Cyclic Redundancy Check

A Digital Systems Design project by

Elena Lucherini (482960, eleluche@gmail.com)
and
Sara Falleni (479374, sarafalleni05@gmail.com).

Master of Science in Embedded Computing Systems

Table of contents

1. Introduction

In this paper, a VHDL implementation of a Cyclic Redundancy Check (CRC) is presented, along with its simulation and synthesis.

A CRC is an error-detecting code used in many different systems to detect accidental changes to exchanged messages. The messages entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents. On retrieval, the calculation is repeated on the attached value and, if the new value found is not equal to 0, the data has been presumably corrupted. From this check value comes the name CRC: it is a redundancy (which means, it expands the message without adding information) and the algorithm is based on cyclic codes.

In the generation module, which can be identified as the `sender` module, the input data are represented by the message on M bits and the polynomial on F bits. Each bit of the polynomial is considered as the coefficient of a particular term which is an exponent of x. The rightmost bit is considered as the 0^{th}, the next is the 1^{st}, then the 2^{nd} and so on. For instance, $x^8+x^4+x^2+1$ is represented by the `100010101` bit sequence. The output is the message itself with an (F-1)-bit-long check value, called CRC, attached to it: the CRC is given by the remainder of the polynomial division between the message and the F-bit-long polynomial (both the message and the polynomial are considered as a string of polynomial coefficients).

In the checking module, which can be identified as the `receiver` module, the input data are represented by the message on M bits and the calculated CRC on F-1 bits, along with the divisor polynomial on F bits. The output is the transmitted message itself with an (F-1)-bit-long value which represent whether any error on the message has been detected: this value is generated with a polynomial division, just as in the `sender` module, only the dividend is the (M+F-1)-bit-long message + CRC.

1.1 The algorithm

The algorithm used to build the `sender` module is based on the modulo-2 polynomial division that works as follows.

In order to obtain the remainder we need, the message is right-padded with as many zeroes as the length of the remainder itself (F-1). Given a 14-bit-long input message equal to `11010011101100` and a polynomial x^3+x+1 (CRC-3), here is the first iteration to the computation of the CRC:

```
11010011101100 000 <--- input right padded by 3 bits
1011               <--- divisor (4 bits)
------------------
01100011101100 000 <--- result
```

The first result obtained above is the bitwise XOR of the polynomial divisor with the bits above it; the bits not above the divisor are simply copied directly below. The divisor is then shifted to the right to align with the next 1 in the dividend (had it just been shifted one bit to align with a zero in the dividend, the quotient for that step would have been zero), and the process is repeated until the divisor reaches the right-hand end of the input row. The rest of the calculation is as follows:

```
01100011101100 000 <--- result (as obtained above)
 1011              <--- divisor
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011      <--- divisor aligns with the next 1 in the dividend
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <--- remainder (3 bits).  Division algorithm stops
here as quotient is equal to zero.
```

As for the `receiver` module, the algorithm used is meant to verify the validity of a received message. The algorithm above can be applied, this time with the CRC added instead of zeroes. The remainder should equal zero if there are no detectable errors. For the example above, we have:

```
11010011101100 100 <--- input with check value
1011               <--- divisor
01100011101100 100 <--- result
 1011              <--- divisor ...
00111011101100 100


......


00000000001110 100
          1011
00000000000101 100
           101 1
------------------
                 0  <--- remainder
```

1.2 Possible applications

Not only is CRC simple to implement; it also has the benefit of being particularly well suited for the direction of burst errors, which are contiguous sequences of erroneous data symbols in messages. Being burst errors common transmission errors in many communication channels, CRC is widely used in digital networks, and magnetic and optical storage devices. Numerous varieties of cyclic redundancy checks have been incorporated into many different technical standards, thus it is recommended to select a polynomial according to the application requirements and the expected distribution of message lengths. A few examples are reported below:

- CRC-5-USB: 0x05;
- CRC-6-CDMA-2000-A: 0x27, used in mobile networks;
- CRC-12: 0x80F, used in telecommunication systems;

- CRC-16-DECT: 0x0589, used in cordless telephones.

1.3 Possible architectures

All the possible architectures are based on the division into two modules: `sender` and `receiver`, which are usually placed on two distinct devices connected through an unreliable transmission link. However, for the purpose of this project, a single system containing both modules will be implemented; the output is decided by a multiplexer that works according to the value of a given signal input, `md`.

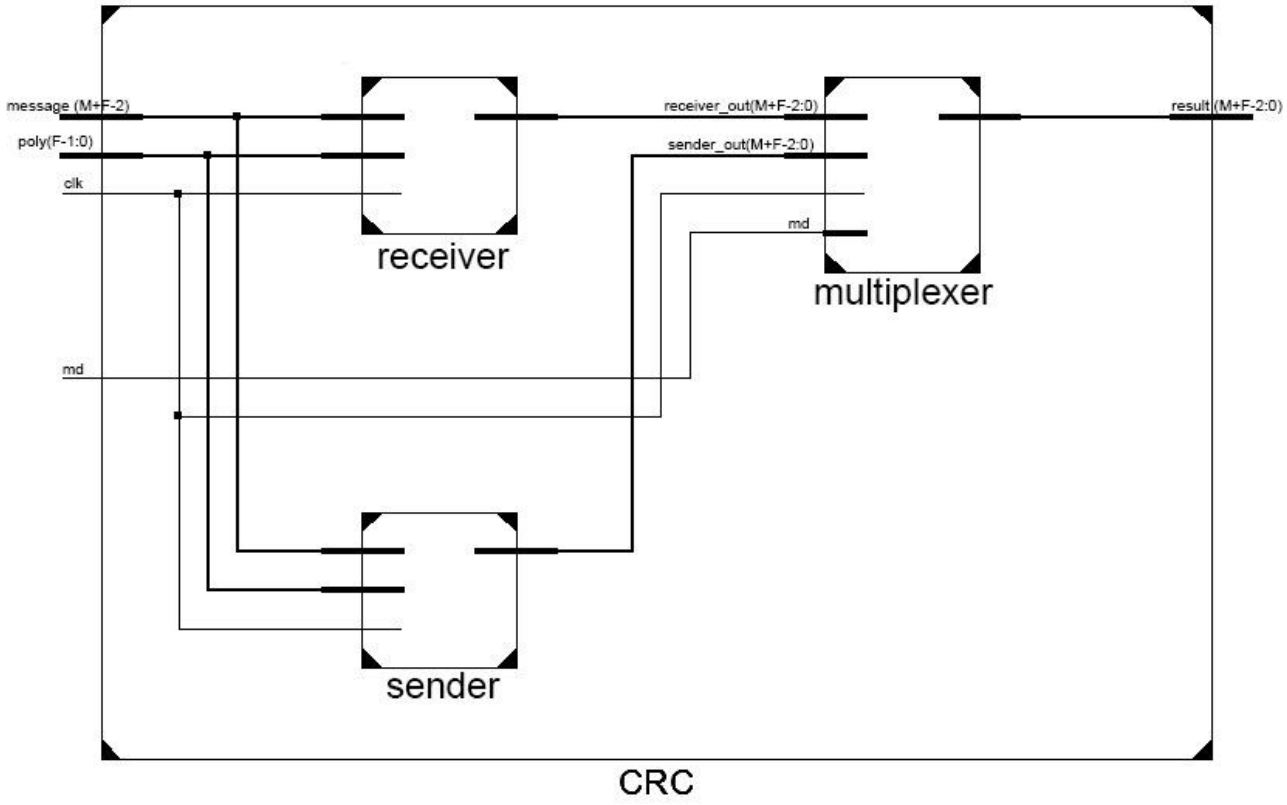
Along with the architecture of the `receiver` module seen in the previous sections, which suggests that the same algorithm used in the `sender` module be applied to the message and CRC input values, another implementation is also possible. The aforementioned algorithm can be applied on the message after padding it with F-1 zeroes, just as in the `sender` module. In order to check if there is any detectable error, the result is then compared to the CRC computed by the `sender`: if it matches, then F-1 zeroes can be generated as a confirmation.

2. Architecture

As introduced in the previous sections, the architecture used to build the `CRC` generator and checking system is divided into two major modules, which represent the `sender` and the `receiver` modules, along with a `multiplexer` that will decide which output to show depending on the value of `md`.

As shown in the picture below, it was decided that both the `sender` and the `receiver` modules receive the polynomial through the `poly` signal input of F bits and the `message` signal input of M+F-1 (where M is the size of the message). As far as `message` is concerned, the following distinctions apply to the `sender` and `receiver` respectively:

1. The `sender` module, which computes the F-1 redundant bits, needs only receive the M bits of the message to transmit. Therefore, the F-1 most significant bits of the `message` input signal are ignored, while the M least significant bits are used to compute the CRC.
2. The `receiver` module needs both the message and the CRC. In particular, the M most significant bits of `message` are dedicated to the transmitted message, while the F-1 following bits are treated as CRC.



Both sender and receiver are subjected to the clock, given by the input signal `clk`.

As for the output, given by the `result` output signal, it is either equal to the output of the `receiver` module (`receiver_out`) or to the output of the `sender` module (`sender_out`), depending on whether the value of `md` is 1 or 0, respectively. In both cases, the M most significant bits of result represent the message, while the remaining F-1 bits vary. The differences between the two different outputs are as follows:

1. If `result` is equal to `sender_out`, then the F-1 least significant bits represent the CRC.
2. If `result` is equal to `receiver_out`, then the F-1 LSBs represent the correctness of the message. If the message transmitted to the receiver contains errors, they are set to all zeroes.

3. Implementation

3.1 Sender

In the VHDL file dedicated to the `sender`, the behavior of the module is described in a process of the clock, where all actions are triggered on its rising edge. The following variables are defined:

- `Mes`: stores the M bits of the message to transmit and only serves as a working variable;
- `V`: it is the M+F–1-bit-long variable which contains the message and the padding of zeroes;
- `U`: stores the divisor polynomial;
- `W`: stores the F most significant bits of `v`;
- `R`: stores the remainder (CRC).

First of all, the M bits of the message to transmit are stored in `mes` and then transferred to `v`, where the F-1 zero-padding is added. The divisor polynomial is stored in `u` and the F MSBs of `v` are stored in `w`. After all the initializations have taken place, the following actions are executed in a 'for' loop:

a. If the MSBs of `w` equals 1, `w` is replaced by `w XOR u`; otherwise, it remains unvaried;
b. `w` is shifted to the left and the MSB is, thus, discarded;
c. The next bit of `v`, starting from the $(F+1)^{th}$, becomes the LSB of `w`;
d. **a**, **b** and **c** are repeated until the end of `v` is reached (there will be a single iteration after the LSB of `v` is added to `w`).

The F-1 MSBs of `w` are then stored in `r` and represent the CRC. They are then appended to the message in the output signal.

3.2 Receiver

The VHDL file dedicated to the `receiver` module works just the same as the sender. The only two differences are that the different composition of the `message` input signal (already described in the previous section) has to be taken into account when storing the M bits of the

actual transmitted message in `mes`, and the fact that the variable `v` is simply equal to the message plus the CRC, so no zero-padding has been applied.

At the end of the process, while in the M MSBs of the output signal the message is stored, the following F-1 bits are equal to the remainder of the division: specifically, they are set to all zeroes if no detectable errors have been found.

3.3 Multiplexer

This module simulates a multiplexer which takes the outputs of `sender` and `receiver` as input, along with `md` and the clock. It is a process of the clock and of `md`, and all the actions are triggered on the rising edge of the clock. The output equals the one of `sender` if the value of `md` is 0; otherwise, the output takes the one of `receiver`. A few other actions are taken, such as storing the old value of md and the output of the multiplexer, in order not to get undefined outputs when the inputs change; thus, the old output is shown while the system is still busy computing the new one.

3.4 CRC

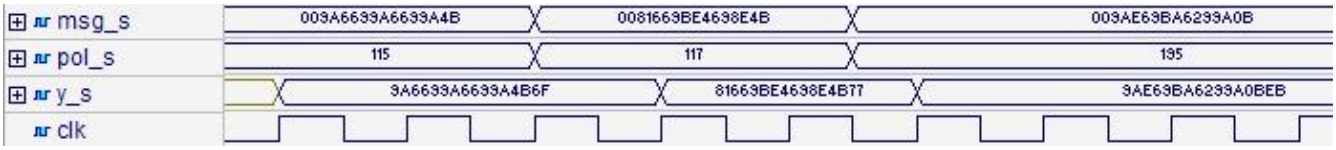
The corresponding VHDL file describes the whole and final system: the input signals of the `CRC` entity are `message`, `poly`, `md` and the clock; the output is `result`. Only the structural architecture is defined: a `sender`, a `receiver` and an `mp` (multiplexer) components are declared and the ports are mapped accordingly, with the aid of a few working signals.

4. Test benches

In order to test the system, two VHDL programs were first built to test the `sender` and `receiver` sub-modules independently; then, two different programs were used on the whole system. Both test benches on the two sub-modules are based off a 56-bit message and a 9-bit polynomial (the redundant part is, then, 8 bits long, which makes the `message` input signal 64 bits long).

4.1 Sender module test

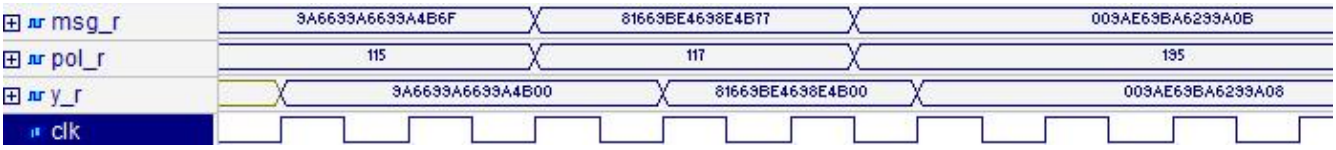
Here follows the Waveform of the test on the `sender` module. As described in the previous sections, the 8 most significant bits of `message` are ignored by the module; we chose to set them to zero. The polynomials used were randomly generated.



All the results were checked with different CRC computational tools and proven to be right.

4.2 Receiver module test

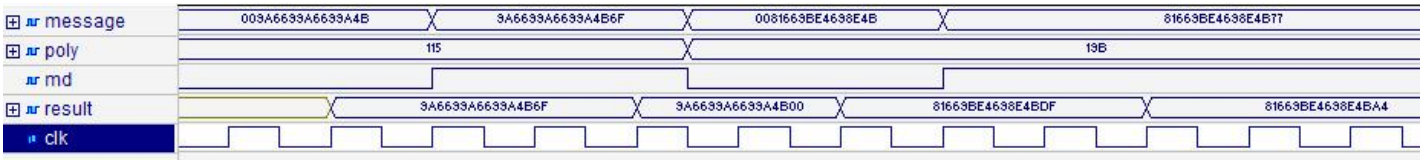
The Waveform of the test on the `receiver` module is the following. As described in the previous sections, the 56 most significant bits of the `y_r` output signal represent the transmitted message, while the final 8 bits are 0x00 if the aforementioned message is error-free.



The correctness of the results can be checked by simply noting that the first two values of `msg_r` are taken from the correct outputs of the previous test (also note that the same polynomial is used), while the last value is deliberately wrong.

4.3 System tests

As for the test benches used for the whole system, two different programs were written in order to show its correctness regardless of the size of the input values. The first test is based off a 56-bit message and a 9-bit polynomial, as seen in the previous tests: two different polynomials are used, the first one of which, $x^8+x^4+x^2+1$, was suggested on the assignment specifications and the second one, $x^8+x^7+x^6+x^4+x^2+1$, is one of the most widely used CRC-8 polynomials. As shown by the value of `md` in the Waveform below, the `sender` and `receiver` modules work alternately.

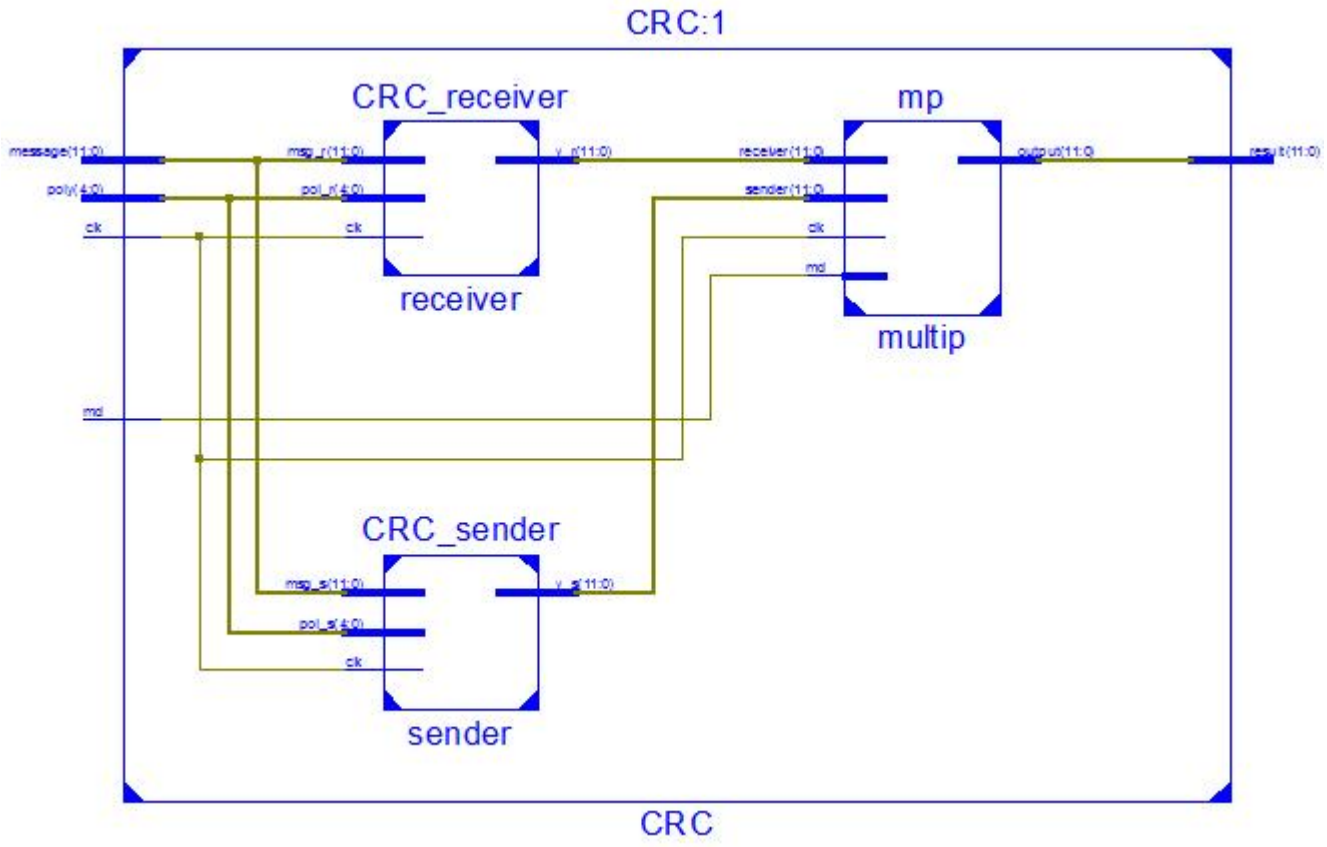


The second test is based off a 46-byte-long message and a 32-bit-long polynomial, specifically the one used in the Ethernet technology. The length of the message is meant to simulate the payload of the packet, assuming that the Ethernet frame is composed only by the payload and the CRC.

5. Synthesis

After verifying the correctness of the VHDL implementation, the system has been synthesized with the Xilinx ISE tool. The following data applies to a Virtex 6 board.

The RTL schematic of a CRC, where M = 8 and F = 5 (CRC-4), is as follows:



The resources used in the synthesis are as follows:

```
Advanced HDL Synthesis Report

Macro Statistics
# Registers                                          : 37
 Flip-Flops                                          : 37
# Multiplexers                                       : 17
 12-bit 2-to-1 multiplexer                           : 1
 5-bit 2-to-1 multiplexer                            : 16
# Xors                                               : 17
 1-bit xor2                                          : 1
 5-bit xor2                                          : 16
```

5.1 Timing analysis and worst path

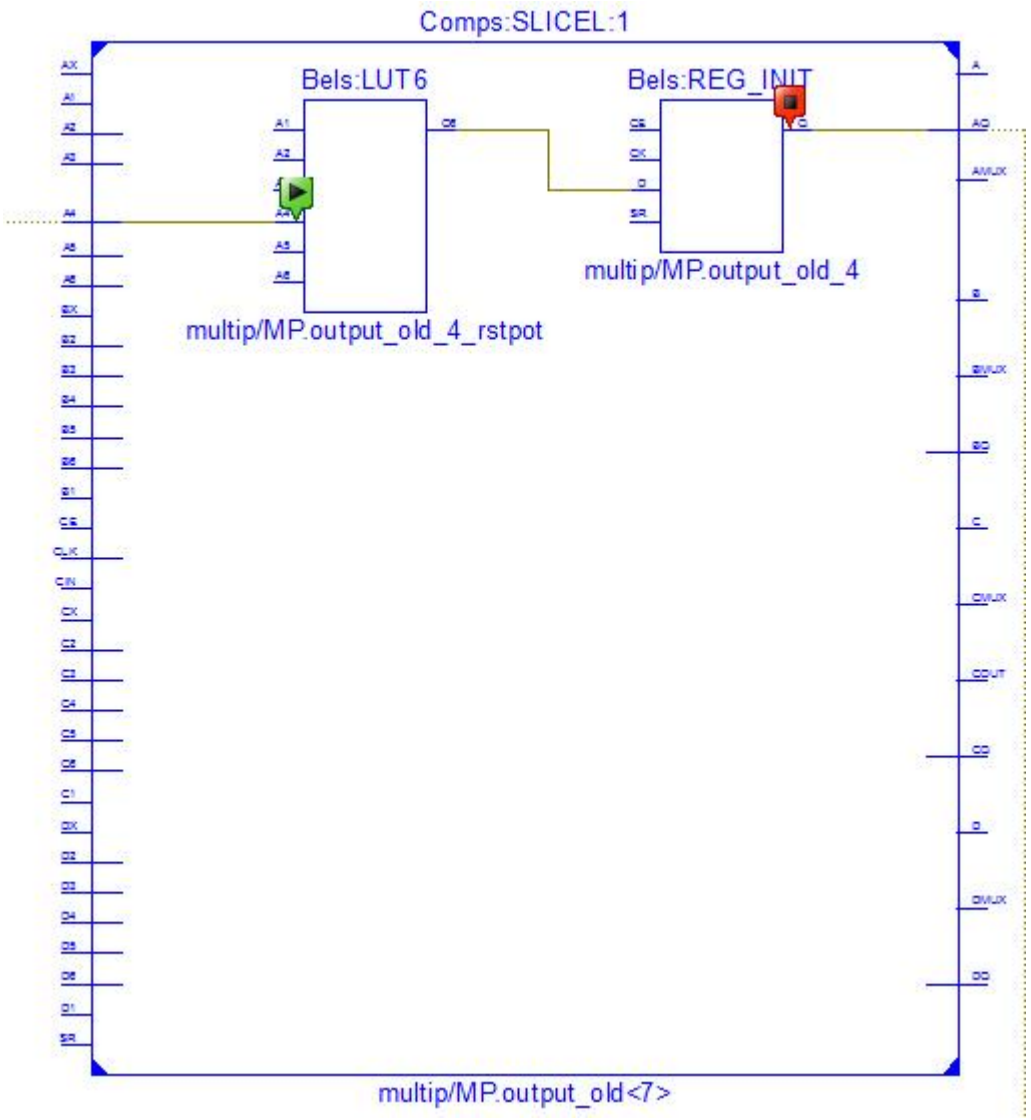
The timing constraints were defined so that the clock have period of 20 ns. The constraint was easily met and, as shown in the report below, it turned out to be quite loose, with 593.824 MHz being the maximum frequency allowed:

```
Timing summary:
 ---------------

 Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

 Constraints cover 48 paths, 0 nets, and 57 connections

 Design statistics:
    Minimum period:   1.684ns{1}   (Maximum frequency: 593.824MHz)
    Maximum path delay from/to any node:   1.684ns
```

The worst path is, then, the one that produces the 1.684 ns delay, and it is explained in details in the following:

```
   Maximum Data Path at Slow Process Corner: multip/MP.output_old_4 to
multip/MP.output_old_4

     Location             Delay type         Delay(ns)  Physical Resource
                                                        Logical Resource(s)

     -------------------------------------------------  -------------------

     SLICE_X0Y61.AQ       Tcko              0.381       multip/MP.output_old<7>
                                                        multip/MP.output_old_4
     SLICE_X0Y61.A5       net (fanout=2)    1.238       multip/MP.output_old<4>
     SLICE_X0Y61.CLK      Tas               0.030       multip/MP.output_old<7>

multip/MP.output_old_4_rstpot
                                                        multip/MP.output_old_4

     -------------------------------------------------  -------------------------

     Total                                    1.649ns (0.411ns logic, 1.238ns route)
                                                      (24.9% logic, 75.1% route)
```

Its view in Technology Viewer is the following:



5.2 Power distribution

Here follows the Power Distribution report generated through the XPower Analyzer tool:

```
2.  Summary
2.1.  On-Chip Power Summary
------------------------------------------------------------------------------
|                          On-Chip Power Summary                             |
------------------------------------------------------------------------------
|        On-Chip        | Power (mW) |  Used  | Available | Utilization (%) |
------------------------------------------------------------------------------
| Clocks                |       1.66 |      1 |    ---    |       ---       |
| Logic                 |       0.21 |     44 |     46560 |               0 |
| Signals               |       0.26 |     78 |    ---    |       ---       |
| IOs                   |      15.82 |     30 |       240 |              13 |
| Static Power          |    1293.15 |        |           |                 |
```

Total	1311.10			

2.2. Thermal Summary

Thermal Summary	
Effective TJA (C/W)	2.7
Max Ambient (C)	81.4
Junction Temp (C)	53.6

2.3. Power Supply Summary

Power Supply Summary			
	Total	Dynamic	Static Power
Supply Power (mW)	1311.10	17.95	1293.15

Power Supply Currents

Supply Source	Supply Voltage	Total Current (mA)	Dynamic Current (mA)	Quiescent Current (mA)
Vccint	1.000	622.15	2.65	619.50
Vccaux	2.500	45.32	0.32	45.00
Vcco25	2.500	6.80	5.80	1.00
MGTAVcc	1.000	303.31	0.00	303.31
MGTAVtt	1.200	212.78	0.00	212.78

6. Conclusions

In this paper, the process of CRC generation and checking has been discussed in detail. The methods applied to detect an error during transmission has been shown in VHDL, along with the synthesis on the Xilinx ISE 14.7 tool and code generation for a Virtex 6 FPGA. Considerations on timing constraints and power distribution have also been included. It is important to note that CRC has two main limitations: first of all, being it only a detecting method, CRC does not offer a method to correct the encountered errors; secondly, the divisor polynomial should be chosen carefully and has to be a multiple of (x + 1), otherwise the choice may result into wrong calculation of the CRC itself.