FPGA BASED SYSTEM DESIGN

(2EC202)

PROJECT REPORT

Department of Electronics and Communication Engineering

Submitted by :

Prem Rajpurohit (22BEC099)
Hitarth Chudasama (22BEC019)

Abstract

This paper presents the design and implementation of a cyclic redundancy check (CRC) module using Verilog hardware description language (HDL). CRC is a widely used error detection technique in digital communication systems to ensure data integrity. The proposed CRC module is designed to perform polynomial division using shift registers and XOR gates to generate a checksum for a given input data stream. The Verilog code is structured to be modular and efficient, allowing for easy integration into larger digital systems. Simulation results demonstrate the functionality and correctness of the CRC module across various input data patterns and polynomial configurations. The implemented CRC module provides a reliable and efficient solution for error detection in digital communication systems.

The implementation involves defining the CRC polynomial, initializing shift registers, and performing polynomial division on the input data stream. The Verilog code iteratively processes each bit of the input data, updating the shift registers and performing XOR operations according to the CRC polynomial. The resulting CRC checksum is then appended to the transmitted data for error detection at the receiver end.

The Verilog implementation provides flexibility in terms of CRC polynomial selection and data width, allowing adaptation to various communication standards and system requirements. Additionally,

the efficiency of the implementation is optimized for FPGA and ASIC platforms, ensuring real-time error detection with minimal resource utilization.

Keywords

1. Verilog
2. CRC
3. Cyclic Redundancy Check
4. Division
5. XOR Gate
6. Bitwise Operations

Literature Survey / State of the art Technology

1.CRC Algorithm Selection:

Choose an appropriate CRC algorithm based on the specific requirements of your application (e.g., CRC-32, CRC-16, etc.). The choice depends on factors such as desired error-detection capabilities, data width, and implementation complexity.

2.Verilog Implementation:

Module Design: Create a Verilog module for the CRC calculation. This module should include inputs for the data to be checked, configuration parameters for the CRC polynomial, initial CRC value, and outputs indicating the validity of the CRC.

CRC Polynomial: Define the CRC polynomial within the Verilog module. This can be represented as a polynomial equation or as a precomputed lookup table, depending on the chosen algorithm.

Shift Register: Implement a shift register to perform the polynomial division operation. This register should be wide enough to accommodate the input data and CRC polynomial.

3.Verification and Testing:

Simulation: Use Verilog simulation tools to verify the correctness of the CRC implementation and ensure that it meets the required performance specifications.

Formal Verification: Employ formal verification techniques to mathematically prove the correctness of the CRC module design, especially for safety-critical applications.

4.Synthesis:

Perform synthesis using a Verilog synthesis tool to map the RTL design onto the target FPGA or ASIC technology. Optimize synthesis settings to achieve the desired balance between performance, area, and power consumption.

Drawbacks

1.Complexity:

Verilog code for CRC implementation can be complex, especially for larger CRC polynomials or when optimizing for speed or resource usage. This complexity can lead to longer development times and increased chances of errors in the code.

2.Resource Usage:

CRC implementation in Verilog may require significant hardware resources, such as LUTs (Look-Up Tables), flip-flops, and DSP (Digital Signal Processing) blocks, depending on the CRC polynomial size and the desired performance. This can limit the scalability of the design and increase the cost of hardware implementation.

3.Speed Limitations:

Achieving high-speed CRC calculation in Verilog may be challenging due to limitations in clock frequency and pipeline depth. Balancing between speed and resource usage is crucial, and achieving both high speed and low resource utilization can be difficult.

4.Verification Challenge:

Verifying the correctness of a CRC implementation in Verilog can be complex, especially for designs with large input data widths or multiple CRC polynomials. Comprehensive testbenches are required to validate the functionality across various input patterns and corner cases, which adds to the development effort.

5.Portability:

Verilog implementations of CRC may not be easily portable across different FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) platforms due to differences in vendor-specific synthesis tools, timing constraints, and resource utilization. This lack of portability can hinder the reuse of the CRC module in different projects or platforms.

Methodology

1.Understand CRC Algorithm:

First, understand the CRC algorithm you want to implement. CRC algorithms vary based on polynomial selection, initial value, and final XOR value. Decide on the parameters for your CRC implementation.

2.Choose CRC Parameters:

Select the parameters for your CRC implementation, including polynomial, initial value, final XOR value, and data width.

3.Design CRC Module:

Create a Verilog module for CRC calculation. This module will take input data and calculate the CRC checksum.

4.CRC Polynomial Calculation:

Implement the polynomial division algorithm in Verilog to calculate the CRC checksum. This involves shifting the data and performing XOR operations based on the polynomial.

5.Validation:

Validate the CRC implementation in the context of your overall system design to ensure it meets the requirements and performs reliably.

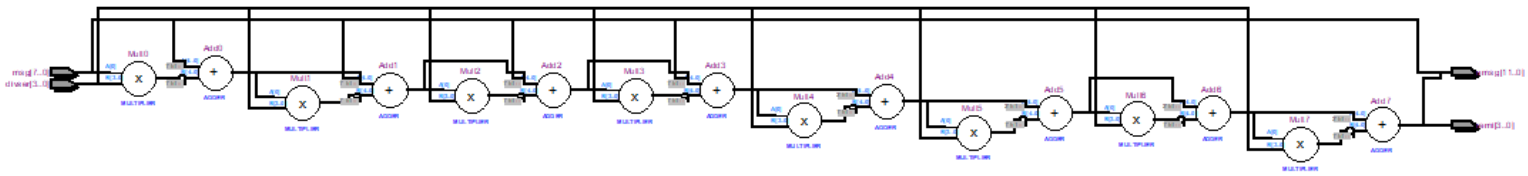
6.Documentation:

Document your Verilog code, including module interfaces, parameters, and functionality, to aid future maintenance and understanding.

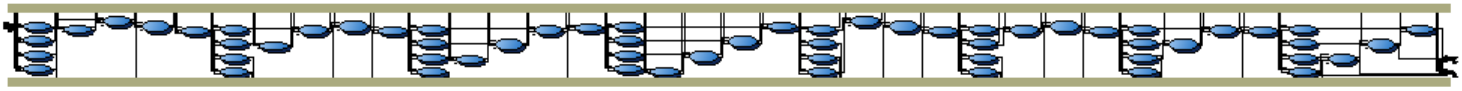
Verilog Code

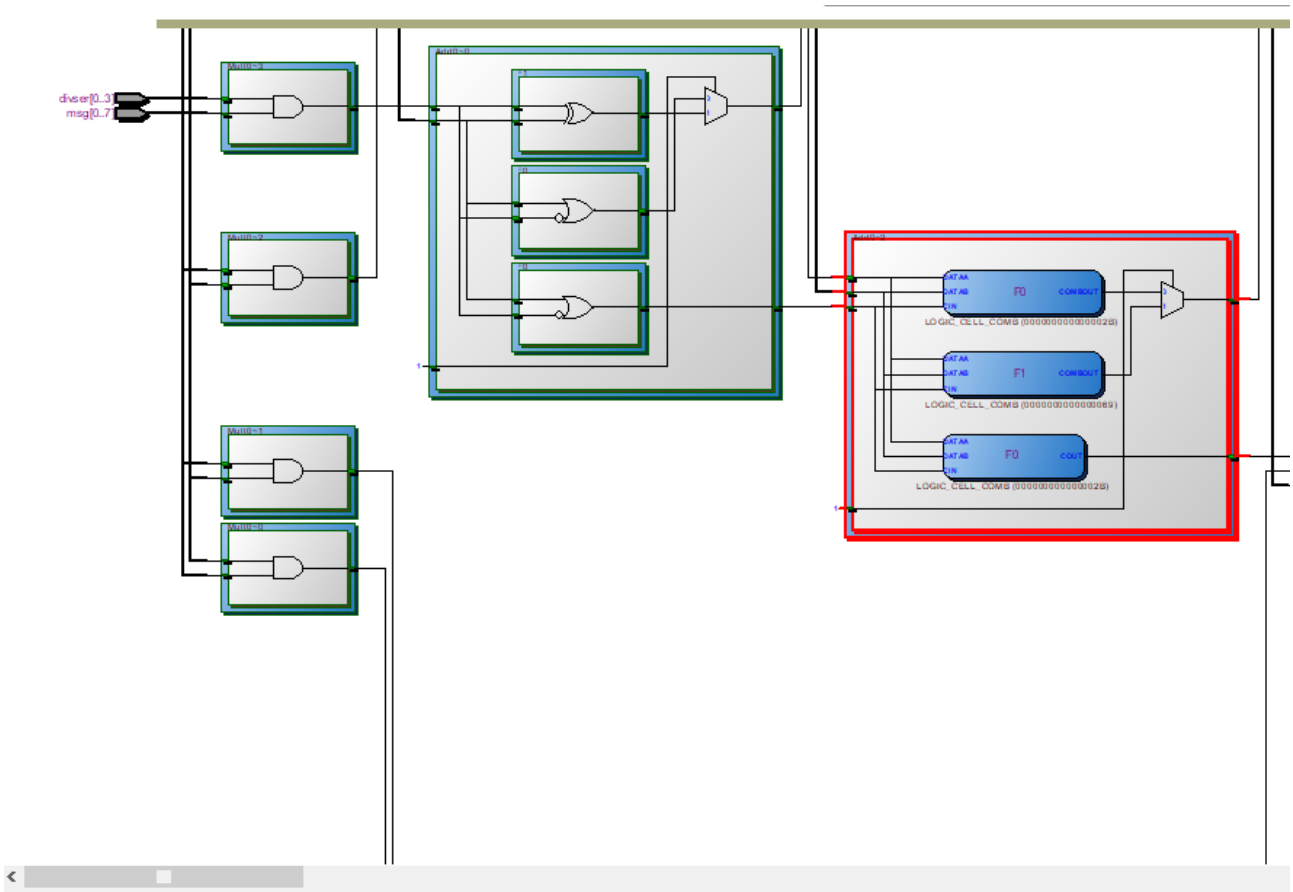
```
module crc(input [7:0] msg,input [3:0] divser,output reg [3:0] remi);
reg [11:0] msgd;
integer i;
reg k;
always@(msg)begin
msgd[3:0]=0000;
msgd[11:4]=msg[7:0];
remi[3:0]=msgd[11:8];
k=remi[3];
remi=remi<<1;
remi[0]=msgd[7];
remi=remi-(k*divser);
for(i=6;i>=0;i=i-1)begin
k=remi[3];
remi=remi<<1;
remi[0]=msgd[i];
remi=remi-(k*divser);
end
end
endmodule
```

RTL View

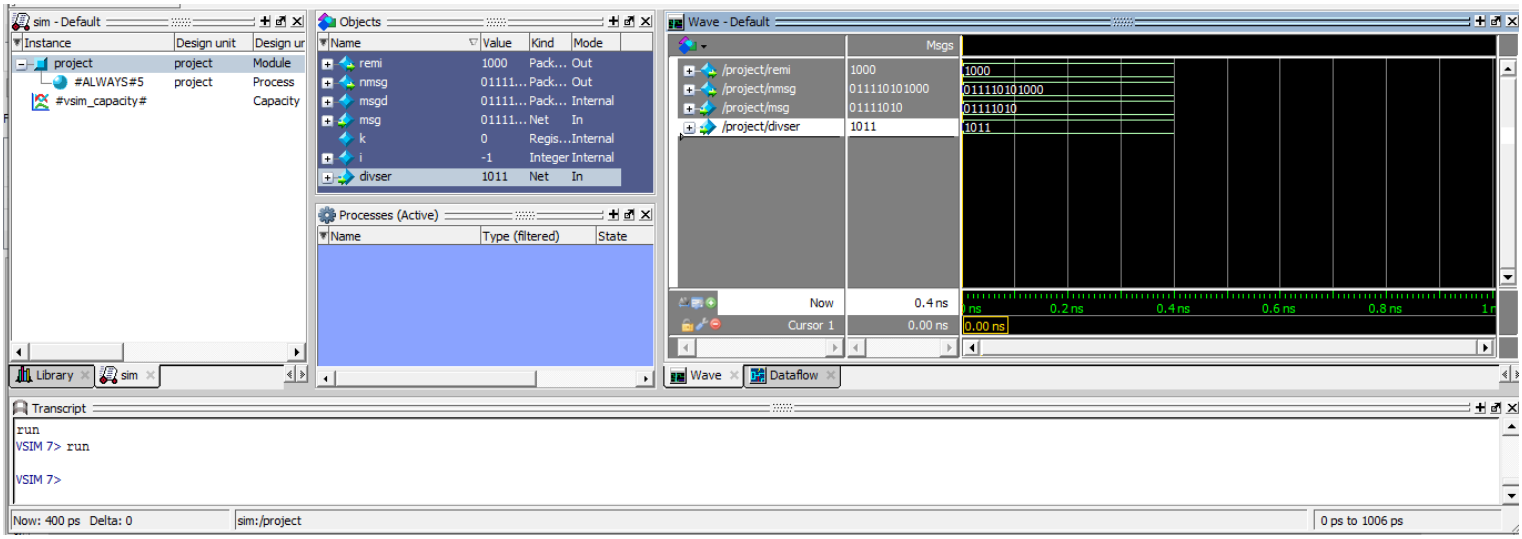


TTL View

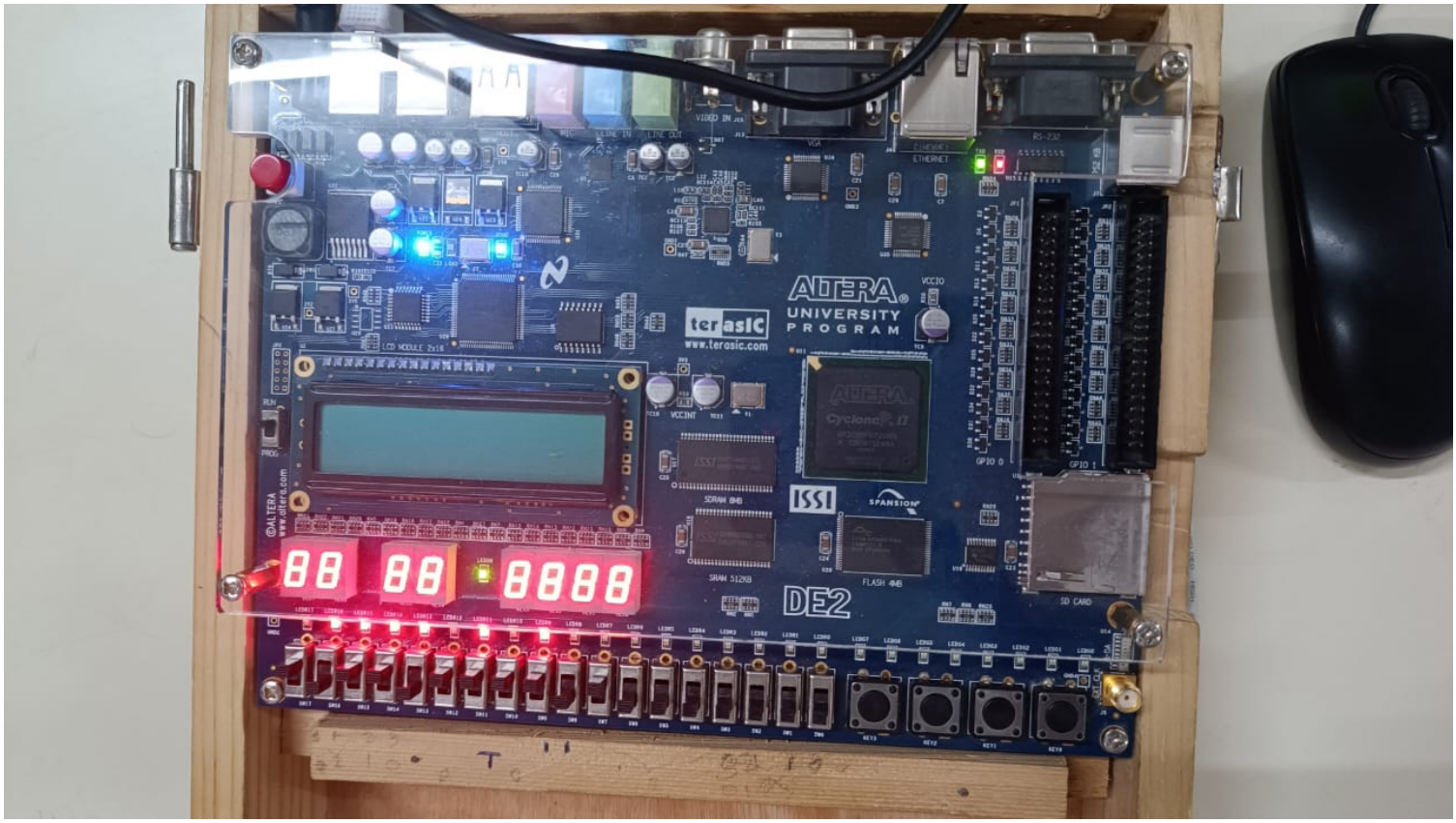




Simulation


sim - Default
Instance
Design unit
Design ur
project
project
Module
#ALWAYS#5
project
Process
#vsim_capacity#
Capacity
Objects
Name
Value
Kind
Mode
remi
1000
Pack...
Out
nmsg
01111...
Pack...
Out
msgd
01111...
Pack...
Internal
msg
01111...
Net
In
k
0
Regis...
Internal
i
-1
Integer
Internal
divser
1011
Net
In
Processes (Active)
Name
Type (filtered)
State
Wave - Default
Msgs
/project/remi
/project/nmsg
/project/msg
/project/divser
1000
01111010 1000
01111010
1011
Now
0.4 ns
Cursor 1
0.00 ns
0.2 ns
0.4 ns
0.6 ns
0.8 ns
Library
sim
Wave
Dataflow
Transcript
run
VSIM 7> run
VSIM 7>
Now: 400 ps Delta: 0
sim:/project
0 ps to 1006 ps

FPGA Implementation



REFERENCES

• Github

• A practical parallel CRC Generation Method by Evgeni Stavinov

• Comprehensive study on 8/12 - bit Cyclic Redundancy Check