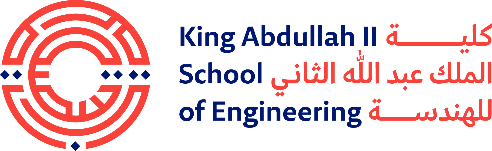
Princess Sumaya University for Technology

King Abdullah II Faculty of Engineering



***Error Detection (CRC)   
Final Report***

|  |  |
| --- | --- |
| *Authors:* |  |
| *Lunar Salameh*  *Dima Hussein*  *Sara Alghzawi* | *20190983*  *20190581*  *20190361* |

Table of Contents

[1 Abstract 2](#_Toc99921828)

[2 Introduction 2](#_Toc99921832)

[3 Proposed Design 3](#_Toc99921833)

[3.1 Design Description 3](#_Toc99921834)

[3.2 Block Diagram 3](#_Toc99921834)

[3.3 Design Analysis 3](#_Toc99921835)

[3.4 Test Runs and Discussions 7](#_Toc99921836)

[3.5 Performance Analysis 7](#_Toc99921836)

[4 Coclusion 8](#_Toc99921837)

**1 Abstract**

In this project, we present a fast CRC (Cyclic Redundancy Check) calculation algorithm for an 8-bit input data. It works by dividing the input into chunks of 4 blocks, each has a fixed size, and calculating each block’s CRC in parallel using lookup tables, which have precomputed CRC values stored in them. The results are then combined using XOR operations. The short pipeline latency of this algorithm enables a fast clock frequency. We built a Verilog implementation of our pipelined CRC. The simulation results show that our approach is faster than other CRC calculation methods.

**1 Introduction**

A CRC (Cyclic Redundancy Check) is a popular method to calculate errors in data transmissions using polynomial divisions. In order to calculate a CRC, the sender treats the binary input as a binary polynomial and performs the modulo-2 division of the polynomial, the remainder of this division is the CRC value. It is then attached to the original input and sent to the receiver. The receiver also performs the modulo-2 division on the received data and the generator polynomial. The results of the two CRC’s are then compared in order to detect an error.

In order to support high throughput CRC at a reasonable frequency, processing multiple bits in parallel and pipelining the processing path are desirable. In addition to that, this algorithm adopts multiple small lookup tables instead of a single large one in order to keep the overall area small.

**3 Proposed Design**

3.1 Design Description

We had designed six stages for this algorithm, each stage was implemented through different independent modules, and tested.

In the first module, which implemented *stage\_1*, the input message which is 8 bits is divided into 4 blocks, each of 2 bits. This is done by concatenating a specific number of zeros, according to each block, from the left and the right, for the size to stay the same and for the bits to remain in their right position.

For the second stage, we had to implement two modules for it. Module *LookUp\_Table*, which has the values of 256 CRC values stored in the memory. While module *stage\_2* has the outputs from *stage\_1* ready in order to calculate their CRC’s, therefore, module *LookUp\_Table* is called four times in order to get the CRC for the four blocks.

The third stage was implemented through module *stage\_3*. This only consisted of registers that forwarded the outputs of *stage\_2* in order to cause a delay for the CRC’s to be calculated.

The fourth stage was implemented through module stage\_4. This stage only consisted of one operation, which is the XOR, where the outputs of stage\_3 which are the CRC values that have been calculated using the lookup tables are XORed together in order to combine them back together, as the result of the XOR operation will be our final CRC value of our transmitted data.

The fifth stage was implemented through module *stage\_5* which had the output CRC from *stage\_4* appended with the input message that was forwarded through the previous stages.

The sixth stage was implemented through module *Detection*. Its purpose is to detect whether any error had occurred in the transmitted message or not. Which we found the code on the internet.

In the *main* module, the input was stored in the memory as a form of 2 bits, as well as for the polynomial in a memory of 32-bits, in order to be forwarded through the stages. This module is mainly for the other modules to be called and for the memory to be used. As we also did our implementation regarding the program reading on the positive edge of the clock and writing on the negative edge of the clock.

3.2 Block Diagram

Diagram

Description automatically generated

3.3 Design Analysis

Our algorithm first consisted of seven stages, however, we first had to adjust that during our Verilog code implementation and add two extra stages, stage 8 and stage 9. Since the first 7 stages only calculate the CRC of the input message, without detecting if there were any errors with the transmitted message. Therefore, in stage 8, the output CRC from stage 7 is appended with the input message which was forwarded through the stages. In stage 9, the output from stage 8 is XORed with the input polynomial, which was forwarded through the stages, in order to detect if any errors had occurred, if the output *Detect* equaled to zero, then the transmitted message had no errors, otherwise, the transmitted message has an error. However, we later decided to adjust our code and the block diagram. We first removed the fifth block, since the input data will only need four blocks to be distributed on. Then we removed the MUX in stage 4 as a result of removing block 5 and the lookup table followed by it in stage 5. Therefore, stage 5 was completely deleted. As a result, stage 6 was also removed since it consisted of an XOR operation between the MUX result and the result of the forwarded first XOR operation. Stage 7 was then replaced by stage 8 which was the module that was responsible for the appending operation, which has now become stage 5, followed by stage 9 which is now stage 6.

We also faced a structural hazard while running the code, this was due to the consecutive stages using the same data from the memory at the same time. In addition to that, we tried to implement our own *Detection* module, however, it showed wrong results even though the implementation concept was correct, therefore we found a ready code on the internet that gave correct results and used it in our implementation, which we got from this website: https://bues.ch/cms/hacking/crcgen.

3.4 Test Runs and Discussions

Stage 1:

## A screenshot of a computer Description automatically generated

The following screenshot is showing the output of stage 1, which is the result of splitting the data into blocks and storing them in the latches, while forwarding the polynomial and the input as a full block.

Stage 2 &3:

## A screenshot of a computer Description automatically generated

The following screenshot is showing the output of stage 2 and 3, which the CRC values have been calculated in stage 2 for each block and forwarded to stage 3 with the input and polynomial.

Stage 4:

## A screenshot of a computer Description automatically generated

The following screenshot is showing the output of stage 4, where the 4 input CRC’s have been XORed together, and the input and polynomial have been forwarded.

Stage 5:

## A screenshot of a computer Description automatically generated

The following screenshot is showing the output of stage 5, where the input had been appended with the CRC.

Stage 6:

## Graphical user interface, text, application Description automatically generated

The following screenshot is showing the output of stage 6, where the appended data have been forwarded with the polynomial and tested to detect if any errors had occurred, which in this case, the output shows that there were no errors in the transmitted data.

## Graphical user interface, text, application Description automatically generated

The following screenshot is showing the output of stage 6, where the appended data have been forwarded with the polynomial and tested to detect if any errors had occurred, which in this case, the output shows that the data have been affected by some noise which caused an error in the transmitted data.

3.5 Performance Analysis

As shown in the screenshots, the time taken in the pipelined algorithm took less time than the usual C-code, where the C-code took 0.000385 seconds, but the pipelined implantation took about 2000 nano seconds.

A screenshot of a computer

Description automatically generated

*C-code implementation*

Graphical user interface

Description automatically generated

*Verilog Pipelined Implementation*

\*\*Tracing The stages of pipeline …

Graphical user interface, text, application

Description automatically generated

Input Three is in

Stage2 of In1

Stage1 of In2

Stage1 of In1

Stage1 of In1

Second Input Is In

First Input is In

A picture containing graphical user interface

Description automatically generated

Stage4 of In1

Stage3 of In2

Stage2 of In2

Stage1 of In3

Stage1 of In3

Stage2 of In2

Stage3 of In1

Stage3 of In1

Stage2 of In2

Stage1 of In2

Text

Description automatically generated

Stage4 of In2

Stage1 of In4

Stage2 of In3

Stage3 of In3

Stage5 of In1

Stage4 of In2

Stage3 of In2

Stage2 of In3

Stage1 of In4

Input four Is In

Stage4 of In1

Stage1 of In3

Stage2 of In3

Stage3 of In2

Text

Description automatically generated

Stage6 of In2

Stage5 of In2

Stage4 of In2

Stage2 of In4

Stage1 of In4

Stage3 of In3

**4 Conclusion**

We learned how to deal with pipelined architectures and how to implement them using Verilog code. We also proved how pipelined architectures are much faster than single cycled ones and how the throughput had also improved. In addition to that, we also learned how to deal with hazards if any had occurred.