synthesis (1 warning)

[Constraints 18-5210] No constraints selected for write.
Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used_in flags are set such that the constraints are ignored. This later case is used when running synth_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.

This is a warning that comes up for every Vivado's project because of a bug.

5.2.4 Setup and hold slacks

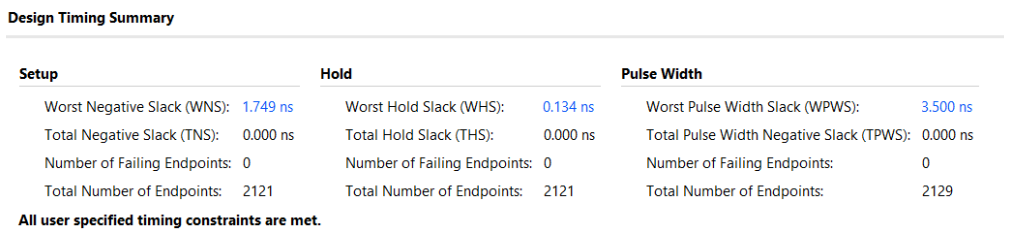
Since it was not asked to generate the bitstream for the board, the synthesis has been performed without the IO planning, but imposing just the clock constraint.

The interleaver works at 125 MHz, so the clock period has been chosen

$$t_{clk} = \frac{1}{1,25 * 10^8 MHz} = 0,8 * 10^{-8} s = 8ns$$

The Worst Negative Slack (WNS) is 1,749 ns. Given this information, the maximum theoretical frequency which the circuit could properly work at is about:

$$f_{max} = \frac{1}{(8 - 1,749) * 10^{-9}} = 159MHz$$



Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	1.749 ns	Worst Hold Slack (WHS):	0.134 ns	Worst Pulse Width Slack (WPWS):	3.500 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:	2121	Total Number of Endpoints:	2121	Total Number of Endpoints:	2129

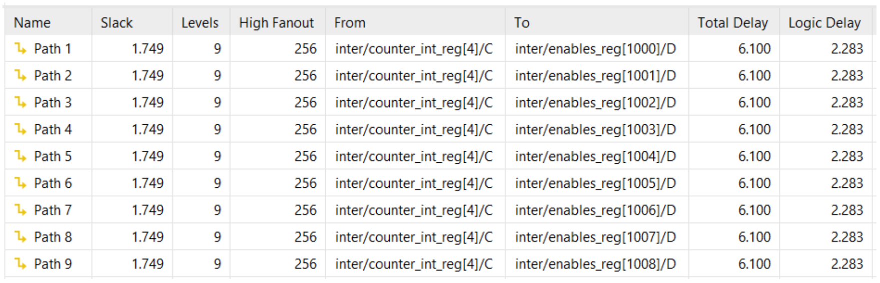
All user specified timing constraints are met.

Figure 5.3: Timing report

As it's clear from the timing summary of the synthesis, since all the time slacks are positive, there are no setup nor hold time violations.

5.2.5 Critical path

The WNS is determined by the Critical Path of the architecture, which is shown in the following picture:



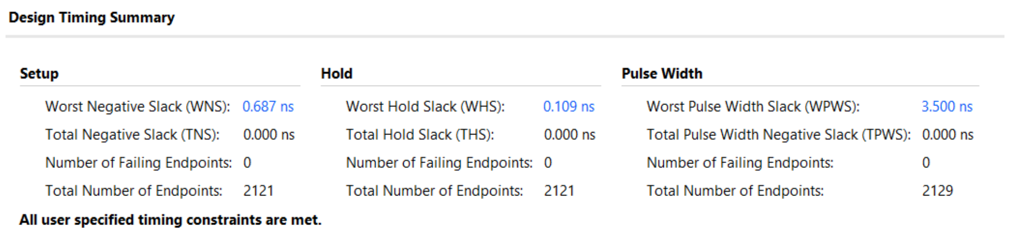
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay
Path 1	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1000]/D	6.100	2.283
Path 2	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1001]/D	6.100	2.283
Path 3	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1002]/D	6.100	2.283
Path 4	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1003]/D	6.100	2.283
Path 5	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1004]/D	6.100	2.283
Path 6	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1005]/D	6.100	2.283
Path 7	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1006]/D	6.100	2.283
Path 8	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1007]/D	6.100	2.283
Path 9	1.749	9	256	inter/counter_int_reg[4]/C	inter/enables_reg[1008]/D	6.100	2.283

Figure 5.4: Critical path report

5.3 Implementation

After the implementation, the Worst Negative Slack (WNS) is 0.687 ns and the maximum theoretical frequency which the circuit could properly work at is about:

$$f_{max} = \frac{1}{(8 - 0.687) * 10^{-9}} = 136MHz$$



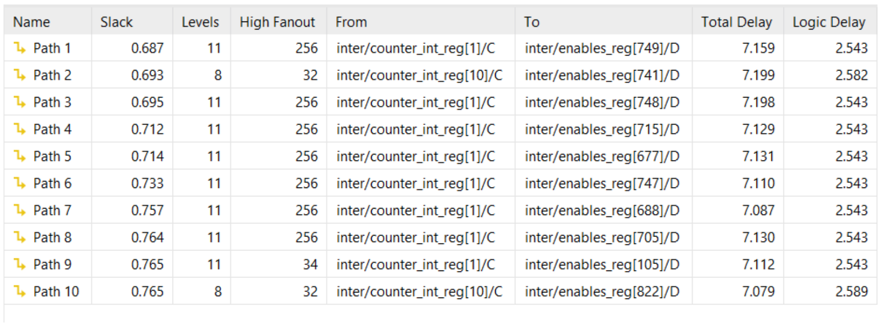
Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	0.687 ns	Worst Hold Slack (WHS):	0.109 ns	Worst Pulse Width Slack (WPWS):	3.500 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:	2121	Total Number of Endpoints:	2121	Total Number of Endpoints:	2129

All user specified timing constraints are met.

Figure 5.5: Timing report

These ones are the register-logic-register paths sorted by tsetup slack time; the first one is the critical one, and provides the global setup slack and the maximum clock frequency of the clock for this implementation:



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay
Path 1	0.687	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[749]/D	7.159	2.543
Path 2	0.693	8	32	inter/counter_int_reg[10]/C	inter/enables_reg[741]/D	7.199	2.582
Path 3	0.695	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[748]/D	7.198	2.543
Path 4	0.712	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[715]/D	7.129	2.543
Path 5	0.714	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[677]/D	7.131	2.543
Path 6	0.733	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[747]/D	7.110	2.543
Path 7	0.757	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[688]/D	7.087	2.543
Path 8	0.764	11	256	inter/counter_int_reg[1]/C	inter/enables_reg[705]/D	7.130	2.543
Path 9	0.765	11	34	inter/counter_int_reg[1]/C	inter/enables_reg[105]/D	7.112	2.543
Path 10	0.765	8	32	inter/counter_int_reg[10]/C	inter/enables_reg[822]/D	7.079	2.589

Figure 5.6: Critical path report

A view of the FPGA after the implementation is shown in Figure 13: as you can see from the light blue regions.

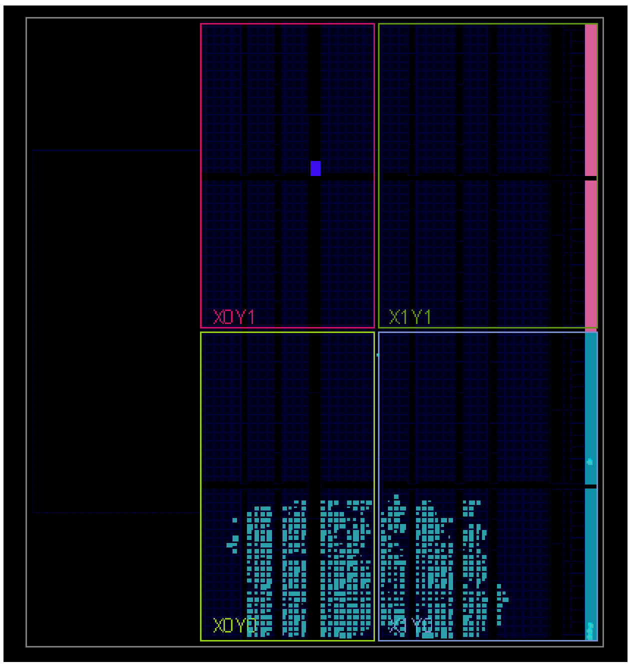
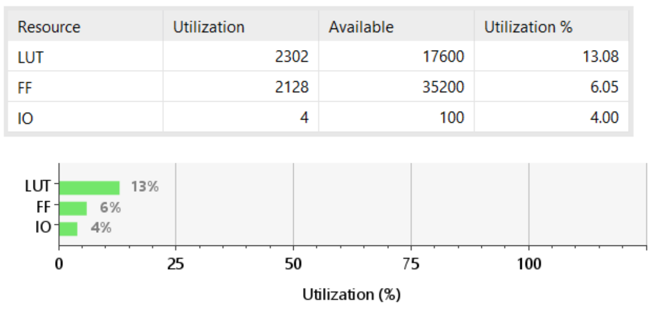


Figure 5.7: FPGA content after implementation

6 Conclusions

6.1 Utilization Report

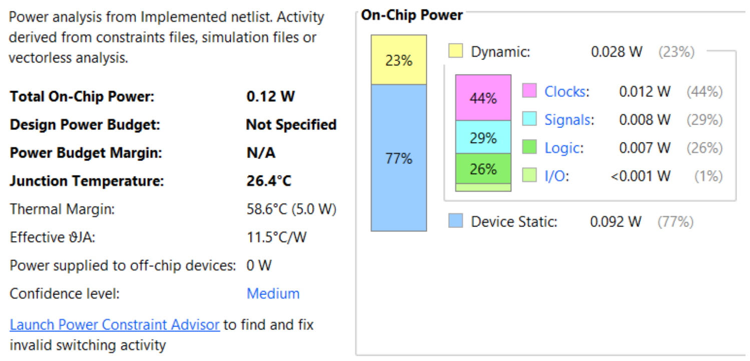


Resource	Utilization	Available	Utilization %
LUT	2302	17600	13.08
FF	2128	35200	6.05
IO	4	100	4.00

LUT
FF
IO
13%
6%
4%
0
25
50
75
100
Utilization (%)

Figure 6.1: Resource utilized by the syntesis

6.2 Power report



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

Total On-Chip Power:	**0.12 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.4°C**
Thermal Margin:	58.6°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

On-Chip Power

🟨	Dynamic:	0.028 W	(23%)
🟪	Clocks:	0.012 W	(44%)
🟦	Signals:	0.008 W	(29%)
🟩	Logic:	0.007 W	(26%)
🟩	I/O:	<0.001 W	(1%)
🟦	Device Static:	0.092 W	(77%)

Figure 6.2: Power Report Summary