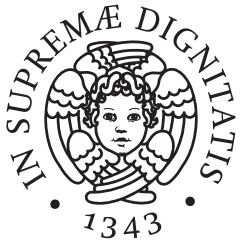

University of Pisa
MSc in Computer Engineering

Electronics and Communication Systems

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

Supervisors
Prof. Luca Fanucci
Prof. Massimiliano Donati
Ing. Pietro Nannipieri

Student
Marco Pinna

November 8, 2022

Contents

1 Introduction **4**
1.1 Algorithm description 4
1.2 Possible applications 6
1.3 Possible architectures 6

2 Architecture **8**
2.1 Bitwise architecture 8
2.2 LUT-based architecture 11
2.3 Subcomponents 14
2.3.1 Double Data Flip Flop (D2FF) 14
2.3.2 D2FF-N 15
2.3.3 PIPO Shift Register 15
2.3.4 Control Unit 16

3 VHDL code **17**

4 Test-plan **18**

5 Synthesis results **19**

6 Conclusions **20**

7 Appendices **21**

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 FPGA Zync platform
- finally in chapter 6 conclusions are drawn and some consideration about possible optimizations or different implementations are made

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.

$$
\begin{array}{l}
01101000011010010000000\underline{0} \\
\quad\underline{100011101} \\
\quad 0101111001\underline{0} \\
\quad\ \underline{100011101} \\
\quad 0011001000\underline{1} \\
\quad\quad\underline{100011101} \\
\quad\quad 010001100\underline{0} \\
\quad\quad\ \underline{100011101} \\
\quad\quad 000000101010\underline{0}00 \\
\quad\quad\quad\quad\underline{100011101} \\
\quad\quad\quad\quad 00100110100 \\
\quad\quad\quad\quad\quad\underline{100011101} \\
\quad\quad\quad\quad\quad 000\mathbf{10100100}
\end{array}
$$

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.
  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 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-component it should have consisted 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).

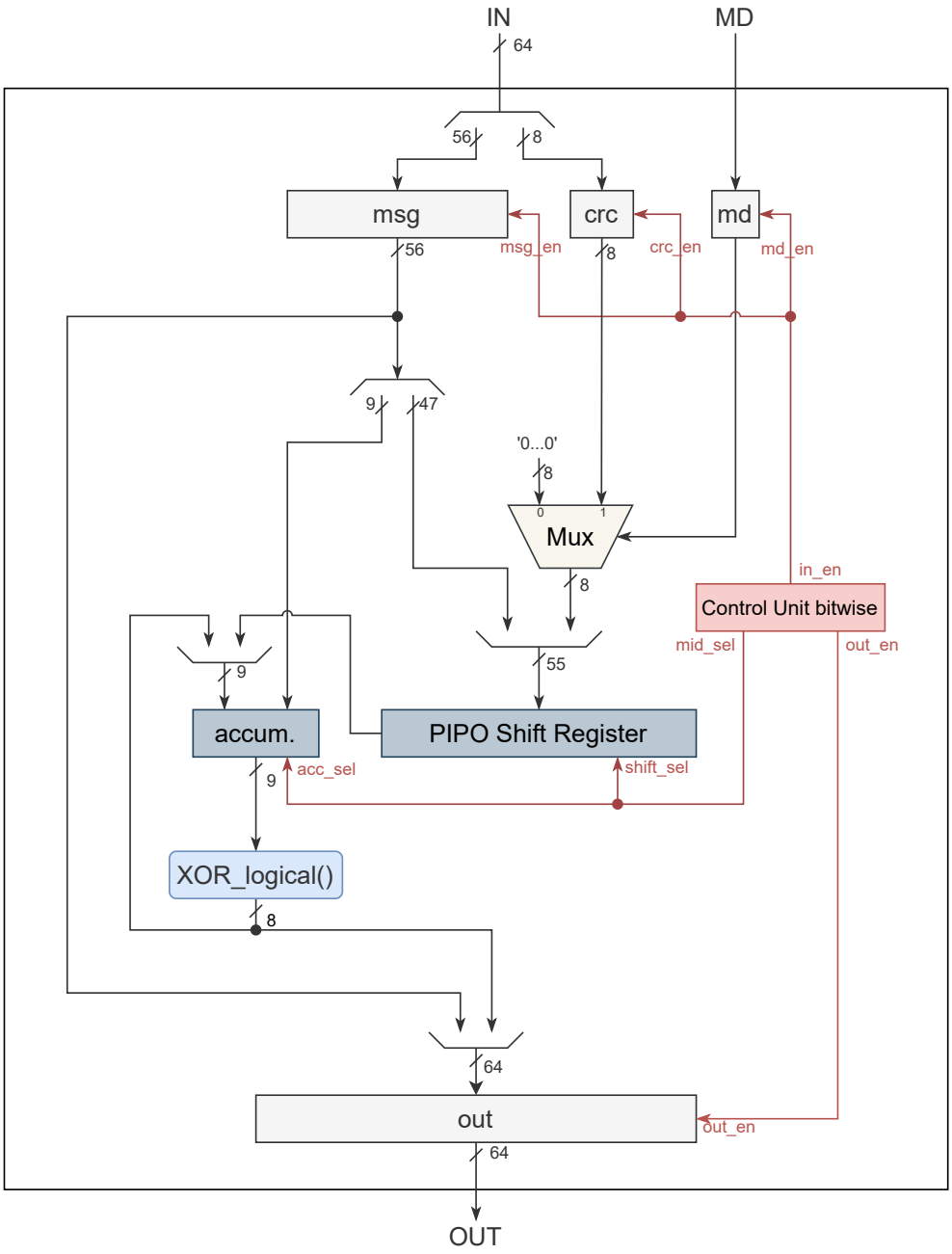

Figure 2.1: Block diagram of the bitwise implementation of the CRC.

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`.

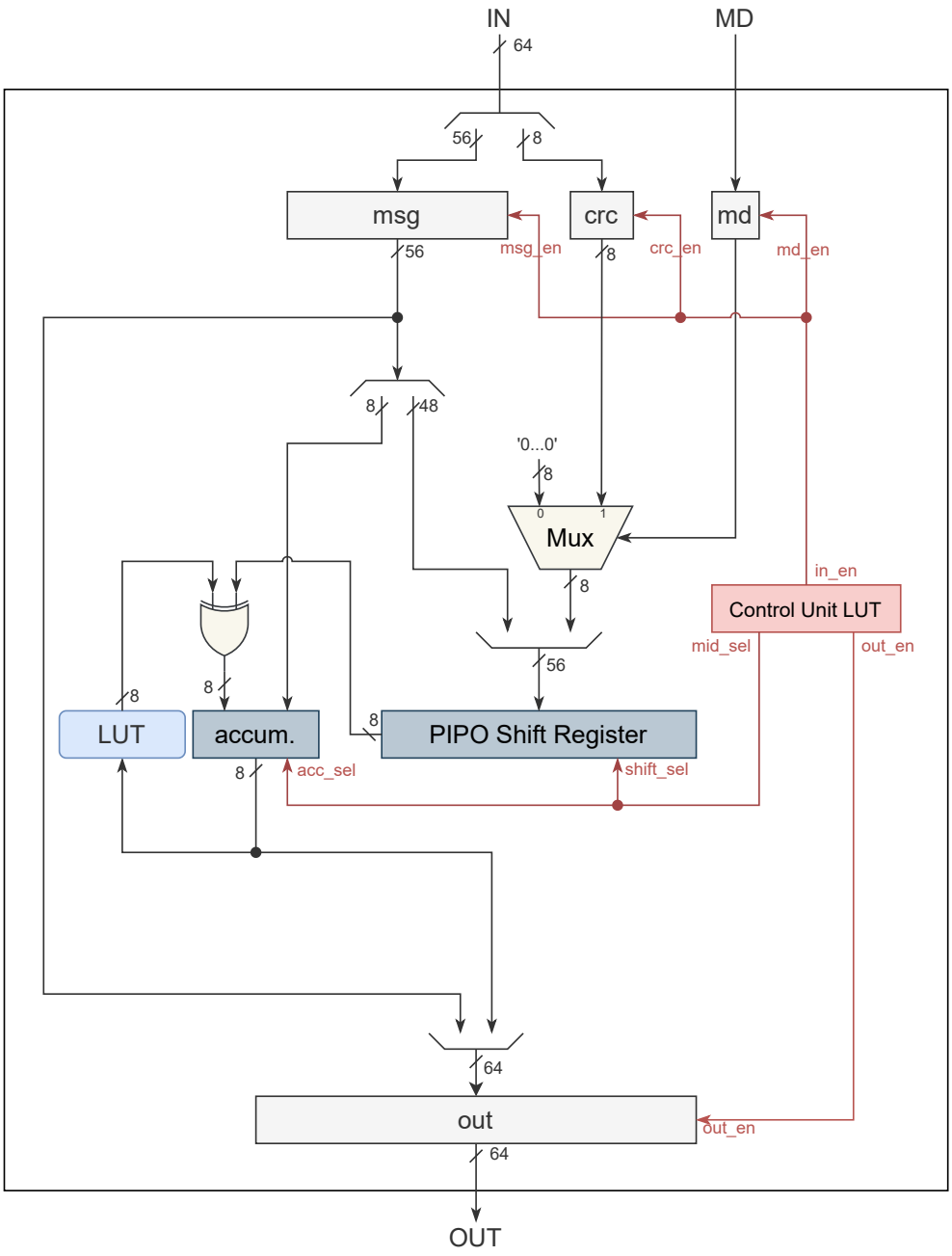

Figure 2.2: Block diagram of the LUT-based implementation of the CRC.

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)

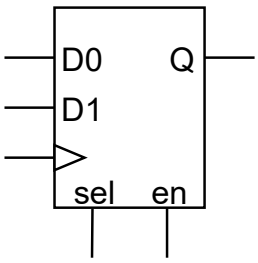

Figure 2.3: D2FF.

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

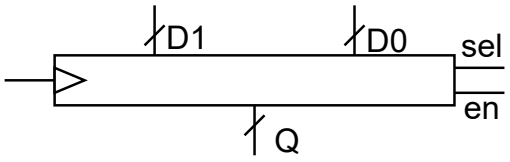

Figure 2.4: D2FF-N.

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

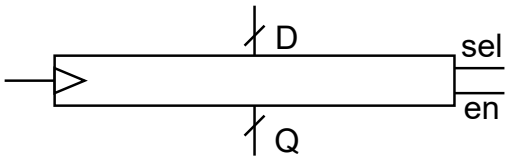

Figure 2.5: PIPO Shift Register.

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

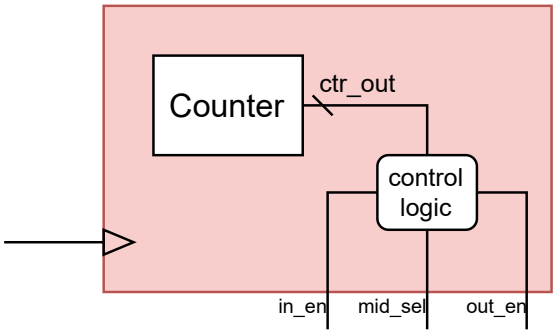

Figure 2.6: Control Unit.

The **Control Unit** is made of a simple counter plus some simple control logic that, depending on the output of the counter, issues the correct output via the three 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).

Chapter 3

VHDL code

Chapter 4

Test-plan

Chapter 5

Synthesis results

Chapter 6

Conclusions

Chapter 7

Appendices