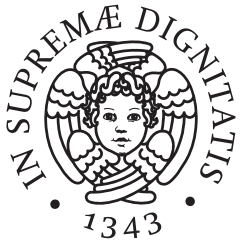


University of Pisa
MSc in Computer Engineering

Electronics and Communication Systems

Design and implementation of a CRC algorithm on a Xilinx Zynq Board

Supervisors
Prof. Luca Fanucci
Prof. Massimiliano Donati

Student
Marco Pinna

November 9, 2022

Contents

Specifications

In what follows, the design and implementation of a CRC algorithm on a Xilinx Zynq Board is carried out.
The specifications are the following (translated from Italian):

Implementation of a CRC algorithm

Design a digital circuit that implements the Cyclic Redundancy Check (CRC) algorithm, both for the *sender* and for the *receiver*, according to the specifications below. Such algorithm is a powerful control method that exploits the idea of redundancy; a sequence of F redundant bits (Frame Control Sequence FCS)is added (by the sender) to a data sequence M, so that the transmitted message, on M+F bit, is divisible by a predefined divisor called CRC polynomial. The receiver, through a division by the same polynomial used by the sender, can detect the correctness of the received data.

Message M=56 bits
Frame Control Sequence F=8 bits
CRC polynomial of degree 9: $x^8 + x^4 + x^3 + x^2 + 1$
(The binary representation of the polynomial will therefore be 100011101)
Transmitted message M+F=64 bit

Through the `md` signal one can set whether the circuit is to be used as sender or as receiver.
When operating as receiver the last F values of OUT indicate the correctness of the received message.

The final project report has to include:

- Introduction (description of the algorithm, possible applications, possible architectures, etc.)
- Description of the architecture selected for the realization (block diagram, inputs/outputs, etc.)
- VHDL code (with detailed comments)
- Test-plan and relative testbench for verification
- Results of the automated logic synthesis on Xilinx FPGA Zync platform: resources used (slice, LUT, etc.), maximum operating frequency, critical path, etc. commenting potential warning messages
- Conclusions

Chapter 1

Introduction

The work is organized as follows:

- this chapter contains the description of the algorithm, some of its applications and possible architectures for the implementation
- in chapter 2 an in-depth explanation of the chosen architectures is given
- in chapter 3 the VHDL code of each component is shown and commented
- chapter 4 concerns the test plan and the testbenches used for the verification of the system
- chapter 5 shows the results of the automated logic synthesis on the Xilinx Vivado Design Suite, with considerations concerning resource use, maximum clock frequency, etc.
- finally in chapter 6 conclusions are drawn and some consideration about possible optimizations or different implementations are made

The entire codebase can be found at `https://github.com/MPinna/CRC285/`.

1.1 Algorithm description

A Cyclic Redundancy Check (CRC) algorithm is a technique used in digital networks that uses redundant bits to detect accidental changes in digital data.
Let us supposed to have a *sender* (S) and a *receiver* (R) at the two end of a communication channel.
To each message M of `m` bits being sent by S, an additional section (the Frame Control Sequence, or FCS) of `f` redundant bits is added, whose value is computed performing a polynomial division between the message M (the dividend) and a polynomial G (the divisor) called *generator* and calculating the remainder of such division.

Upon reception, R performs the same division and, depending on the value of the remainder, it is able to check whether the message M has been corrupted during transmission.

More in detail:

- let M be an `m` bits long binary string.
  An `m-1` degree polynomial $M(x)$ is associated to it, such that the i-th coefficient of the polynomial is equal to the i-th bit of the string (e.g. 100101111 $\Rightarrow x^8+x^5+x^3+x^2+x+1$).

- Let $G(x)$ be the generator polynomial whose binary representation is `f + 1` bits long (the degree of G(x) will therefore be `f`).
  M is shifted to the left by `f` positions, padding to the right with `f` zeros. This corresponds to multiplying $M(x)$ by x^f.

- The FCS is built as follows:
  a polynomial *long division* between $x^f \cdot M(x)$ and $G(x)$ is performed, using finite field arithmetic on the Galois Field GF(2). Let $Q(x)$ and $R(X)$ be the quotient and the remainder of such division, respectively.
  It follows that

$$x^f \cdot M(x) = Q(x) \cdot G(x) + R(x) \tag{1.1}$$

  If we subtract $R(x)$ from both sides of the equation, we get

$$x^f \cdot M(x) - R(x) = Q(x) \cdot G(x) \tag{1.2}$$

- The polynomial on the left-hand side of the equation is divisible by $G(x)$ and contains the original message M in the `m` highest bits and the FCS in the `f` lower bits (by construction, the degree of $R(X)$ is strictly less than `f`, so it can be represented on `f` bits).

- This `m+f` bits long string M|FCS is what will be sent to the receiver.

- The receiver can check the integrity of the message by dividing M|FCS by $G(x)$ and checking whether the remainder is equal to 0: if it is, then the message M has likely not been corrupted during transmission, otherwise it has and thus needs to be discarded.

In this particular implementation, M will be 56 bits long, the generator will be

$$x^8 + x^4 + x^3 + x^2 + 1 \quad \leftrightarrow \quad 100011101 \tag{1.3}$$

therefore the FCS will be 8 bits long, for a total of 64 bits to be sent for each message.

Example
Let us use as an example the transmission of the word “hi” encoded in ASCII.

The binary string corresponding to "hi" is:

$$01101000\ 01101001 \tag{1.4}$$

We first pad it with zeros

$$01101000\ 01101001\ 00000000 \tag{1.5}$$

then we compute the FCS by performing the long division between 1.5 and 1.3.

$$\begin{array}{l}
01101000011010010000000 \\
\underline{100011101} \\
0101111001 \\
\ \underline{100011101} \\
\ 00110010001 \\
\ \ \underline{100011101} \\
\ \ 0100011000 \\
\ \ \underline{100011101} \\
\ \ 000000101010000 \\
\ \ \ \ \ \underline{100011101} \\
\ \ \ \ \ 00100110100 \\
\ \ \ \ \ \ \underline{100011101} \\
\ \ \ \ \ \ 000\mathbf{10100100}
\end{array}$$

The remainder R is 10100100. R is subtracted from (1.5) to create the FCS in the 8 lower bits, yielding the following string

$$01101000\ 01101001\ 10100100. \tag{1.6}$$

1.2 Possible applications

CRC algorithms are used extensively in various data transmission technologies and protocols: SATA, USB, Ethernet, CDMA, Bluetooth, etc.
They are also found in several standards, programs and file formats such as MPEG-2, PNG, Gzip, to name a few.
There are different implementations of CRC algorithms, each with its own polynomial. The choice of the polynomial (both its degree and its coefficients) depend on several factors such as the maximum desired overhead and the sensitivity to different errors.

An important remark to be made is that such algorithms protect against *accidental* corruption of data, but **they are not suited against *intentional* alteration of data**, since a valid CRC to attach to the tampered message can easily be computed once the algorithm is known.

1.3 Possible architectures

Two different architectures were identified to implement the algorithm:

- the first one simply implements the long division shown in 1.1 using shift registers, accumulators, counters , XOR gates, etc.
  In this implementation the algorithm advances bit by bit, therefore in order to "consume" the whole message (56 bits), a total of at least 56 clock cycles is needed.

- the second one exploits the facts that the divisor (i.e. the generator) is fixed and that in GF(2) there is no carry, therefore adjacent bytes have no influence on each other during the algorithm. It is then possible to pre-compute the division for each possible byte and store the results in a LUT.
  At each step, the LUT will then be used to perform a bitwise XOR between its corresponding entry and the next byte of the message.
  This allows to perform the division byte by byte, thus speeding up the whole process. This solution will be more efficient in terms of speed, at the expense of a probable greater use of resources on the FPGA since a 256x1B LUT has to be created.

Chapter 2

Architecture

As mentioned in 1.3, two architectures were identified to implement the algorithm: the first one carries out the procedure in a **bitwise** fashion; the second one works **bytewise** by means of a look-up table (LUT).
In this chapter both the architectures will be analysed in detail, showing advantages and drawbacks of the two.
The initial stage of the development process followed a top-down approach to identify the individual components that would comprise the entire architecture. For each component, it was studied which simpler sub-components it should consist of.
The whole development process of the components was carried out with modularity in mind, so as to ease the reuse of them in the different architectures.

2.1 Bitwise architecture

Figure 2.1 shows a high-level view of the bitwise architecture of the CRC (clocks and resets are not shown for the sake of clarity).
The architecture is divided in **datapath** (black connections) and **control unit** (red connections). The datapath consists of all the register, and processing or combinatorial units required to execute each step of the algorithm. The control unit takes care of issuing, at certain clock cycles, specific control signals to the datapath, to ensure the correct execution of each step of the algorithm.

Light grey rectangles are **registers** made of D Flip-Flops;
dark grey rectangles are a different kind of registers whose basic unit is a *Double Data Flip-Flop (D2FF)*, whose architecture will be discussed later; this allows to build a register with two different inputs (e.g. the accumulator) or a Parallel-Input-Parallel-Output (*PIPO*) Shift Register;
the light blue block is a purely combinatorial block which implements a XOR between the input and the CRC generator polynomial G (the most significant bit of the input behaves as a sort of *enable*, i.e. when it is `0` the combinatorial block will act as if it was

transparent).

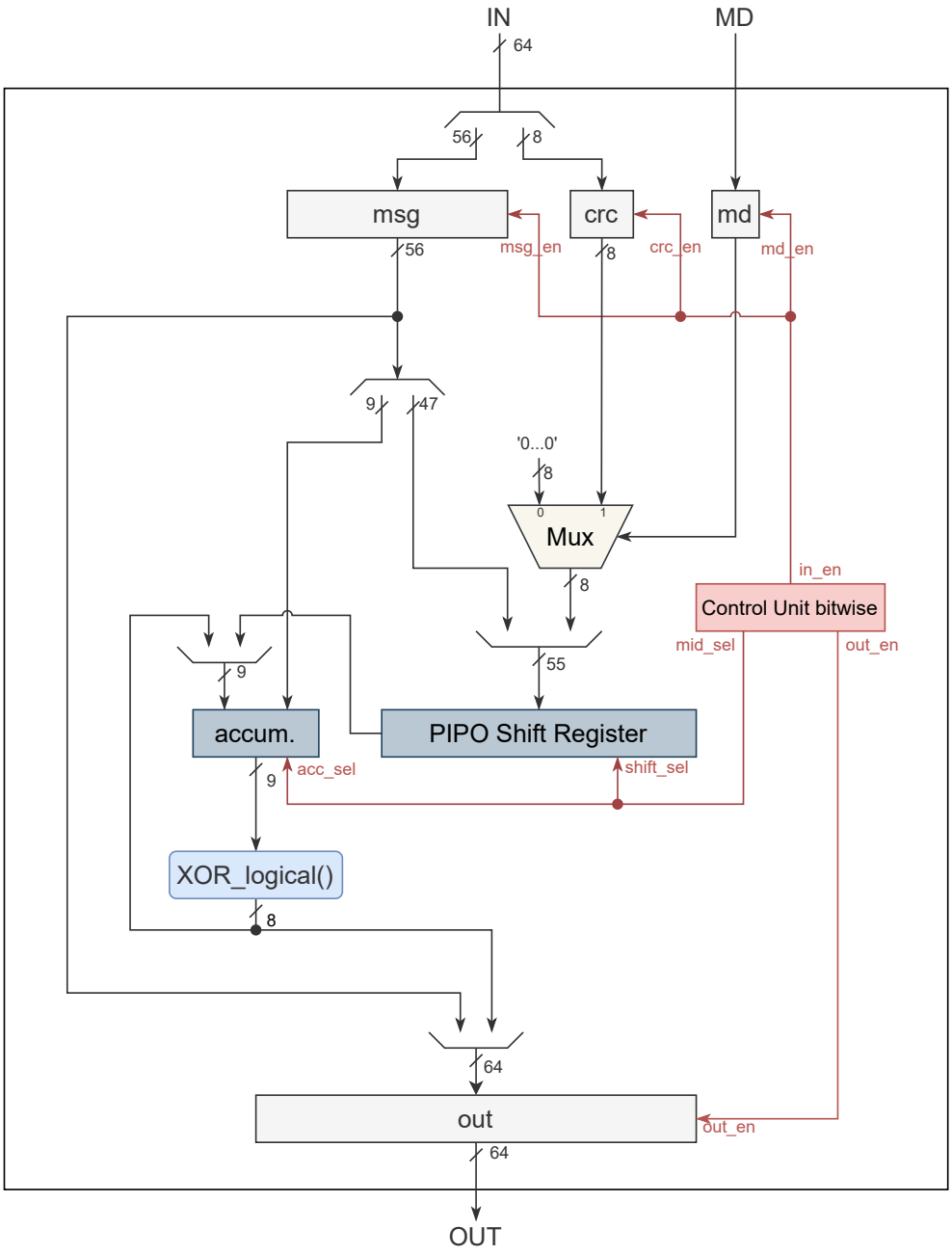


Figure 2.1: Block diagram of the bitwise implementation of the CRC.

The bitwise CRC works as follows:

- Input data is read from `IN` and `MD` and stored into the input registers `msg`, `crc` and `md`
- Input data is then fed into the processing section of the circuit, which computes the polynomial long division advancing bit by bit.
- At the end of the computation, the original message along with the CRC (or the CRC check, depending on the input value of `MD`) is placed into the output register `out`.

The following pseudocode shows what happens during each clock cycle, in a finite-state machine fashion:

```
@posedge(clock 0):
    # input
    msg[55:0] <- IN[63:8]
    crc[7:0] <- IN[7:0]
    md <- MD

    ControlUnit.in_en = 0;
    ControlUnit.mid_sel = 0;
    ControlUnit.out_en = 0;

@posedge(clock 1):
    # shift register
    PIPOShiftRegister[54:8] <- msg[46:0]
    PIPOShiftRegister[7:0] <- md == 0 ? '00000000' : crc[7:0]
    # accumulator
    accumulator[8:0] <- msg[55:47]

    ControlUnit.mid_sel = 1;

@posedge(clock 2..55):
    # shift register
    PIPOShiftRegister[0] <- '0'
    for i in 1..54:
        PIPOShiftRegister[i] <- [i-1]

    # accumulator
    accumulator[0] <- PIPOShiftRegister[55]
    accumulator[8:1] <- accumulator[7:0] XOR (GENERATOR AND accumulator
    [8])

@posedge(clock 56):
    # shift register
    for i in range(1, 55):
        PIPOShiftRegister[i] <- [i-1]
    PIPOShiftRegister[0] <- '0'
    # accumulator
    accumulator[0] <- PIPOShiftRegister[55]
    accumulator[8:1] <- accumulator[7:0] XOR (GENERATOR AND accumulator
    [8])
```

```
38
39      ControlUnit.mid_sel = 0;
40      ControlUnit.out_en = 1;
41
42
43  @posedge(clock 57):
44      # output
45      out[7:0] <- accum[7:0]
46      out[63:8] <- msg[55:0]
47
48      ControlUnit.out_en = 0;
49      ControlUnit.in_en = 1;
```

Listing 2.1: FSM pseudocode of bitwise architecture

A total of 5 different phases can be identified. This is reflected in the internal logic of the relative Control Unit

2.2 LUT-based architecture

Figure 2.2 shows a high-level view of the LUT-based architecture of the CRC (clocks and resets are not shown for the sake of clarity).
As it can be seen, the structure is quite similar to the bitwise one: it only differs by some minor changes in the sizes of signals and registers and in the feedback loop that goes back into the accumulator.
The light blue block is a combinatorial network that contains a pre-computed 1x256B look-up table.

The LUT-based CRC works as follows:

- Input data is read from `IN` and `MD` and stored into the input registers `msg`, `crc` and `md`
- Input data is then fed into the processing section of the circuit, which computes the polynomial long division advancing byte by byte.
- At the end of the computation, the original message along with the CRC (or the CRC check, depending on the input value of `MD`) is placed into the output register `out`.

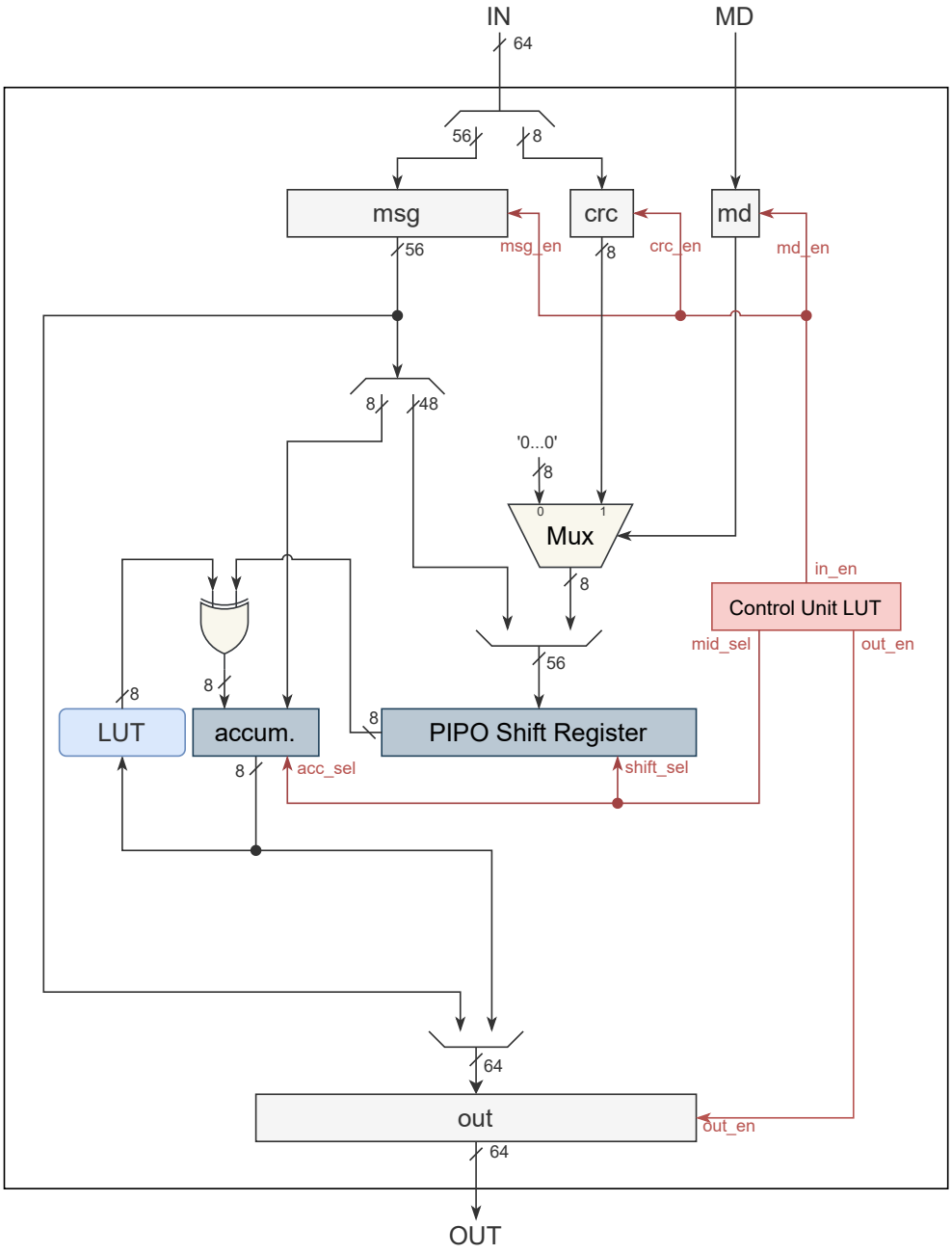


Figure 2.2: Block diagram of the LUT-based implementation of the CRC.

The following pseudocode shows what happens during each clock cycle, in a finite-state machine fashion:

```
@posedge(clock 0):
    #input
    msg[56:8] <- IN[63:8]
    crc[7:0] <- IN[7:0]
    md <- MD

    ControlUnit.in_en = 0;
    ControlUnit.mid_sel = 0;
    ControlUnit.out_en = 0;

@posedge(clock 1):
    # shift register
    PIPOShiftRegister[55:8] <- msg[47:0]
    PIPOShiftRegister[7:0] <- md == 0 ? '00000000' : crc[7:0]

    # accumulator
    accumulator[7:0] <- msg[55:48]

    ControlUnit.mid_sel = 1;

 @posedge(clock 2..7):
    # shift register
    for i in range(8, 55):
        PIPOShiftRegister[i] <- [i-8]

    PIPOShiftRegister[7:0] <- '00000000'

    # accumulator
    accumulator[7:0] <- crc_LUT(accumulator[7:0]) XOR PIPOShiftRegister
    [55:48]

 @posedge(clock 8):
    # shift register
    for i in range(8, 55):
        PIPOShiftRegister[i] <- [i-8]
    PIPOShiftRegister[7:0] <- '00000000'
    # accumulator
    accumulator[7:0] <- crc_LUT(accumulator[7:0]) XOR PIPOShiftRegister
    [55:48]

    ControlUnit.mid_sel = 0;
    ControlUnit.out_en = 1

@posedge(clock 9):
    # output
    out[7:0] <- accum[7:0]
    out[63:8] <- msg[55:0]

    ControlUnit.out_en = 0;
    ControlUnit.in_en = 1;
```

Listing 2.2: FSM pseudocode of LUT based architecture

Analogously to the previous architecture, a total of 5 different phases can be identified in this one too.

As it can be seen by the clock numbers, the actual computation phases in this architecture are only 7 clock cycles long, against 55 clock cycles of the former architecture. Both architecture further have 2 preliminary clock cycles to read the inputs and 1 final clock cycle to set the output. This yields a total of 58 and 10 clock cycles respectively. The speed-up factor obtained from the second implementation is therefore equal to 5.8, provided the clock frequency is equal. This clock frequency assumption will be tested in chapter 5 during the synthesis process.

2.3 Subcomponents

In this section the various subcomponents that make up the two architectures will be analyzed.

2.3.1 Double Data Flip Flop (D2FF)

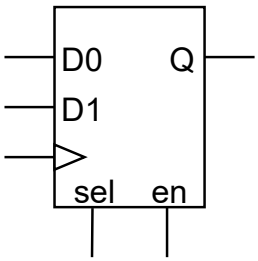


Figure 2.3: D2FF.

The **D2FF (Double Data Flip Flop)** is the basic unit the accumulator and the shift register are made of.
It is similar to a common D Flip-Flop, with the only difference that it has two different data inputs (hence the name), `d0` and `d1`, and a `sel` input which controls which of the two data inputs goes to the output `q`.
Its truth table is the following:

clk	en	sel	D0	D1	Q_{next}
non-rising	X	X	X	X	Q
rising	0	X	X	X	Q
	1	0	0	X	0
			1		1
		1	X	0	0
				1	1

2.3.2 D2FF-N

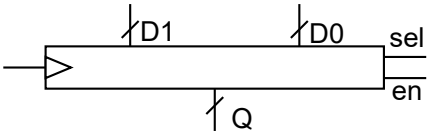


Figure 2.4: D2FF-N.

The accumulator is an instance of a **D2FF-N**, i.e. a register of N bits with 2 N-bit data inputs and a `sel` input to toggle between them.
Just as classical registers are arrays of D Flip Flops, a D2FF-N is an array of D2FF: its `en`, `sel` and `clk` are mapped to the respective input of each individual D2FF and the i-th bit of the inputs and the output is mapped accordingly to the respective input or output of the i-th D2FF.

2.3.3 PIPO Shift Register

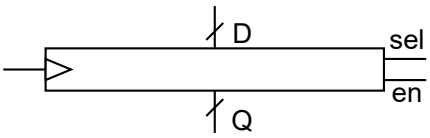


Figure 2.5: PIPO Shift Register.

The **PIPO (Parallel-Input-Parallel-Output) Shift Register** is a shift register whose input and output can be written/read in parallel. It is also an array of D2FF, but the `d1` input of each D2FF is connected to the `q` output of the previous one (except for the very first D2FF which always reads `0`). The `sel` input therefore acts as a *switch* which can enable/disable the shifting.

2.3.4 Control Unit

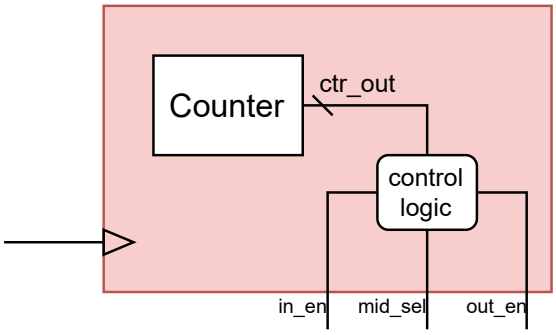


Figure 2.6: Control Unit.

The **Control Unit** is made of a counter plus some simple control logic that, depending on the output of the counter, issues the correct commands via the output signals `in_en`, `mid_sel` and `out_en`, the three of which are connected respectively to the input registers *enable*, the accumulator and shift register *select* and the output register *enable*.
Depending on the architecture, the control unit will differ in the maximum count and the control logic (cfr. pseudocode in Listings 2.1 and 2.2).

Both architectures were designed with the following assumption in mind:
the upstream and downstream circuits (with respect to the CRC circuit) are both synchronous with the designed component and they know how many clock every implementation takes to compute the CRC.
Should this assumption fall, a handshake mechanism would be required, both for the producer (upstream) component and for the consumer (downstream) component.
The *phases-structured* logic lends itself well to this type of addition: the two handshakes would be connected to the Control Unit, which could pause/resume the advance of the counter and keep the circuit in the first (last) phase as long as it is required for the producer or consumer to write or read the data.

Chapter 3

VHDL code

This chapter consists of an in-depth analysis of the VHDL codebase of the project, commenting the more interesting parts and providing the reasons behind the development choices that were made.
While the initial development process followed a top-down approach to find the most appropriate architecture(s) and identify the components, the actual coding process followed a **bottom-up** approach, starting from the simplest components and basic units and going up to build progressively more complex modules.
As mentioned in chapter 2, the development process was carried out with modularity in mind: this allowed to ease the reuse of components and simplify debugging and maintenance. Furthermore, some well-known VHDL best practices were followed, especially for naming signals, constants and modules.

The directory structure of the project is the following:

```
CRC
├── modelsim
├── src
│   ├── DFF.vhd
│   ├── DFF_N.vhd
│   ├── D2FF.vhd
│   ├── D2FF_N.vhd
│   ├── PIPOShiftReg.vhd
│   ├── xor_logical.vhd
│   ├── xor_LUT.vhd
│   ├── control_unit/
│   ├── CRC_bitwise.vhd
│   └── CRC_LUT.vhd
├── tb
└── vivado
```

In order not to clutter the document too much, only the most important parts of each source file will be displayed here. The complete source files can be found in the project repository linked in chapter 1.

3.1 D2FF

```
1  entity D2FF is
2      port (
3          clk      :   in  std_logic;
4          a_rst_n  :   in  std_logic;
5          en       :   in  std_logic;
6          sel      :   in  std_logic;
7          d0       :   in  std_logic;
8          d1       :   in  std_logic;
9          q        :   out std_logic
10     );
11 end entity D2FF;
12
13 architecture rtl of D2FF is
14
15     constant RST_VAL    :   std_logic   := '0'; -- reset is active low
16
17 begin
18
19    d2ff_p: process(clk, a_rst_n)
20    begin
21         if a_rst_n = RST_VAL then -- asynch reset
22             q    <=  '0';
23         elsif rising_edge(clk) then
24             if en = '1' then
25                 if sel = '0' then
26                     q <= d0;
27                 elsif sel = '1' then
28                     q <= d1;
29                 end if;
30             end if;
31        end if;
32    end process d2ff_p;
33
34 end architecture rtl;
```

Listing 3.1: VHDL source code of D2FF

As 3.1 shows, the source for D2FF is quite simple and very similar to a typical description of a classic D Flip Flop: the only difference is the presence of an additional `if..else` statement needed to multiplex the two input data signal by means of the `sel` input.

3.2 D2FF-N

The D2FF-N source code is exactly the same as the D2FF, except for the type of the input ports `d0` and `d1` and the output port `q`, which are all declared as `std_logic_vectors` and the presence of a `generic` which allows to size the component to suit one's needs.

3.3 PIPO Shift Register

```
entity PIPOShiftReg is
    generic(
        ShiftReg_size   : positive  := 8;
        ShiftLen        : natural   := 1
    );
    port(
        clk     :   in  std_logic;
        reset   :   in  std_logic;
        sel     :   in  std_logic;
        d       :   in  std_logic_vector(ShiftReg_size - 1 downto 0);
        q       :   out std_logic_vector(ShiftReg_size - 1 downto 0)
    );
end entity PIPOShiftReg;

architecture struct of PIPOShiftReg is

    component D2FF
        -- [...]
    end component D2FF;

    signal q_s  : std_logic_vector(ShiftReg_size - 1 downto 0);

begin
    -- generation of N instances of the Double Data flip-flop
    GEN: for i in 0 to ShiftReg_size - 1 generate
        -- first D2FF, d1 is always '0'
        FIRST: if i < ShiftLen generate
            FF_1: D2FF
            port map(
                clk         =>  clk,
                a_rst_n     =>  reset,
                en          =>  '1',
                sel         =>  sel,
                d0          =>  d(i),
                d1          =>  '0',
                q           =>  q_s(i)
            );
        end generate FIRST;

        LAST: if i >= ShiftLen generate
            FF_I: D2FF
            port map(
```

```
43                  clk          =>  clk,
44                  a_rst_n      =>  reset,
45                  en           =>  '1',
46                  sel          =>  sel,
47                  d0           =>  d(i),
48                  d1           =>  q_s(i-ShiftLen),
49                  q            =>  q_s(i)
50              );
51          end generate LAST;
52
53      end generate GEN;
54      q <= q_s;
55
56  end architecture struct;
```

Listing 3.2: VHDL source code of the PIPO Shift Register

Listing 3.3 shows the source code for the PIPO Shift Register. As mentioned in 2.3.3, its building unit is the D2FF. The presence of the generic `ShiftLen` at line 4 made this component suitable for both the chosen architectures: `ShiftLen` will be equal to 1 for the bitwise logic and it will be equal to 8 for the bytewise logic.
The port maps at lines 32 and 45 to the `en` input of the D2FF are set to `1` since in neither of the architectures there is the need to stop the shifting.

3.4 XOR logical

```
1  entity XOR_logical is
2      generic(
3          XOR_input_size : positive := 9
4      );
5      port (
6          d_in     : in    std_logic_vector(XOR_input_size - 1 downto 0);
7          d_out    : out   std_logic_vector(XOR_input_size - 2 downto 0)
8      );
9  end entity XOR_logical;
10
11 architecture rtl of XOR_logical is
12     constant C_GENERATOR_LEN  :   natural := 8;
13     constant C_GENERATOR      :   std_logic_vector(C_GENERATOR_LEN - 1
       downto 0) := "00011101";
14
15 begin
16     do_xor: for i in C_GENERATOR_LEN - 1 downto 0 generate
17         d_out(i) <= d_in(i) xor (C_GENERATOR(i) and d_in(XOR_INPUT_SIZE -
        1));
18     end generate;
19 end architecture rtl;
```

Listing 3.3: VHDL source code of the XOR logical component

The **XOR_logical** component is a combinatorial component which implements a single subtraction step needed for the polynomial long division procedure.
It computes the XOR between the 8 least significant bits of the input and the generator. As mentioned in 2.1, the input most significant bit is used as an "enable", i.e. when it is `0` the circuit simply outputs the input 8 least significant bit. This is implemented in line 17 by exploiting the fact that `0` is the neutral element of the XOR operation.

3.5 XOR LUT

```
1  entity XOR_LUT is
2      generic(
3          XOR_LUT_input_size : positive := 8
4      );
5      port(
6          d_in  : in  std_logic_vector(XOR_LUT_input_size - 1 downto 0);
7          d_out : out std_logic_vector(XOR_LUT_input_size - 1 downto 0)
8      );
9  end entity XOR_LUT;
10
11 architecture rtl of XOR_LUT is
12
13     constant GENERATOR_LEN  :   natural := 8;
14
15     signal  addr_int     :   integer range 0 to 255;
16
17     type lut_t is array (0 to 255) of std_logic_vector(GENERATOR_LEN -1
       downto 0);
18
19     constant lut: lut_t :=
20     (
21         x"00", x"1D", x"3A", x"27", x"74", x"69", x"4E" -- [etc...]
22     );
23 begin
24     addr_int    <= TO_INTEGER(unsigned(d_in));
25     d_out       <= lut(addr_int);
26 end architecture rtl;
```

Listing 3.4: VHDL source code of the XOR LUT component

The **XOR_LUT** component is again a combinatorial component which contains a look-up table for the specified generator polynomial.
It computes the XOR between the 8 least significant bits of the input and the generator. The input is converted into an integer and then used as an index to retrieve the corresponding entry of the LUT.
Since the input is 8 bits long and so are the output, the total size of the LUT is 256 B. The LUT was generated with a simple Python script.

3.6 Control Unit

```
CRC/src/control_unit/
├── fullAdder.vhd
├── rippleCarryAdder.vhd
├── counter.vhd
└── controlUnit.vhd
```

The `control_unit` directory contains the source code for the Control Unit along with the subcomponent it is made of.
The source code of the **full adder** and the **ripple-carry adder** will not be shown here since they are both very simple.

3.6.1 Counter

```
1  entity Counter is
2    generic
3    (
4      N_cycles    : natural   := 58
5    );
6      port(
7      clk        : in  std_logic;
8      a_rst_n    : in  std_logic;
9      increment : in  std_logic_vector(natural(ceil(log2(real(N_cycles))))
       - 1 downto 0);
10     cntr_out   : out std_logic_vector(natural(ceil(log2(real(N_cycles))))
       - 1 downto 0)
11     );
12 end Counter;
13
14 architecture struct of Counter is
15
16   constant C_COUNTER_BITS : natural := natural(ceil(log2(real(N_cycles)))
       );
17   constant C_N_CYCLES : std_logic_vector := std_logic_vector(to_unsigned(
       N_cycles, C_COUNTER_BITS));
18   constant C_A_RST_VALUE  : std_logic :=  '0';
19
20   signal fullAdder_out  : std_logic_vector(C_COUNTER_BITS - 1 downto 0)
         := (others => '0');
21   signal dffn_out     : std_logic_vector(C_COUNTER_BITS - 1 downto 0)
       := (others => '0');
22   -- Needed to implement the max count check
23   signal accumulator    : std_logic_vector(C_COUNTER_BITS - 1 downto 0)
       := (others => '0');
24
25   component RippleCarryAdder is
26     -- [...]
27   end component;
28
```

```
29    component DFF_N is
30      -- [...]
31    end component;
32
33    begin
34
35      RIPPLE_CARRY_ADDER_MAP :  RippleCarryAdder
36        generic map(RCA_N =>  C_COUNTER_BITS)
37        port map(
38          a    => increment,
39          b    => accumulator,
40          cin  => '0',
41          s    => fullAdder_out,
42          cout => open
43        );
44
45      DFF_N_MAP : DFF_N
46        generic map(DFF_N_size => C_COUNTER_BITS)
47        port map(
48          clk     => clk,
49          a_rst_n => a_rst_n,
50          d       => fullAdder_out,
51          en      => '1',
52          q       => dffn_out
53            );
54
55        accumulator <= dffn_out when dffn_out < C_N_CYCLES
56            else (others => '0');
57      cntr_out <= accumulator;
58
59  end struct;
```

Listing 3.5: VHDL source code of the Counter

Listing 3.6.1 shows the VHDL source code for the Counter.
It is made of a ripple-carry adder, a register of D Flip-Flop and some logic to wrap the count back to 0 when the maximum value is reached.
As with the PIPO Shift Register, this Counter also has a generic which allows the user to set the max count. When instancing a Control Unit component (v. next subsection), this will also have a generic for the number of cycles; this generic will be inherited by the Counter generic.

3.6.2 Control Unit

```
entity ControlUnit is
    generic(
        CU_cycles   :   natural := 58
    );
    port(
        clk     :   in  std_logic;
        a_rst_n :   in  std_logic;
        in_en   :   out std_logic;
        mid_sel :   out std_logic;
        out_en  :   out std_logic
    );
end entity ControlUnit;

architecture beh of ControlUnit is

    -- Constants
    constant C_CTR_INCREMENT  :   natural :=  1;
    constant C_CTR_CYCLES     :   natural :=  CU_cycles;

    constant C_PHASE_0_END    :   natural :=  0;
    constant C_PHASE_1_END    :   natural :=  2;
    constant C_PHASE_2_END    :   natural :=  C_CTR_CYCLES - 3;
    constant C_PHASE_3_END    :   natural :=  C_CTR_CYCLES - 2;
    constant C_PHASE_4_END    :   natural :=  C_CTR_CYCLES - 1;

    constant A_RST_VALUE    :   std_logic := '0';

    -- Signals
    signal  cntr_out_s      :   std_logic_vector(natural(ceil(log2(real(
    C_CTR_CYCLES)))) -1 downto 0) := (others => '0');

    -- Components
    component Counter is
        -- [...]
    end component;

begin

    COUNTER_MAP : Counter
        generic map(
            N_cycles => C_CTR_CYCLES
            )
        port map(
            clk         =>  clk,
            a_rst_n     =>  a_rst_n,
            increment   =>  std_logic_vector(to_unsigned(C_CTR_INCREMENT,
     natural(ceil(log2(real(C_CTR_CYCLES))))),
            cntr_out    =>  cntr_out_s
        );


```

```
50      drive_out_signals: process(clk, cntr_out_s, a_rst_n)
51          variable current_count_v  :   integer := 0;
52      begin
53          if(a_rst_n = A_RST_VALUE) then
54              null;
55          elsif(rising_edge(clk)) then
56              current_count_v := to_integer(unsigned(cntr_out_s));
57
58              if current_count_v <= C_PHASE_0_END then
59                  in_en   <= '0';
60                  mid_sel <= '0';
61                  out_en  <= '0';
62              elsif current_count_v <= C_PHASE_1_END then
63                  mid_sel <=  '1';
64              elsif current_count_v <= C_PHASE_2_END then
65                  null;
66              elsif current_count_v <= C_PHASE_3_END then
67                  mid_sel <=  '0'; --
68                  out_en  <=  '1';
69              elsif current_count_v <= C_PHASE_4_END then
70                  out_en <= '0';
71                  in_en <= '1';
72              else
73                  null;
74              end if;
75          end if;
76      end process drive_out_signals;
77  end architecture beh;
```

Listing 3.6: VHDL source code of the Control Unit

Listing 3.6.2 shows the VHDL source code of the Control Unit.
Thanks to the generic `CU_cycles`, both the CRC architectures can use the same *ControlUnit* component, by just changing the generic map upon instantiation.
This was made possible by avoiding the so-called "magic numbers" in the code and by noticing that all the phases can be defined in terms of the same offsets with respect to the start or the end of the computation, regardless of the architecture.

3.7 Bitwise CRC

```
entity CRC_bitwise is
    generic(
        msg_size    : natural := 56;
        CRC_size    : natural := 8
    );
    port(
        clk     :   in  std_logic;
        a_rst_n :   in  std_logic;
        d_in    :   in  std_logic_vector(msg_size + CRC_size - 1 downto 0);
        md      :   in  std_logic;
        d_out   :   out std_logic_vector(msg_size + CRC_size - 1 downto 0)
    );
end entity CRC_bitwise;

architecture struct of CRC_bitwise is

    -- declaration of the following components: DFF, DFF_N, D2FF_N
    -- PIPOShiftReg, XOR_logical, ControlUnit
    -- declarations of constants...

    signal  msg_reg_out     :   std_logic_vector(C_MSG_SIZE - 1 downto 0) := (others => '0');
    signal  crc_reg_out     :   std_logic_vector(C_CRC_SIZE - 1 downto 0) := (others => '0');
    signal  md_reg_out      :   std_logic   := '0';

    signal  mux_out         :   std_logic_vector(C_CRC_SIZE - 1 downto 0) := (others => '0');

    signal  accum_d1        :   std_logic_vector(C_ACCUM_SIZE - 1 downto 0) := (others => '0');
    signal  accum_out       :   std_logic_vector(C_ACCUM_SIZE - 1 downto 0) := (others => '0');

    signal  xor_out         :   std_logic_vector(C_XOR_OUTPUT_SIZE - 1 downto 0) := (others => '0');

    signal  shift_reg_in  :  std_logic_vector(C_SHIFT_REG_SIZE - 1 downto 0) := (others => '0');
    signal  shift_reg_out :  std_logic_vector(C_SHIFT_REG_SIZE - 1 downto 0) := (others => '0');

    signal  out_reg_in :  std_logic_vector(C_OUT_SIZE - 1 downto 0) := (others => '0');

    signal  CU_in_en    :   std_logic   := '0';
    signal  CU_mid_sel  :   std_logic   := '0';
    signal  CU_out_en   :   std_logic   := '0';

```

```
41      begin
42
43      msg_reg : DFF_N
44          generic map(
45              DFF_N_size  =>  C_MSG_SIZE
46          )
47          port map(
48              en      => CU_in_en,
49              d       => d_in(C_MSG_SIZE + C_CRC_SIZE - 1 downto C_CRC_SIZE
    ),
50              q       => msg_reg_out
51          );
52
53      crc_reg : DFF_N
54          generic map(
55              DFF_N_size  =>  C_CRC_SIZE
56          )
57          port map(
58              en      => CU_in_en,
59              d       => d_in(C_CRC_SIZE - 1 downto 0),
60              q       => crc_reg_out
61          );
62
63      md_reg : DFF
64          port map(
65              en      => CU_in_en,
66              d       => md,
67              q       => md_reg_out
68          );
69
70      accumulator : D2FF_N
71          generic map(
72              D2FF_N_size => C_ACCUM_SIZE -- 9
73          )
74          port map(
75              en      => '1',
76              sel     => CU_mid_sel,
77              d0      => msg_reg_out(C_MSG_SIZE - 1 downto C_MSG_SIZE -
    C_ACCUM_SIZE),
78              d1      => accum_d1,
79              q       => accum_out
80          );
81
82      shift_register : PIPOShiftReg
83          generic map(
84              ShiftReg_size   => C_SHIFT_REG_SIZE, -- 55
85              ShiftLen        => C_SHIFT_LEN -- 1
86          )
87          port map(
88              sel     =>  CU_mid_sel,
89              d       =>  shift_reg_in,
90              q       =>  shift_reg_out
91          );
```

```
92
93      xor_log : XOR_logical
94          generic map(
95              XOR_input_size  =>  C_XOR_INPUT_SIZE -- 9
96          )
97          port map(
98              d_in    =>  accum_out,
99              d_out   =>  xor_out
100         );
101
102     CU : ControlUnit
103         generic map(
104             CU_cycles => C_N_CYCLES -- 58
105         )
106         port map(
107             in_en   => CU_in_en,
108             mid_sel => CU_mid_sel,
109             out_en  => CU_out_en
110         );
111
112     out_reg : DFF_N
113         generic map(
114             DFF_N_size => C_OUT_SIZE
115         )
116         port map(
117             en      => CU_out_en,
118             d       => out_reg_in,
119             q       => d_out
120         );
121
122
123     mux_proc: process(clk, md_reg_out, crc_reg_out)
124     begin
125         if(md_reg_out = '1') then
126             mux_out <= crc_reg_out;
127         else
128             mux_out <= (others => '0');
129         end if;
130     end process mux_proc;
131
132     shift_reg_in    <= msg_reg_out(C_MSG_SIZE - C_ACCUM_SIZE - 1 downto
        0) & mux_out;
133     accum_d1        <= xor_out & shift_reg_out(C_SHIFT_REG_SIZE - 1);
134     out_reg_in      <= msg_reg_out & xor_out;
135
136     end architecture struct;
```

Listing 3.7: VHDL source code of the bitwise architecture of the CRC

Listing 3.7 shows the VHDL source code of the bitwise architecture of the CRC. Components and constants declarations were omitted to avoid cluttering, as well as the clock and reset mapping for each component. Please refer to the repository for the full source

code.
The structure of the CRC source code is actually quite straightforward, as it mainly consists of the previously shown subcomponents and signals that interconnect them.
The `mux_proc` process at line 123, as the name suggests, implements the multiplexing process that feeds the 8 least significant bits into the shift register, depending on the value of `MD`.
The signal assignments at the end of the architecture use the concatenation operator `&` to implement the "merging" of the signals.

3.8 LUT-based CRC

Given the strong similarity of the LUT-based CRC implementation with respect to the bitwise one, below is a highly reduced version of VHDL source code of the former, which only contains the parts that differ from listing 3.7.
It is worth mentioning that the CRC "interface" (i.e. the set of input and output ports) is obviously the same in both architectures and compliant with the project specifications.

```
1  entity CRC_LUT is
2      -- [...]
3  end entity CRC_LUT;
4
5  architecture struct of CRC_LUT is
6
7      -- declaration of the following components: DFF, DFF_N, D2FF_N
8      -- PIPOShiftReg, XOR_logical, ControlUnit
9
10     -- ... declarations of constants and signals ...
11
12 begin
13     -- instatiation and map of various components
14     -- ...
15
16     accumulator : D2FF_N
17         generic map(
18             D2FF_N_size => C_ACCUM_SIZE -- 8
19         )
20         port map(
21             -- [...]
22         );
23
24     shift_register : PIPOShiftReg
25         generic map(
26             ShiftReg_size   => C_SHIFT_REG_SIZE, -- 56
27             ShiftLen        => C_SHIFT_LEN -- 8
28         )
29         port map(
30             -- [...]
31         );
32
33     lut : XOR_LUT
34         generic map(
```

```
35              XOR_LUT_input_size  =>  C_LUT_INPUT_SIZE -- 8
36          )
37          port map(
38              d_in    =>  accum_out,
39              d_out   =>  lut_out
40          );
41
42      CU : ControlUnit
43          generic map(
44              CU_cycles => C_N_CYCLES -- 10
45          )
46          port map(
47              -- [...]
48          );
49
50      mux_proc: process(clk, md_reg_out, crc_reg_out)
51          -- [...]
52      end process mux_proc;
53
54      shift_reg_in    <= msg_reg_out(C_MSG_SIZE - C_ACCUM_SIZE - 1 downto
        0) & mux_out;
55      accum_d1        <= lut_out xor shift_reg_out(C_SHIFT_REG_SIZE - 1
        downto C_SHIFT_REG_SIZE - C_ACCUM_SIZE);
56      out_reg_in      <= msg_reg_out & accum_out;
57
58  end architecture struct;
```

Listing 3.8: VHDL source code of the LUT-based architecture of the CRC

As listing 3.8 shows, the only few differences with the bitwise version are some minor changes in the size of some signals and registers, the presence of the **XOR_LUT** combinatorial block in place of the previously used **XOR_logical** and the assignment to the `accum_d1` signal at line 54. At each step, the next input of the accumulator consists of the bitwise XOR between the appropriate LUT entry and the next byte in the message.

Chapter 4

Test-plan and testbench

The design of each component was followed by a testing phase performed via testbench modules, also written in VHDL, to ensure that said component was designed properly. Only the testbench of the final CRC modules will be presented in this chapter. Please refer to the repository for the rest of them.

All the simulations of the testbenches were performed on *ModelSim - Intel FPGA Starter Edition v. 2020.1.*

4.1 CRC testbench

As mentioned in 3.8, the set of input and output ports of the CRC is the same, regardless of the implementation, because of the compliance with the specifications. This naturally led to a single testbench that could test both versions.

A Python script was used to generate 10 random 7 bytes long messages and compute the CRC of each of them.

- `0526abfa59289d 75`
- `1ad743298a5b0c 13`
- `49dbf2d3fca778 7a`
- `58de7943c3b4e1 f7`
- `7a32768bdb8fb4 58`
- `8d73243271fdf2 86`
- `c387f7b71ddd50 2e`
- `d8c66625791098 b7`
- `e34a300fa4c345 1d`
- `f9e70f4d2b6ed3 89`

The testbench consists of three phases:

- sender mode
- receiver mode with intact messages and correct CRCs
- receiver mode with corrupted messages

During the first phase `MD` is equal to `0` and the 10 random messages are input in succession. The output should show each of the 10 messages, in succession, and the correct CRC as the least significant byte.
During the second phase, `MD` is equal to `1` and the input consists of the 10 random messages each with its own CRC attached, in succession. The output should show each of the 10 messages, in succession, and the least significant byte equal to `0x00`. The third phase is equal to the second phase, except for the output which should show non-zero values, proving the error detecting properties of the CRC algorithm.
Below are some screenshots from the testing phase on Modelsim.

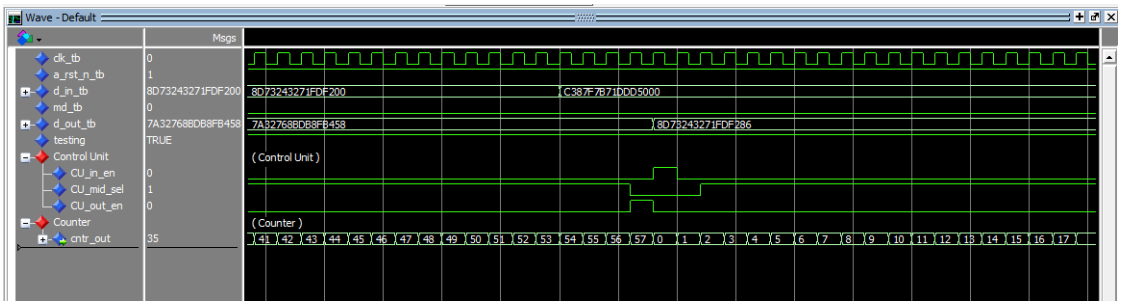


Figure 4.1: Testbench simulation of the bitwise architecture (phase 1)

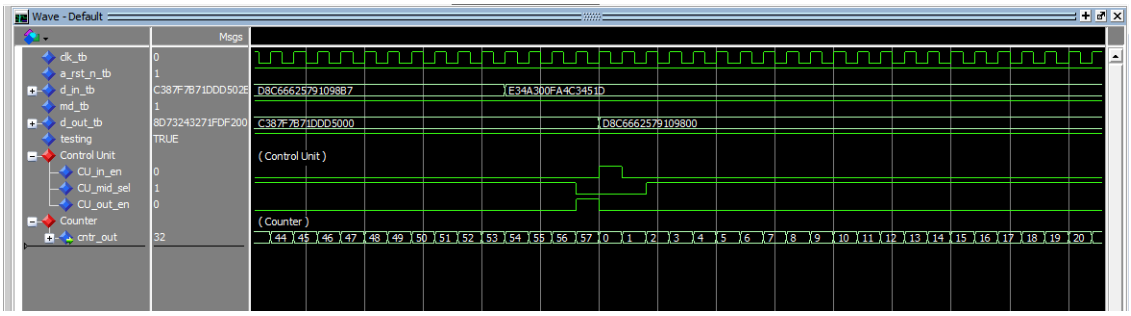


Figure 4.2: Testbench simulation of the bitwise architecture (phase 2)

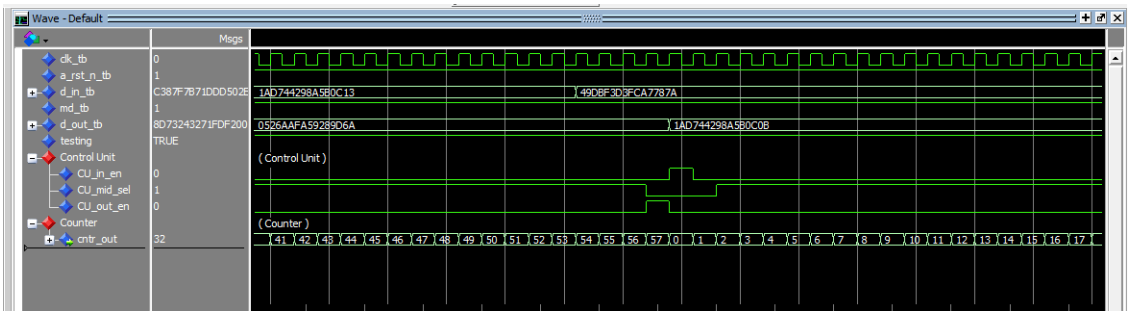


Figure 4.3: Testbench simulation of the bitwise architecture (phase 3)

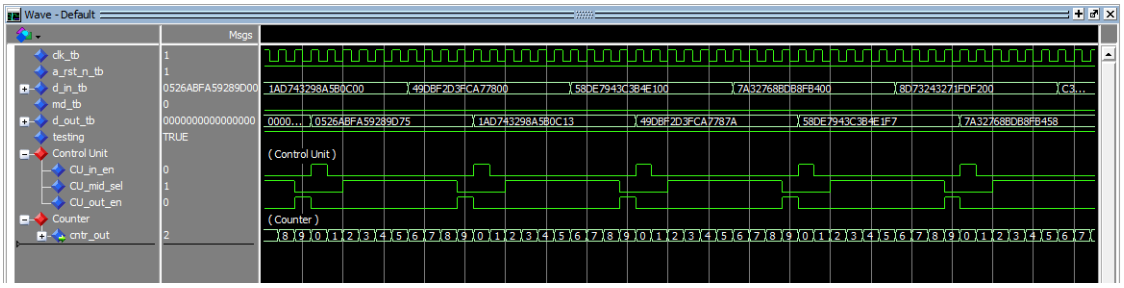


Figure 4.4: Testbench simulation of the LUT-based architecture (phase 1)

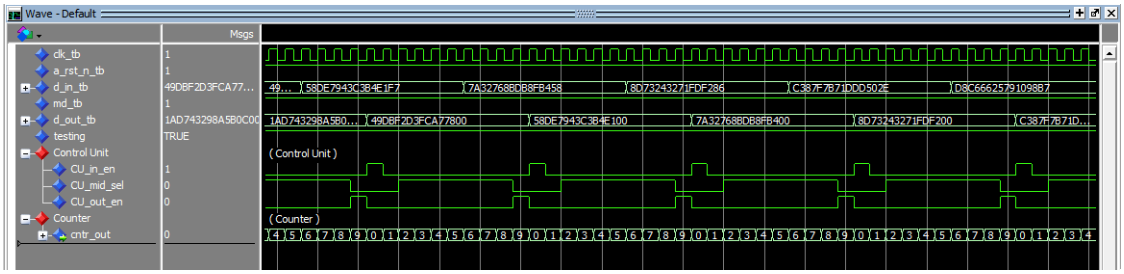


Figure 4.5: Testbench simulation of the LUT-based architecture (phase 2)

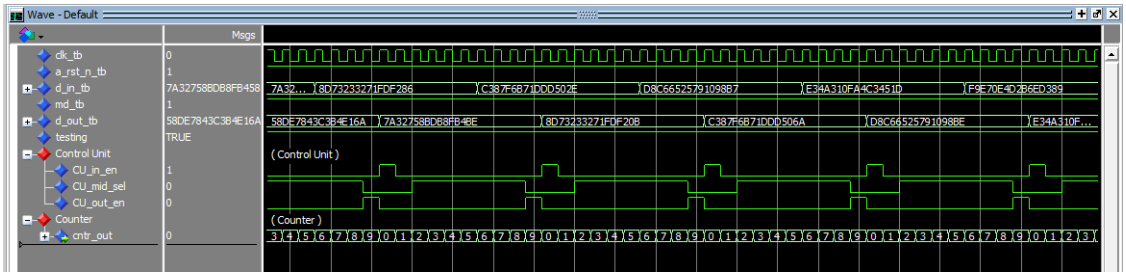


Figure 4.6: Testbench simulation of the LUT-based architecture (phase 3)

Chapter 5

Synthesis results

This chapter concerns the automated logic synthesis process on the Xilinx Vivado Design Suite. The synthesis was performed using a Xilinx Zynq-7000 general purpose board (xc7z010clg400-1) as a target SoC.
Due to an insufficient number of I/O pins available on the actual board with respect to the ones requested by the specifications, the **implementation phase** was not carried out. Nevertheless, some results regarding resource usage, maximum clock frequency, critical path and power consumption could still be obtained during the synthesis phase.

5.1 Bitwise architecture synthesis

The synthesis process on Vivado showed no errors or warnings.

5.1.1 Timing report

After adding a time constraint for a clock period of 8 ns (125 MHz), the synthesis was re-run to obtain the timing report, whose summary is shown below:

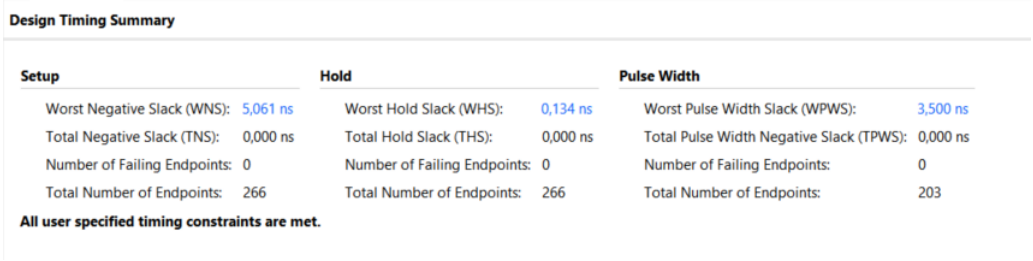
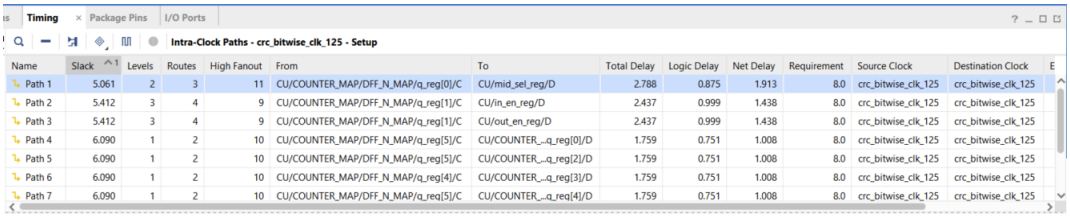


Figure 5.1: Vivado timing report summary for the bitwise architecture synthesis

The Worst Negative Slack (WNS)is determined by the critical path of the circuit, which is highlighted in the figure below:



Timing | Package Pins | I/O Ports

Intra-Clock Paths - crc_bitwise_clk_125 - Setup

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	5.061	2	3	11	CU/COUNTER_MAP/DFF_N_MAP/q_reg[0]/C	CU/mid_sel_reg/D	2.788	0.875	1.913	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 2	5.412	3	4	9	CU/COUNTER_MAP/DFF_N_MAP/q_reg[1]/C	CU/in_en_reg/D	2.437	0.999	1.438	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 3	5.412	3	4	9	CU/COUNTER_MAP/DFF_N_MAP/q_reg[1]/C	CU/out_en_reg/D	2.437	0.999	1.438	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 4	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[0]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 5	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[2]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 6	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[4]/C	CU/COUNTER_...q_reg[3]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 7	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[4]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125

Figure 5.2: Critical path of the bitwise architecture synthesis

The figure shows that the critical path of the architecture is inside the Control Unit component.
Since the WNS has a positive value, this means that the board can be driven at a higher frequency than the one set in the timing constraints file.
The maximum allowed frequency can be computed with the following formula

$$f_{max} = \frac{1}{T_{clk} - WSN} = \frac{1}{8ns - 5,061ns} = \frac{1}{2,939ns} \approx 340,25MHz \tag{5.1}$$

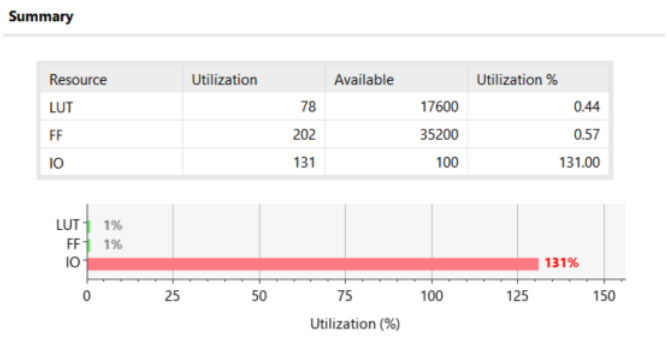
which is compatible with the 667 MHz maximum frequency stated in the board datasheet.
A simple computation can provide the throughput of the bitwise algorithm implementation: since every computation requires 58 clock cycles, the maximum number of CRCs that can be computed in a second is

$$R_{CRC} = \frac{f_{max}}{58} \approx 5866379 \tag{5.2}$$

which, in terms of bit throughput, corresponds to approximately 328.52 Mbit/s.

5.1.2 Resource utilization report

The following figure shows a summary of the resource utilization report for the bitwise architecture.



Summary

Resource	Utilization	Available	Utilization %
LUT	78	17600	0.44
FF	202	35200	0.57
IO	131	100	131.00

Figure 5.3: Summary of the resource utilization report of the bitwise architecture

As mentioned at the beginning of the chapter, the IO utilization exceeds 100% since the specifications require a total of 131 ports while the IOB on the board amount to 100. The LUT and FF utilization, on the other hand, is very low, standing at around 0.5%.

5.1.3 Power consumption report

The following figure shows a summary of the power consumption utilization report for the bitwise architecture.

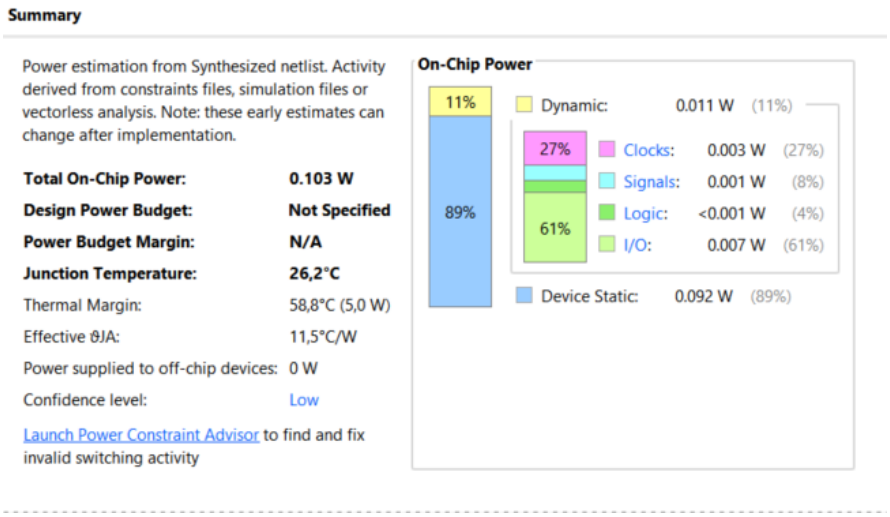


Figure 5.4: Summary of the power consumption report of the bitwise architecture

As it can be seen in the figure, the total power absorbed by the chip is around 0.1 W, most of which is static power consumption. This data however has to be taken with a grain of salt, since, as also stated in the figure, it has a low confidence level and the estimates could change after implementation.

5.2 LUT-based architecture synthesis

The same exact procedure was followed for the LUT-based architecture.

The synthesis process on Vivado showed no errors or warnings.

5.2.1 Timing report

After adding a time constraint for a clock period of 8 ns (125 MHz), the synthesis was re-run to obtain the timing report, whose summary is shown below:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5,560 ns	Worst Hold Slack (WHS):	0,139 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:	272	Total Number of Endpoints:	272	Total Number of Endpoints:	209

All user specified timing constraints are met.

Figure 5.5: Vivado timing report summary for the LUT-based architecture synthesis

The Worst Negative Slack (WNS) has increased with respect to the bitwise architecture. The critical path of the circuit is highlighted in the figure below:

Timing | Package Pins | I/O Ports

Intra-Clock Paths - crc_bitwise_clk_125 - Setup

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	5.061	2	3	11	CU/COUNTER_MAP/DFF_N_MAP/q_reg[0]/C	CU/mid_sel_reg/D	2.788	0.875	1.913	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 2	5.412	3	4	9	CU/COUNTER_MAP/DFF_N_MAP/q_reg[1]/C	CU/in_en_reg/D	2.437	0.999	1.438	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 3	5.412	3	4	9	CU/COUNTER_MAP/DFF_N_MAP/q_reg[1]/C	CU/out_en_reg/D	2.437	0.999	1.438	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 4	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[0]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 5	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[2]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 6	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[4]/C	CU/COUNTER_...q_reg[3]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125
Path 7	6.090	1	2	10	CU/COUNTER_MAP/DFF_N_MAP/q_reg[5]/C	CU/COUNTER_...q_reg[4]/D	1.759	0.751	1.008	8.0	crc_bitwise_clk_125	crc_bitwise_clk_125

Figure 5.6: Critical path of the bitwise architecture synthesis

The figure shows that the critical path is now inside the PIPOShiftRegister.
Since in this case too the WNS has a positive value, this means that the board can be driven at a higher frequency than the one set in the timing constraints file.
Again, we can compute the maximum allowed frequency:

$$f_{max} = \frac{1}{T_{clk} - WSN} = \frac{1}{8ns - 5,560ns} = \frac{1}{2,44ns} \approx 409,84MHz \tag{5.3}$$

which is higher than the other one and still compatible with the 667 MHz maximum frequency stated in the board datasheet.

The throughput of the LUT-based algorithm implementation in terms of CRCs per second is:

$$R_{CRC} = \frac{f_{max}}{10} \approx 40984000 \tag{5.4}$$

which, in terms of bit throughput, corresponds to approximately 2,295 Gbit/s.
The total speed-up from the bitwise implementation is about 600%.

5.2.2 Resource utilization report

The following figure shows a summary of the resource utilization report for the LUT-based architecture.

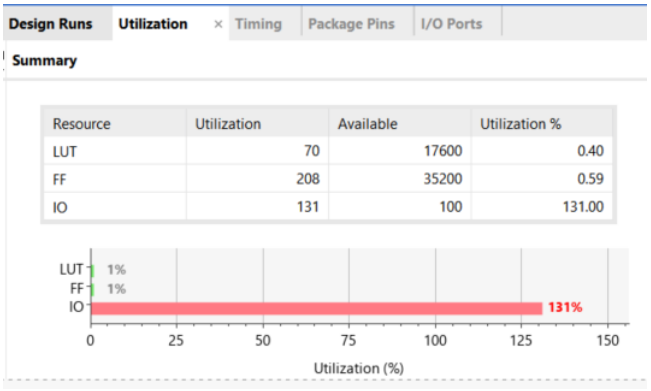


Figure 5.7: Summary of the resource utilization report of the LUT-based architecture

The report obviously present the same issue as before, as far as it concerns the IO utilization.
The LUT and FF utilization, surprisingly, are lower. This is thought to be a consequence of the optimizations performed by Vivado during the synthesis process.

5.2.3 Power consumption report

The following figure shows a summary of the power consumption utilization report for the LUT-based architecture.

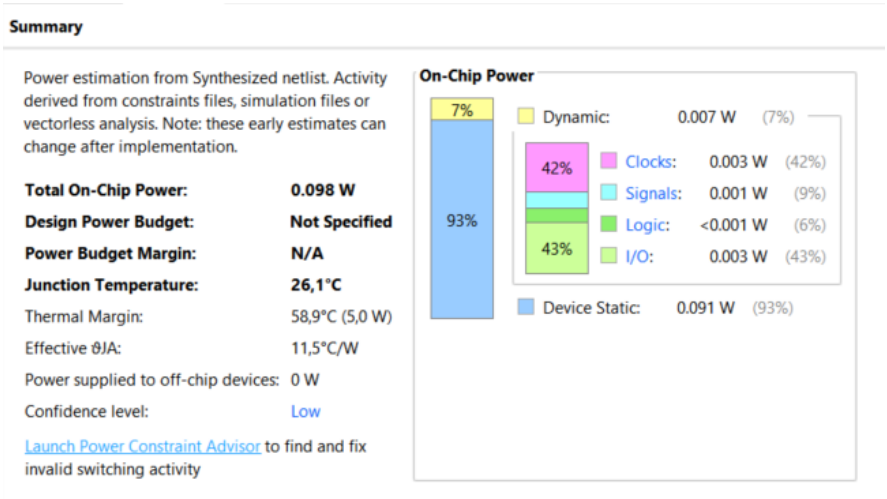


Figure 5.8: Summary of the power consumption report of the LUT-based architecture

The total power absorbed by the chip is again in the order of 0.1 W, although slightly lower than the previous implementation. In this architecture too, most of it is static power consumption.
Just as before, this values should be taken with a grain of salt for the same reasons set forth.

Chapter 6

Conclusions and possible optimizations

The LUT-based architecture seems to perform better than the bitwise one in every aspect: fewer clock cycles needed, higher maximum clock frequency, lower resource optimization. The power consumption also seems to be lower but, as stated in the previous chapter, it would be better to complete the implementation process for a greater confidence level on this aspect.

There surely is room for further optimizations, for instance one could try to improve the components which contain the critical path in order to further increase the WNS and therefore the maximum clock frequency.
Another optimization that could be made is to implement some mechanism to skip all the leading zeroes in the input, since they don't affect the final value of the CRC.

While looking for optimizations on the LUT-based architecture, it was realized that **the look-up table of CRC algorithms is associative with respect to the XOR operator**, i.e.

$$table[a] \oplus table[b] = table[a \oplus b] \tag{6.1}$$

This opens the way for a series of great optimizations: instead of having the entire LUT hardcoded on the chip, one could store only the LUT values for the powers of 2 and then perform a XOR between them depending on which bits are set in the number being used as index.