2020 年新工科联盟-Xilinx 暑期学校团队项目设计文档

设计文稿提交格式

(Project Paper Submission Template)

设计作品名称	CRC 校验
Github 链接	https://github.com/yumi-del/proof-of-CRC-algorithm#proof-of-crc-algorithm

第一部分

设计概述 /Design Introduction

设计目的：

本项目的设计目的是通过 RTL 仿真验证 CRC 算法的加密和校验原理,实现对数据的加密和校验功能。通过学习和编写 CRC 算法在实验过程中提高了对 verilog 语言、对 vivado 操作平台等的熟悉。

技术背景：

CRC 即循环冗余校验码是数据通信领域中最常用的一种查错校验码,其特征是信息字段和校验字段的长度可以任意选定。循环冗余检查（CRC）是一种数据传输检错功能，对数据进行多项式计算，并将得到的结果附在帧的后面，接收设备也执行类似的算法，以保证数据传输的正确性和完整性。

校验原理：

CRC 的根本思想就是先在要发送的帧后面附加一个数，生成一个新帧发送给接收端，其满足所生成的新帧能与发送端和接收端共同选定的某个特定数整除（“模 2 除法”）。到达接收端后，再把接收到的新帧除以这个选定的除数。因为在发送端发送数据帧之前就已通过附加一个数，做了“去余”处理，所以结果应该是没有余数。如果有余数，则表明该帧在传输过程中出现了差错。在数据后附加的这个数就是我们要计算的 CRC 循环冗余校验码。

CRC 的版本也很多，我的设计研究是基于 16-bits 的欧洲标准 CRC-CCITT(0x1021)，即数据位 16 位，冗余校验码也为 16 位，使用的多项式$\mathrm{G(x)} = x^{16} + x^{12} + x^5 + 1.$

计算 CRC 的步骤：

①选择合适的除数，这里的多项式为$\mathrm{x}^{16} + \mathrm{x}^{12} + \mathrm{x}^5 + 1$；

②看选定除数的二进制位数，然后再要发送的数据帧后面加上（除数位数-1）位的 0，即 16 个 0，然后用新生成的帧以模 2 除法的方式除上面的除数，得到的余数就是该帧的 CRC 校验码。注意，余数的位数一定只比除数位数少一位，也就是 CRC 校验码位数比除数位数少一位，如果前面位是 0 也不能省略。

③将计算出来的 CRC 校验码附加在原数据帧后面，构建成一个新的数据帧进行发送；最后接收端在以模 2 除法方式除以前面选择的除数，如果没有余数，则说明数据帧在传输的过程中没有出错。

应用领域：

在诸多检错手段中，CRC 是最著名的一种，其特点是:检错能力强，开销小，易于用编码器及检测电路实现。从其**检错能力**来看，它所不能发现的错误的几率仅为 0.0047%以下。从**性能上和开销上**考虑，均远远优于奇偶校验及算术和校验等方式。因而，在数据存储和数据通讯领域，CRC 无处不在：著名的通讯协议 X.25 的 FCS(帧检错序列)采用的是 CRC-CCITT，WinRAR、NERO、ARJ、LHA 等压缩工具软件采用的是 CRC32，磁盘驱动器的读写采用了 CRC16，通用的图像存储格式 GIF、TIFF 等也都用 CRC 作为检错手段。

第二部分

系统组成及功能说明 /System Construction & Function Description

功能说明：

我所设计的是一个采用欧洲标准 CRC-CCITT(0x1021)的 32 位并行输出的 CRC 加密编码和校验功能的项目。当并行输入 16 位原数据，在下一个周期上升沿时即可并行输出 32 位 CRC 加密数据，完成加密功能；当并行接收 32 位传输数据，在一个周期时间后可通过 LED 显示出传输数据是否有误，完成校验功能。

系统组成：

系统由核心模块 CRC16 和具体实现功能的加密模块 Encode 以及校验模块 Proof 组成。核心模块 CRC16 负责在给定 16 位输入数据时计算出相应的正确的 16 位 CRC 冗余校验码；加密模块 Encode 和校验模块 Proof 分别实现上述功能描述中加密和校验的功能。以下为各模块的具体介绍：

- **CRC 循环冗余码计算模块**

CRC 循环冗余码计算模块的具体实现部分如下图 2-2 所示。moduleCRC16 的功能是计算出输入的 16 位数据的 16 位冗余校验码，使用多项式$G(x) = x^{16} + x^{12} + x^5 + 1$。

首先定义端口，输入有时钟 i_clk,同步复位 i_rst_n，输入数据有效位 i_din_vaild，和 16 位的数据 i_din;输出有 16 位的校验码 o_dout 和输出有效位 o_dout_vaild。当输入数据有效位为 1 说明数据已输入可以开始计算；当输出有效位为 1 说明已经完成了 CRC 冗余校验码的计算，此时输出端口的输出值正是 CRC 校验码；若输出有效位为 0，那么这时的输出无效。

之后定义了几个变量，r_dout 储存计算所得的 CRC 校验码，是与输出 o_dout 相连的；r_dout_vaild 暂存输出有效位，是与 o_dout_vaild 相连的；数组 d 是为了方便计算设置的，存储数据与输入数据相同；数组 c 存入的是上一次计算所得的 CRC 冗余校验码。

CRC 冗余校验码的具体计算是利用 LFSR（线性反馈移位寄存器）计算 CRC 的模型编写的，由于多项式$G(x) = x^{16} + x^{12} + x^5 + 1$，模型如下图 2-1 所示。由于不断输入，在实际应用中 CRC 校验码的值与上一次寄存器里存放的 CRC 校验码有关系，因此我们在计算 CRC 校验码时候，不仅与多项式选择有关，还与寄存器初值有关。在并行输出输入的情况下，可以将计算逻辑具体编写如图 1-3 所示。

当 i_rst_n=0 时，重置寄存器，CRC 初始值为 0xffff。当 i_rst_n=1，输入数据有效位 i_din_vaild=1 时，开始计算 CRC。在输入数据后的下一个上升沿时 r_dout_vaild 和 o_dout_vaild 变为 1，表示输出有效，同时 o_dout 会输出相应的结果，因此计算的时间仅需 1 个周期。

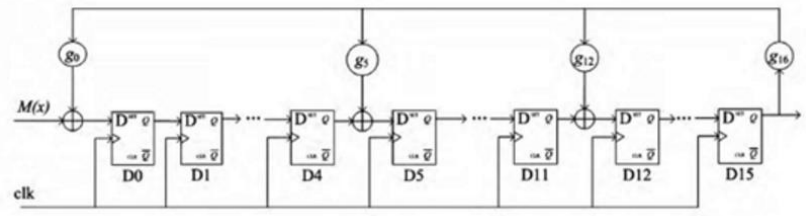

图 2-1 LFSR 计算 CRC 模型

```
module CRC16(
    input wire i_clk,         //时钟;
    input wire i_rst_n,       //同步复位;
    input wire i_din_valid,   //输入数据有效;
    input wire [15:0]i_din,   //输入数据;
    output wire o_dout_valid ,//输出CRC值有效;
    output wire [15:0]o_dout  //输出CRC;
    );
reg [15:0] r_dout;
wire [15:0] d;
wire [15:0] c;
assign d = i_din;
assign c = r_dout;
always @(posedge i_clk) begin
    if (~i_rst_n)
        r_dout <= 16'hffff; //初始值为ffff;
    else if (i_din_valid)
    begin //计算逻辑;
        r_dout[0]  = d[12] ^ d[11] ^ d[8] ^ d[4] ^ d[0] ^ c[0] ^ c[4] ^ c[8] ^ c[11] ^ c[12];
        r_dout[1]  = d[13] ^ d[12] ^ d[9] ^ d[5] ^ d[1] ^ c[1] ^ c[5] ^ c[9] ^ c[12] ^ c[13];
        r_dout[2]  = d[14] ^ d[13] ^ d[10] ^ d[6] ^ d[2] ^ c[2] ^ c[6] ^ c[10] ^ c[13] ^ c[14];
        r_dout[3]  = d[15] ^ d[14] ^ d[11] ^ d[7] ^ d[3] ^ c[3] ^ c[7] ^ c[11] ^ c[14] ^ c[15];
        r_dout[4]  = d[15] ^ d[12] ^ d[8] ^ d[4] ^ c[4] ^ c[8] ^ c[12] ^ c[15];
        r_dout[5]  = d[13] ^ d[12] ^ d[11] ^ d[9] ^ d[8] ^ d[5] ^ d[4] ^ d[0] ^ c[0] ^ c[4] ^ c[5] ^ c[8] ^ c[9] ^ c[11] ^ c[12] ^ c[13];
        r_dout[6]  = d[14] ^ d[13] ^ d[12] ^ d[10] ^ d[9] ^ d[6] ^ d[5] ^ d[1] ^ c[1] ^ c[5] ^ c[6] ^ c[9] ^ c[10] ^ c[12] ^ c[13] ^ c[14];
        r_dout[7]  = d[15] ^ d[14] ^ d[13] ^ d[11] ^ d[10] ^ d[7] ^ d[6] ^ d[2] ^ c[2] ^ c[6] ^ c[7] ^ c[10] ^ c[11] ^ c[13] ^ c[14] ^ c[15];
        r_dout[8]  = d[15] ^ d[14] ^ d[12] ^ d[11] ^ d[8] ^ d[7] ^ d[3] ^ c[3] ^ c[7] ^ c[8] ^ c[11] ^ c[12] ^ c[14] ^ c[15];
        r_dout[9]  = d[15] ^ d[13] ^ d[12] ^ d[9] ^ d[8] ^ d[4] ^ c[4] ^ c[8] ^ c[9] ^ c[12] ^ c[13] ^ c[15];
        r_dout[10] = d[14] ^ d[13] ^ d[10] ^ d[9] ^ d[5] ^ c[5] ^ c[9] ^ c[10] ^ c[13] ^ c[14];
        r_dout[11] = d[15] ^ d[14] ^ d[11] ^ d[10] ^ d[6] ^ c[6] ^ c[10] ^ c[11] ^ c[14] ^ c[15];
        r_dout[12] = d[15] ^ d[8] ^ d[7] ^ d[4] ^ d[0] ^ c[0] ^ c[4] ^ c[7] ^ c[8] ^ c[15];
        r_dout[13] = d[9] ^ d[8] ^ d[5] ^ d[1] ^ c[1] ^ c[5] ^ c[8] ^ c[9];
        r_dout[14] = d[10] ^ d[9] ^ d[6] ^ d[2] ^ c[2] ^ c[6] ^ c[9] ^ c[10];
        r_dout[15] = d[11] ^ d[10] ^ d[7] ^ d[3] ^ c[3] ^ c[7] ^ c[10] ^ c[11];
    end
   //else idle;

end
reg r_dout_valid = 0;
always @(posedge i_clk) //输入数据在一个时钟内完成CRC计算，下一个时钟输出;
begin
    r_dout_valid <= i_din_valid;
end

assign o_dout_valid = r_dout_valid;
assign o_dout = r_dout ;

endmodule
```

图 2-2 CRC16 代码

✧ 加密模块

加密模块的代码如图 2-3 所示。加密模块可以对并行输入的 16 位数据对应并行输出 32 位 CRC 加密后的数据。

首先定义端口，输入有时钟信号 clk、同步复位 rst、数据有效位 i_valid 和 16 位数据 din，输出 32 位加密后数据以及通过 LED 来表示输出有效位。当 LED 亮时说明此时输出有效，反之输出无效。

之后调用 CRC16 的 module，将计算出对应的 CRC 冗余校验码附在数据后形成完整的 32 位加密数据并输出。

```
module Encode(
    input wire clk,          //时钟;
    input wire rst,          //同步复位;
    input wire i_valid,      //输入数据有效;
    input wire [15:0]din,    //输入数据;
    output wire LED,         //输出CRC值有效;
    output wire [31:0]dout //输出CRC;
    );
    wire [15:0]crc;
    CRC16 CRC1(
    .i_clk(clk),
    .i_rst_n(rst),
    .i_din_valid(i_valid),
    .i_din(din),
    .o_dout_valid(LED),
    .o_dout(crc)
    );
    assign dout={din , crc};

endmodule
```

图 2-3 Encode 代码

✧ 校验模块

校验模块如图 2-4 所示，通过对并行输入的 32 位已加密数据解密，并判断是否出现传输错误，若传输无误 LED 亮，若有误 LED 灭。

首先定义端口，输入为时钟信号 clk、加密的 32 位数据 din，输入有效位 i_valid，输出为 LED 来表示校验结果。定义中间变量 data 为传输过来的原数据部分，crc 为传输过来的校验码部分，crc1 为通过原数据计算出的 CRC 校验码，通过比较 crc1 与 crc 是否相等来判断数据传输是否有误。调用 CRC16 的 module，计算出 crc1，当计算结束时中间变量 sign 由 0 跳变为 1，此时并行进行 crc 与 crc1 的位异或操作，并将结果存入 test 中。在每个时钟周期上升沿时，将 test 与 0x0000 进行比较，一旦进行上述异或操作后且传输无误，那么 test 将存放 0x0000，则 LED=1。

```
module Proof(
    input wire clk,          //时钟;
    input wire [31:0]din,    //总输入;
    input wire i_valid,      //输入数据有效;
    output reg LED
     );
     wire [15:0]data;        //数据部分;
     wire [15:0]crc;         //校验码部分;
     wire [15:0]crc1;
     reg [15:0]test;
     wire sign;

     assign data=din[31:16];//分配;
     assign crc=din[15:0];

     CRC16 CRC2(
     .i_clk(clk),
     .i_rst_n(i_valid),
     .i_din_valid(i_valid),
     .i_din(data),
     .o_dout_valid(sign),
     .o_dout(crc1)
     );
     always @ (posedge sign)
     begin
       test[0]=crc[0]^crc1[0];
       test[1]=crc[1]^crc1[1];
       test[2]=crc[2]^crc1[2];
       test[3]=crc[3]^crc1[3];
       test[4]=crc[4]^crc1[4];
       test[5]=crc[5]^crc1[5];
       test[6]=crc[6]^crc1[6];
       test[7]=crc[7]^crc1[7];
       test[8]=crc[8]^crc1[8];
       test[9]=crc[9]^crc1[9];
       test[10]=crc[10]^crc1[10];
       test[11]=crc[11]^crc1[11];
       test[12]=crc[12]^crc1[12];
       test[13]=crc[13]^crc1[13];
       test[14]=crc[14]^crc1[14];
       test[15]=crc[15]^crc1[15];
     end
  always @(posedge clk)
       begin
        if(test==15'h0000) LED=1;
        else LED=0;
       end
endmodule
```

图 2-4 Proof 代码

第三部分

完成情况及性能参数 /Final Design & Performance Parameters

✧ CRC16 模块仿真

首先编修仿真文件代码，如下图 3-1 所示，初始 rst=0，重置状态，在 100ns 时输入数据 0x0011，并且置 i_din_valid=1，一个周期后输入结束，再经历一个周期的时间间隔输入数据 0x0013，一周前后输入结束。

```
module TestBench();
  reg i_clk;
  reg i_rst_n;
  reg i_din_valid;
  reg [15:0]i_din;
  wire o_dout_valid;
  wire [15:0]o_dout;

  CRC16 test(
  .i_clk(i_clk),
  .i_rst_n(i_rst_n),
  .i_din_valid(i_din_valid),
  .i_din(i_din),
  .o_dout_valid(o_dout_valid),
  .o_dout(o_dout)
  );
  initial
    begin
      i_clk=0;
      i_rst_n=0;
      i_din_valid=0;
      #100;
      i_rst_n=1;
      i_din_valid=1;
      i_din=16'h0011;
      #20;
      i_din_valid=0;
      #20;
      i_din_valid=1;
      i_din=16'h0013;
      #20;
      i_din_valid=0;
    end
always #10 i_clk=~i_clk;
endmodule
```

图 3-1 CRC16 仿真文件

仿真结果图如下图 3-2 所示，数据 0x0011 的 CRC 校验码为 0x1f1f，计算正确；再次输入数据 0x0013，输出结果为 0xd2c1，与计算结果一致，说明 CRC16 功能正确。计算结果参考图 3-3、3-4。

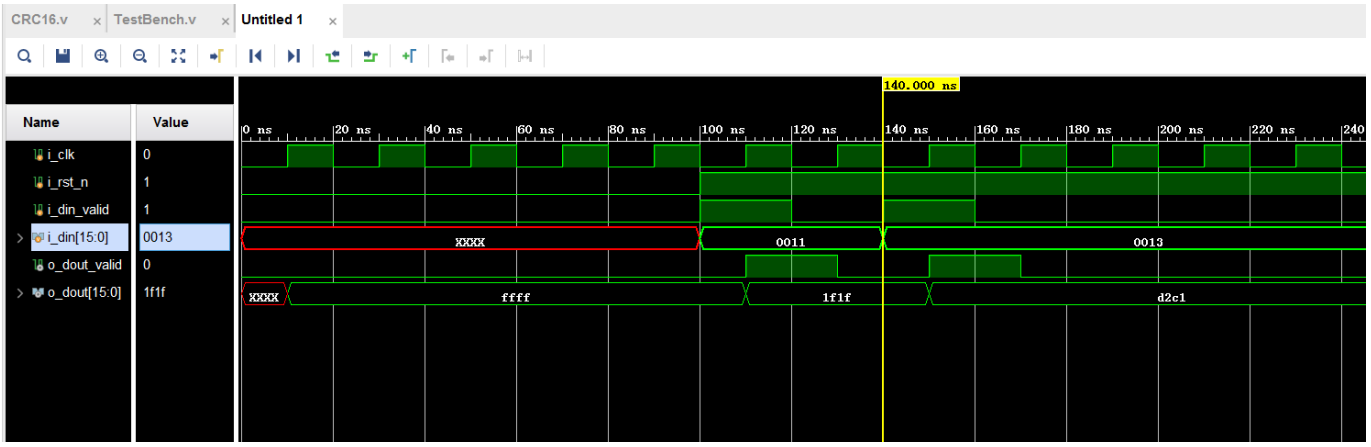

图 3-2 CRC 冗余码计算仿真

图 3-3 计算结果 1

图 3-4 计算结果 2

✧ 加密仿真

加密模块仿真结果如图 3-5 所示。输入的数据与上一个仿真实验一样，并行输出的是加密后的数据帧，与计算结果一样，验证了实验的正确性。

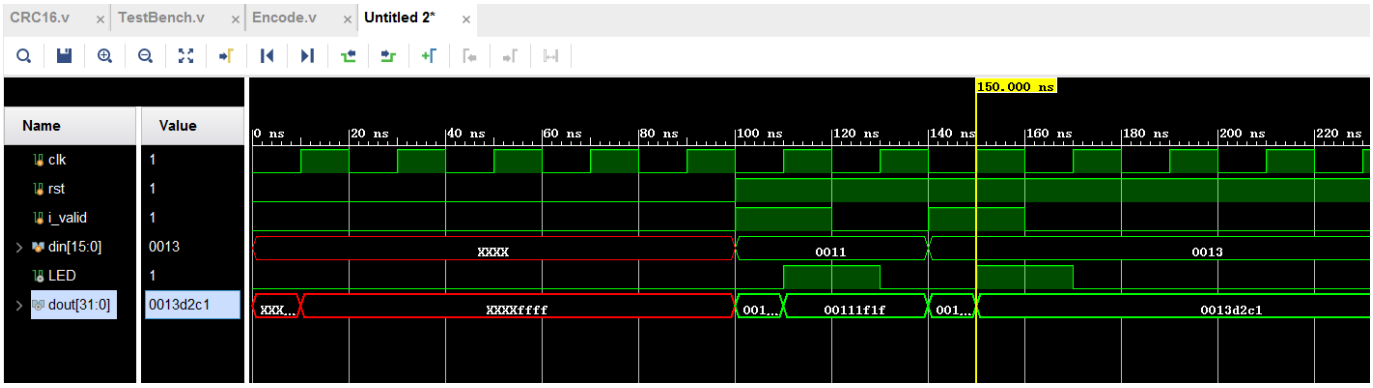

图 3-5 加密仿真

✧ 校验仿真

校验仿真的仿真代码如下图 3-6 所示。初始时，无数据输入，在 100ns 时输入数据 0x00111f1f，并且输入有效位 i_valid=1，保持 1 个周期输入完毕后，有效位置 0；间隔 100ns 后输入数据 0x00110000，保持 1 个周期输入后有效位置 0。

仿真的结果如图 3-7 所示，当输入 0x00111f1f 后，由于数据与 CRC 校验码匹配，说明传输无误，在 1 个周期的校验计算完成后 LED 亮；当输入数据 0x00110000 后，这是我人为输入的错误数据，LED 灭，验证了程序的正确性。

```
module TestBench();
  reg clk;
  reg i_valid;
  reg [31:0]din;
  wire LED;

  Proof test(
  .clk(clk),
  .i_valid(i_valid),
  .din(din),
  .LED(LED)
  );
  initial
    begin
      clk=0;
      i_valid=0;
      #100;
      i_valid=1;
      din=32'h00111f1f;
      #20;
      i_valid=0;
      #100;
      i_valid=1;
      din=32'h00110000;
      #20;
      i_valid=0;
    end
always #10 clk=~clk;
endmodule
```

图 3-6 校验仿真文件

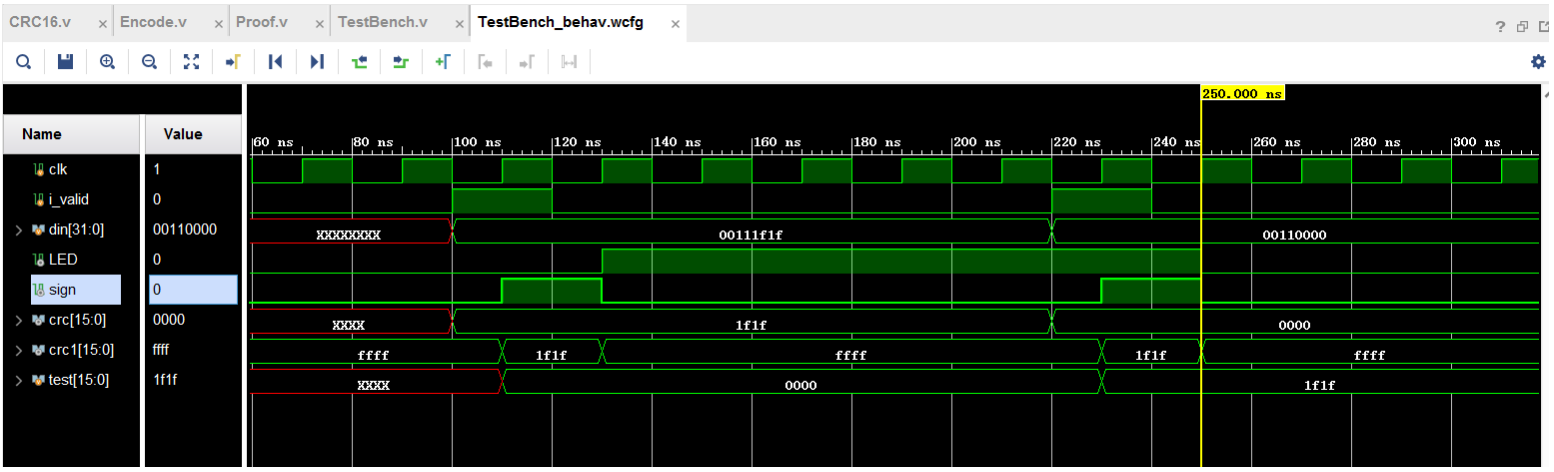

图 3-7 校验仿真结果

第四部分

总结 /Conclusions

这个项目完成了 CRC 加密/校验的 RTL 仿真，验证了 CRC 加密/校验的过程，同时也实现了对 16 位数据 CRC 加密的功能和对接收的 32 位数据校验的功能，具有一定的实用性。当然这个项目的实现也有一些不足，比如没有通过具体通信程序来实践操作 CRC 加密和校验的功能。

通过这次实验的编写给予我个人能力很大的提高。首先我了解 CRC 校验的基本原理，学会了 CRC 算法实现，也更加熟悉 verilog 语言，一定程度上提高了我的软硬件编程调试能力。在这过程中也遇到了一些困难和问题，例如对于有些变量在时序中变化顺序把握不准导致结果不正确，所以通过引入一些中间变量来进行逻辑计算,这个问题也让我对于时序逻辑的理解更加深刻。通过这次课程的学习，我对 FPGA 的了解更加深入，也对它功能的强大所震撼到，也希望通过学习我可以通过 FPGA 实现更多的功能。