

# CRC-32 Verilog 晶片模擬

# CRC-32 Verilog Chip Simulation

作者：陳躍升

系級：通訊工程學系 三甲

學號：D9932275

開課老師：趙啟時

課程名稱：網路通訊與協定

開課系所：通訊工程學系

開課學年： 101 學年度 第 1 學期

# 中文摘要

(1)目的 :

在數位通訊中,可能會因為各種原因導致資料在傳輸過程中或接收時發生錯誤,為了保證資料傳輸的可靠性和資料校驗的高效性,常常採用一些差錯控制方法。

迴圈冗餘校驗 CRC (Cyclic Redundancy Code) 就是一種被廣泛採用的差錯控制方法和資料編碼方法。它具有編碼簡單,檢錯和糾錯能力強等特點,能有效地對資料進行編碼,並可以顯著地提高系統地檢錯能力,從而保證資料傳輸的可靠性和正確性,因此在大多數的乙太網路協定中都採用了 CRC 校驗來進行差錯控制。

(2)過程及方法 :

使用硬體描述語言 Verilog HDL 並在 ModelSim 10.1c 版本的編譯器來進行程式的模擬以及校驗。對 CRC-16 的串列傳輸/並列傳輸以及 CRC-32 的並列傳輸一一進行模擬

(3)結果 :

依據不同 CRC 的生成多項式以及不同的資料傳輸的方式,會影響檢錯能力和資料傳輸的正確性、可靠性。

根據不同的 CRC 生成多項式,CRC-16 應用在 BM、SDLC 等等;CRC-32 應用在 ZIP、RAR、IEEE802、LAN/FDD I、PPP-FCS 等等,依據不同的傳輸方式,並列傳輸方式比串列傳輸方式更快速有效率但相對電路成本也會相對提高。

**關鍵字：**

1.迴圈冗餘校驗
2.硬體描述語言
3.Cyclic Redundancy Code (CRC)
4.ModelSim
5.Verilog HDL

# Abstract

(1) Purpose:

In digital communication, there has many reason cause the procedure of data transmission or receive happen mistake, in order to ensure the reliability of data transmission and the efficiency of data verify, it is often using error control.

Cyclic Redundancy Code (CRC) is a widely using in error control and data encoding. It has easy encoding, the strong ability of error detection and error correction, can encode data effectively, and significantly improve the ability of system's error correction. Hence, in many Ethernet network protocols are using CRC for error control.

(2) Procedure and Method:

It use hardware description language(HDL) Verilog and the compiler is in the version of ModelSim 10.1c for the purpose of simulation and verification. The simulation is for CRC-16 serial transmission/ parallel transmission and CRC-32 parallel transmission.

(3) Result:

Accroding to different generator polynomial of CRC and different mode of data transmission, it can affect the ability of error detection and the correctness, reliability of data transmission.

Under different generator polynomial of CRC, CRC-16 has the application of BM、SDLC and so on；CRC-32 has the application of ZIP、RAR、IEEE802、LAN/FDD I、PPP-FCS etc. Accroding to different type of transmission, the parallel transmission has better efficiency but the circuit cost also relative increase.



## Keyword：

1.Hardware Description Language (HDL)
2.Cyclic Redundancy Code (CRC)
3.ModelSim
4.Verilog HDL

# 目　次

# 一. CRC-16 串列傳輸

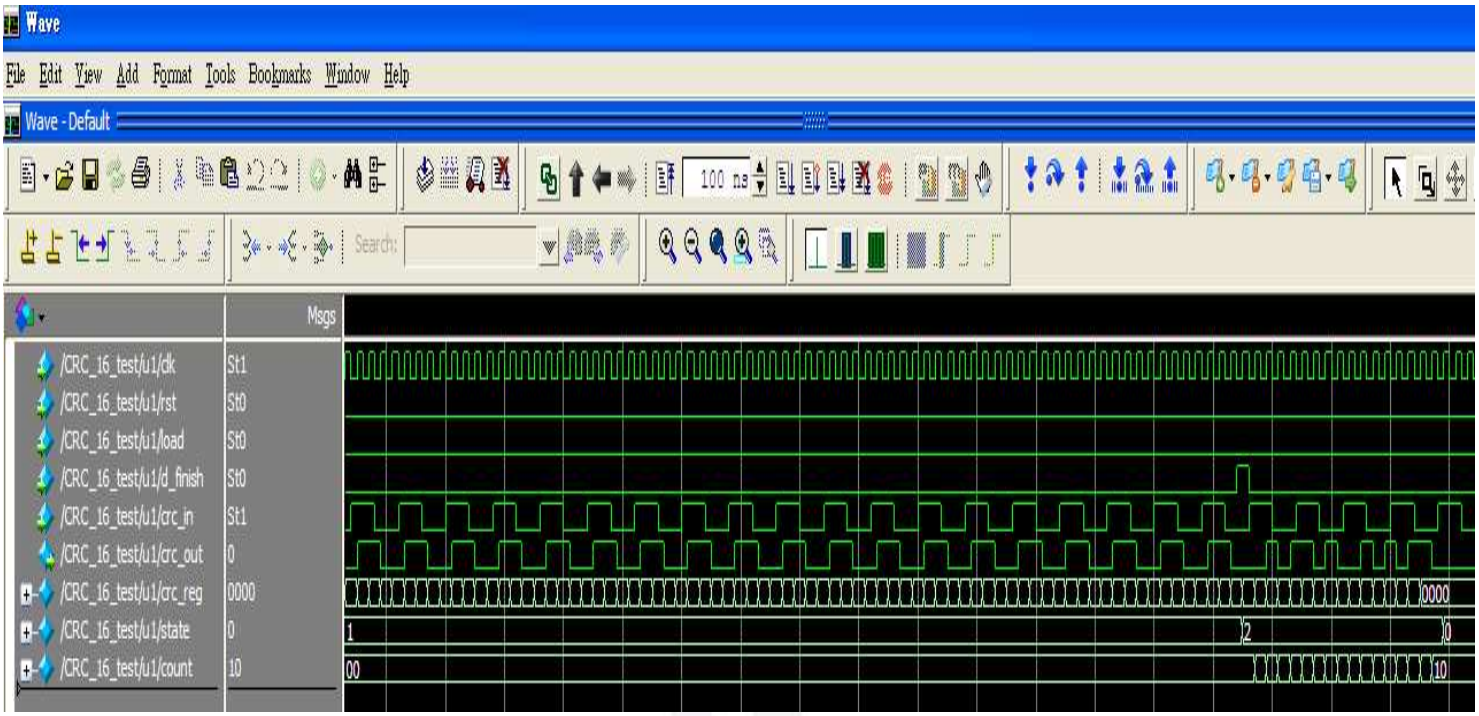
## (1)程式碼

```
`timescale 1ns/1ns //聲明本模組中所用到的時間單位和時間精度

//模組的宣告,CRC_16_serial(它的阜名)
//模組功能:該模組是 crc-16 校驗碼的編碼電路,該電路採用串列線性移位暫存器
module CRC_16_serial(clk,rst,load,d_finish,crc_in,crc_out);
input clk;          //輸入信號
input rst;          //重設信號
input load;         //開始編碼信號
input d_finish; //編碼結束信號
input crc_in;     //待編碼字輸入
output crc_out; //編碼後碼字輸出

reg crc_out;              //碼字輸出暫存器,1bit
reg [15:0] crc_reg; //線性移位暫存器,16bit
reg [1:0] state;       //狀態暫存器,2bit
reg [4:0] count;       //計數暫存器,5bit

parameter idle = 2'b00;     //等待狀態
parameter compute = 2'b01;//計算狀態
parameter finish = 2'b10; //計算結束狀態

always@(posedge clk) //在每次 clk 為正緣觸發時執行
begin                          //可看成 c 語言裡的上括弧,end 則可看成下括弧
case(state)                 //以 state 來選擇 case
      idle:begin            //是等待狀態
            if(load)       //load 信號有效進入 compute 狀態
                state <= compute;
            else
                state <= idle;
      end
      compute:begin       //d_finish 信號有效進入 finish 狀態
                if(d_finish)
                    state <= finish;
                else
```

```
                    state <= compute;
        end
        finish:begin        //判斷是否 16 個暫存器中的資料都完全輸出
                if(count==16)
                    state <= idle;
                else
                    count <= count+1;
        end
endcase
end

always@(posedge clk or negedge rst)//每當 clk 正緣觸發或是 rst 負緣觸發就執行
    if(rst)
    begin
        crc_reg[15:0] <= 16'b0000_0000_0000_0000;//暫存器預裝初值
        count <= 5'b0_0000;
        state <= idle;
    end
    else
    case(state)
         idle:begin
                crc_reg[15:0] <= 16'b0000_0000_0000_0000;
         end
         compute:begin
           //產生多項式 x^16+x^15+x^2+1
                crc_reg[0] <= crc_reg[15] ^ crc_in;
                crc_reg[1] <= crc_reg[0];
                crc_reg[2] <= crc_reg[1] ^ crc_reg[15] ^ crc_in;
                crc_reg[14:3] <= crc_reg[13:2];
                crc_reg[15] <= crc_reg[14] ^ crc_reg[15] ^ crc_in;
                crc_out <= crc_in;    //輸入作為輸出
         end
         finish:begin
                    crc_out <= crc_reg[15]; //暫存器 15 作為輸出
                    crc_reg[15:0] <= {crc_reg[14:0],1'b0}; //移位
         end
      endcase
endmodule
```

```
//模組 CRC-16-test
//模組功能:該模組是 crc16 校驗碼串列編碼電路,
//該編碼電路可將串列輸入的資料進行編碼
module CRC_16_test;
reg clk;
reg rst;
reg load;
reg d_finish;
reg crc_in;

wire crc_out;//線網型資料,1bit

parameter clk_period = 40; //clk 週期為 40ns

initial //初始化相關阜值
begin
    #clk_period clk = 1;
    #clk_period rst = 1;
    #clk_period rst = 0;
    #clk_period crc_in = 1; //輸入待編碼資料
    #clk_period load = 1;
    #clk_period load = 0;
    #clk_period d_finish = 0;
    #(80*clk_period) d_finish = 1;
    #clk_period d_finish = 0;
end

always #(clk_period/2) clk = ~clk;          //每 20ns 換電位
always #(2*clk_period) crc_in = ~crc_in;//每 80ns 更換待編碼字輸入電位

//調用串列編碼模組
CRC_16_serial
u1(.clk(clk), .rst(rst), .load(load), .d_finish(d_finish), .crc_in(crc_in), .crc_out(crc_out
);
Endmodule
```

## (2)CRC-16 串列編碼電路仿真圖(ModelSim)




## (3)仿真圖分析

上圖是測試程式的仿真結果，圖中輸入信號是”crc_in”，當資料沒有完全輸入編碼器時，輸入信號”crc_in”作為”crc_out”阜的輸出資料:
當”d_finish”信號為高電平指示資料登陸完成，這時將線性位移暫存器”crc_reg”中的校驗位依次輸出到”crc_out”阜。

# 二.CRC-16 並列傳輸

## (1)程式碼

```
‘timescale 1ns/1ns//聲明本模組中所用到的時間單位和時間精度

//模組功能:該模組是 crc-16 校驗碼的並行編碼電路,該編碼電路可以將並行輸入的 8bits 資料進行編碼
module CRC_16_paraller(clk,rst,load,d_finish,crc_in,crc_out);
input clk;          //輸入阜 clk,時脈訊號
input rst;          //輸入阜 rst,重設訊號
input load;         //輸入阜 load,開始編碼訊號
input d_finish;     //輸入阜,編碼結束信號
input [7:0] crc_in; //編碼器並行 8bits 資料登錄
```

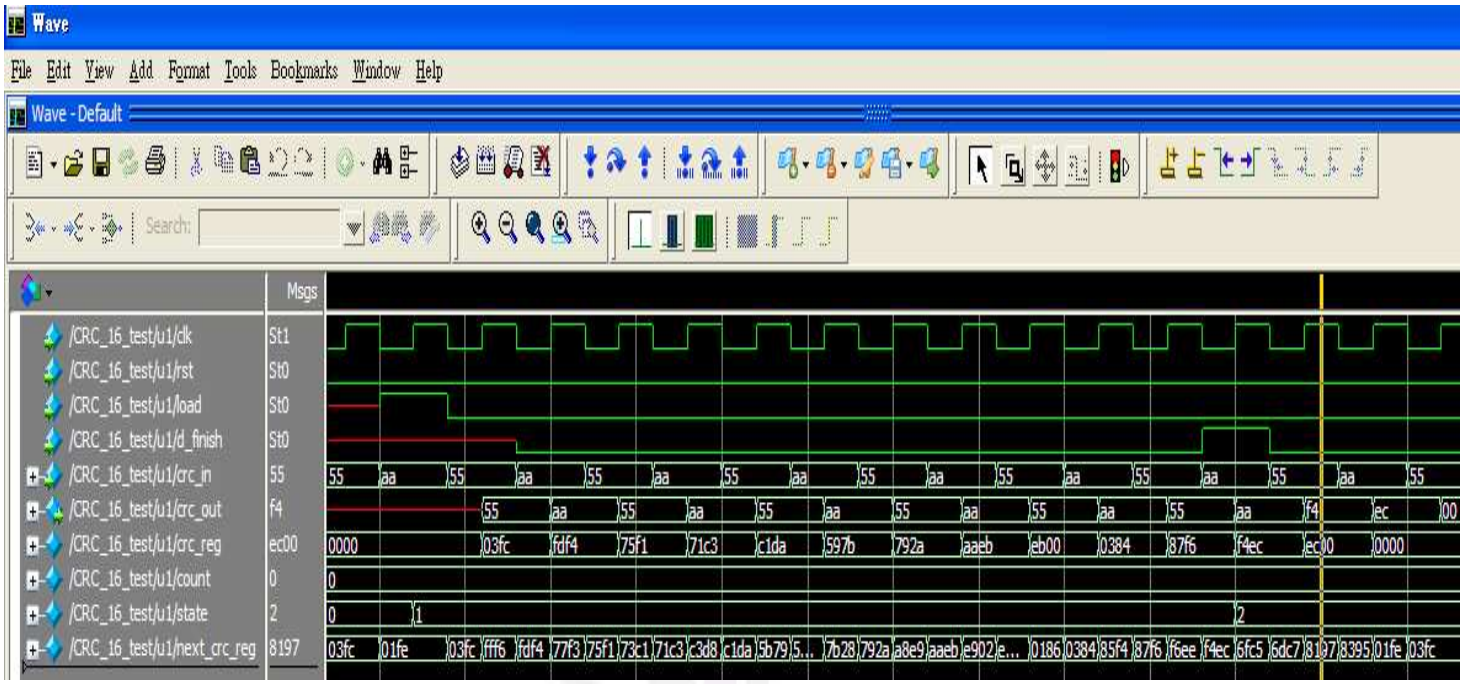
```
output  [7:0] crc_out;  //編碼器並行 8bits 輸出

reg [7:0] crc_out; //資料輸出暫存器
reg [15:0] crc_reg;//編碼器移位暫存器
reg [1:0] count;   //計數暫存器
reg [1:0] state;   //狀態暫存器

wire [15:0] next_crc_reg; //移位暫存器輸入變數

parameter idle = 2'b00;   //宣告參數 idle 為二進位 00
parameter compute = 2'b01;//宣告參數 compute 為二進位 01
parameter finish = 2'b10; //宣告參數 finish 為二進位 10


//暫存器輸出和暫存器初始狀態以及輸入碼字組合邏輯關係
assign next_crc_reg[0] = (^crc_in[7:0]) ^ (^crc_reg[15:8]);
assign next_crc_reg[1] = (^crc_in[6:0]) ^ (^crc_reg[15:9]);
assign next_crc_reg[2] = crc_in[7] ^ crc_in[6] ^ crc_reg[9] ^ crc_reg[8];
assign next_crc_reg[3] = crc_in[6] ^ crc_in[5] ^ crc_reg[10] ^
crc_reg[9];
assign next_crc_reg[4] = crc_in[5] ^ crc_in[4] ^ crc_reg[11] ^
crc_reg[10];
assign next_crc_reg[5] = crc_in[4] ^ crc_in[3] ^ crc_reg[12] ^
crc_reg[11];
assign next_crc_reg[6] = crc_in[3] ^ crc_in[2] ^ crc_reg[13] ^
crc_reg[12];
assign next_crc_reg[7] = crc_in[2] ^ crc_in[1] ^ crc_reg[14] ^
crc_reg[13];
assign next_crc_reg[8] = crc_in[1] ^ crc_in[0] ^ crc_reg[15] ^ crc_reg[14]
^ crc_reg[0];
assign next_crc_reg[9] = crc_in[0] ^ crc_reg[15] ^ crc_reg[1];
assign next_crc_reg[14:10] = crc_reg[6:2];
assign next_crc_reg[15] = (^crc_in[7:0]) ^ (^crc_reg[15:7]);

always@(posedge clk) //每當 clk 正緣觸發就執行
begin
case(state)          //以 state 來選擇 case
    idle:begin       //是等待狀態
```

```
          if(load)    //load 信號有效進入 compute 狀態
             state <= compute;
          else
             state <= idle;
     end
     compute:begin
          if(d_finish)//d_finish 信號有效進入 finish 狀態
             state <= finish;
          else
             state <= compute;
     end
     finish:begin
         if(count==2)//count 信號不等於 2 進入 finish 狀態
            state <= idle;
         else
            state <= finish;
     end
endcase
end

always@(posedge clk or negedge rst)//每當 clk 正緣觸發或是 rst 負緣觸發
就執行
   if(rst)
   begin
      crc_reg[15:0] <= 16'b0000_0000_0000_0000;//暫存器預裝初值
      state <= idle;
      count <= 2'b00;
   end
   else
   case(state)
       idle:begin     //暫存器裝初值狀態
            crc_reg[15:0] <= 16'b0000_0000_0000_0000;
       end
       compute:begin  //編碼計算狀態
               crc_reg[15:0]<= next_crc_reg[15:0];
               crc_out[7:0] <= crc_in[7:0];
       end
       finish:begin   //編碼結束狀態
```

```
                crc_reg[15:0] <= {crc_reg[7:0],8'b0000_0000};
                crc_out[7:0] <= crc_reg[15:8];
        end
    endcase
endmodule

//模組功能:該模組是 crc_16 校驗碼的並行編碼電路,該編碼電路可以將並行輸
入的 8bits 資料進行編碼
module CRC_16_test;
reg clk;
reg rst;
reg load;
reg d_finish;
reg [7:0] crc_in;

wire [7:0] crc_out;

parameter clk_period = 40;

//初始值的設置
initial
begin
   #clk_period clk = 1;
   #clk_period rst = 1;
   #clk_period rst = 0;
   #clk_period crc_in[7:0] = 8'b1010_1010; //輸入待編碼資料
   #clk_period load = 1;
   #clk_period load = 0;
   #clk_period d_finish = 0;
   #(10*clk_period) d_finish = 1;
   #clk_period d_finish = 0;
end

always #(clk_period/2) clk = ~clk;
always #(clk_period) crc_in[7:0] = ~crc_in[7:0]; //輸入待編碼資料

//調用並行編碼模組
CRC_16_paraller
```

```
ul(.clk(clk), .rst(rst), .load(load), .d_finish(d_finish), .crc_in(cr
c_in), .crc_out(crc_out));

endmodule
```

## (2) CRC-16 並列編碼電路仿真圖(ModelSim)




## (3)仿真圖分析

如上圖所示是 CRC-16 並行編碼電路測試程式仿真結果，crc_in 是輸入信號，當資料沒有完全輸入編碼器時，輸入信號 crc_in 作為 crc_out 阜的輸出資料:當 d_finish 信號為高電平指示資料登度完成，此時將線性移位暫存器 crc_reg 中的校驗位並行 8 為輸出到 crc_out 阜。

## 三.CRC-32 並列傳輸

**(X^32+ X^26+ X^23+ X^22+ X^16+ X^12+ X^11+ X^10+ X^8+ X^7+ X^5+ X^4+ X^2+ X^1+ 1)**

## (1)程式碼

```
‘timescale 1ns/1ns //聲明本模組中所用到的時間單位和時間精度
//////////////////////////////////////////
// Module Name:    CRC_32_paraller
// Tool versions:  ModelSim 10.1c
```

```
//////////////////////////////////////////

//模組功能:該模組是 crc-16 校驗碼的”並行”編碼電路,該編碼電路可以將並行輸入的 8bits 資料進行編碼
module CRC_32_paraller(clk,rst,load,d_finish,crc_in,crc_out);
input clk; //輸入阜 clk,時脈訊號
input rst; //輸入阜 rst,重設訊號
input load; //輸入阜 load,開始編碼訊號
input d_finish; //輸入阜,編碼結束信號
input [7:0] crc_in; //編碼器並行 8bits 資料登錄

output  [7:0] crc_out; //編碼器並行 8bits 輸出

reg [7:0] crc_out;  //資料輸出暫存器
reg [31:0] crc_reg; //編碼器移位暫存器
reg [1:0] count;    //計數暫存器
reg [1:0] state;    //狀態暫存器

wire [31:0] next_crc_reg;  //移位暫存器輸入變數

parameter idle = 2'b00;     //宣告參數 idle 為二進位 00
parameter compute = 2'b01;  //宣告參數 compute 為二進位 01
parameter finish = 2'b10;   //宣告參數 finish 為二進位 10


//暫存器輸出和暫存器初始狀態以及輸入碼字組合邏輯關係
assign next_crc_reg[0] = crc_reg[24] ^ crc_reg[30] ^ crc_in[0] ^
crc_in[6];
assign next_crc_reg[1] = crc_reg[24] ^ crc_reg[25] ^ crc_reg[30] ^
crc_reg[31] ^ crc_in[0] ^ crc_in[1] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[2] = crc_reg[24] ^ crc_reg[25] ^ crc_reg[26] ^
crc_reg[30] ^ crc_reg[31] ^ crc_in[0] ^ crc_in[1] ^ crc_in[2] ^ crc_in[6]
^ crc_in[7];
assign next_crc_reg[3] = crc_reg[25] ^ crc_reg[26] ^ crc_reg[27] ^
crc_reg[31] ^ crc_in[1] ^ crc_in[2] ^ crc_in[3] ^ crc_in[7];
assign next_crc_reg[4] = crc_reg[24] ^ crc_reg[26] ^ crc_reg[27] ^
crc_reg[28] ^ crc_reg[30] ^ crc_in[0] ^ crc_in[2] ^ crc_in[3] ^ crc_in[4]
^ crc_in[6];
```

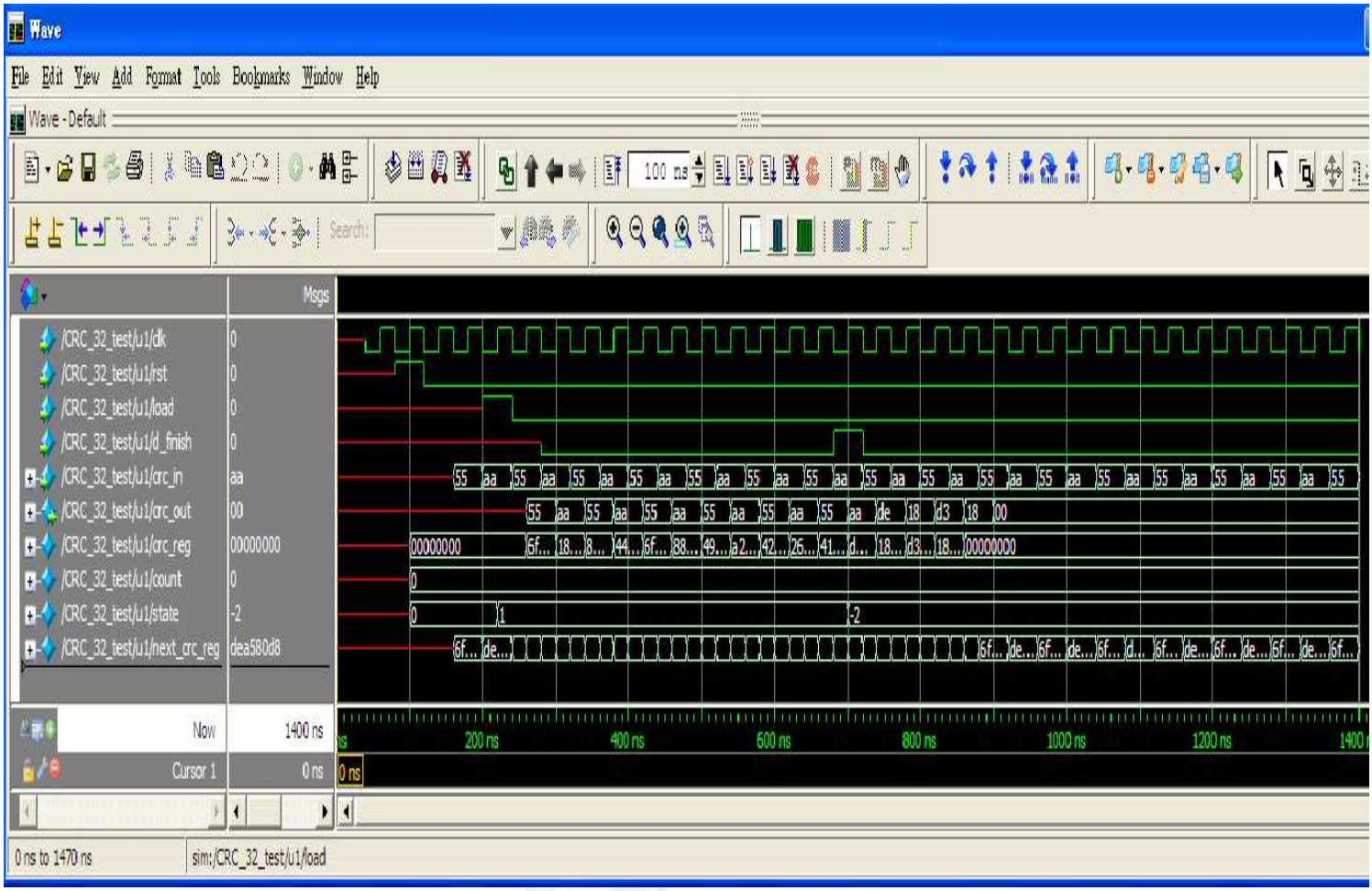
```
assign next_crc_reg[5] = crc_reg[24] ^ crc_reg[25] ^ crc_reg[27] ^
crc_reg[28] ^ crc_reg[29] ^ crc_reg[30] ^ crc_reg[31] ^ crc_in[0] ^
crc_in[1] ^ crc_in[3] ^ crc_in[4] ^ crc_in[5] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[6] = crc_reg[25] ^ crc_reg[26] ^ crc_reg[28] ^
crc_reg[29] ^ crc_reg[30] ^ crc_reg[31] ^ crc_in[1] ^ crc_in[2] ^ crc_in[4]
^ crc_in[5] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[7] = crc_reg[24] ^ crc_reg[26] ^ crc_reg[27] ^
crc_reg[29] ^ crc_reg[31] ^ crc_in[0] ^ crc_in[2] ^ crc_in[3] ^ crc_in[5]
^ crc_in[7];
assign next_crc_reg[8] = crc_reg[0] ^ crc_reg[24] ^ crc_reg[25] ^
crc_reg[27] ^ crc_reg[28] ^ crc_in[0] ^ crc_in[1] ^ crc_in[3] ^ crc_in[4];
assign next_crc_reg[9] = crc_reg[1] ^ crc_reg[25] ^ crc_reg[26] ^
crc_reg[28] ^ crc_reg[29] ^ crc_in[1] ^ crc_in[2] ^ crc_in[4] ^ crc_in[5];
assign next_crc_reg[10] = crc_reg[2] ^ crc_reg[24] ^ crc_reg[26] ^
crc_reg[27] ^ crc_reg[29] ^ crc_in[0] ^ crc_in[2] ^ crc_in[3] ^ crc_in[5];
assign next_crc_reg[11] = crc_reg[3] ^ crc_reg[24] ^ crc_reg[25] ^
crc_reg[27] ^ crc_reg[28] ^ crc_in[0] ^ crc_in[1] ^ crc_in[3] ^ crc_in[4];
assign next_crc_reg[12] = crc_reg[4] ^ crc_reg[24] ^ crc_reg[25] ^
crc_reg[26] ^ crc_reg[28] ^ crc_reg[29] ^ crc_reg[30] ^ crc_in[0] ^
crc_in[1] ^ crc_in[2] ^ crc_in[4] ^ crc_in[5] ^ crc_in[6];
assign next_crc_reg[13] = crc_reg[5] ^ crc_reg[25] ^ crc_reg[26] ^
crc_reg[27] ^ crc_reg[29] ^ crc_reg[30] ^ crc_reg[31] ^ crc_in[1] ^
crc_in[2] ^ crc_in[3] ^ crc_in[5] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[14] = crc_reg[6] ^ crc_reg[26] ^ crc_reg[27] ^
crc_reg[28] ^ crc_reg[30] ^ crc_reg[31] ^ crc_in[2] ^ crc_in[3] ^ crc_in[4]
^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[15] = crc_reg[7] ^ crc_reg[27] ^ crc_reg[28] ^
crc_reg[29] ^ crc_reg[31] ^ crc_in[3] ^ crc_in[4] ^ crc_in[5] ^ crc_in[7];
assign next_crc_reg[16] = crc_reg[8] ^ crc_reg[24] ^ crc_reg[28] ^
crc_reg[29] ^ crc_in[0] ^ crc_in[4] ^ crc_in[5];
assign next_crc_reg[17] = crc_reg[9] ^ crc_reg[25] ^ crc_reg[29] ^
crc_reg[30] ^ crc_in[1] ^ crc_in[5] ^ crc_in[6];
assign next_crc_reg[18] = crc_reg[10] ^ crc_reg[26] ^ crc_reg[30] ^
crc_reg[31] ^ crc_in[2] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[19] = crc_reg[11] ^ crc_reg[27] ^ crc_reg[31] ^
crc_in[3] ^ crc_in[7];
assign next_crc_reg[20] = crc_reg[12] ^ crc_reg[28] ^ crc_in[4];
assign next_crc_reg[21] = crc_reg[13] ^ crc_reg[29] ^ crc_in[5];
```

```
assign next_crc_reg[22] = crc_reg[14] ^ crc_reg[24] ^ crc_in[0];
assign next_crc_reg[23] = crc_reg[15] ^ crc_reg[24] ^ crc_reg[25] ^
crc_reg[30] ^ crc_in[0] ^ crc_in[1] ^ crc_in[6];
assign next_crc_reg[24] = crc_reg[16] ^ crc_reg[25] ^ crc_reg[26] ^
crc_reg[31] ^ crc_in[1] ^ crc_in[2] ^ crc_in[7];
assign next_crc_reg[25] = crc_reg[17] ^ crc_reg[26] ^ crc_reg[27] ^
crc_in[2] ^ crc_in[3];
assign next_crc_reg[26] = crc_reg[18] ^ crc_reg[24] ^ crc_reg[27] ^
crc_reg[28]^crc_reg[30]^crc_in[0]^crc_in[3]^crc_in[4]^crc_in[6];
assign next_crc_reg[27] = crc_reg[19] ^ crc_reg[25] ^ crc_reg[28] ^
crc_reg[29]^crc_reg[31]^crc_in[1]^crc_in[4]^crc_in[5]^crc_in[7];
assign next_crc_reg[28] = crc_reg[20] ^ crc_reg[26] ^ crc_reg[29] ^
crc_reg[30] ^ crc_in[2] ^ crc_in[5] ^ crc_in[6];
assign next_crc_reg[29] = crc_reg[21] ^ crc_reg[27] ^ crc_reg[30] ^
crc_reg[31] ^ crc_in[3] ^ crc_in[6] ^ crc_in[7];
assign next_crc_reg[30] = crc_reg[22] ^ crc_reg[28] ^ crc_reg[31] ^
crc_in[4] ^ crc_in[7];
assign next_crc_reg[31] = crc_reg[23] ^ crc_reg[29] ^ crc_in[5];
//每當 clk 正緣觸發就執行
always@(posedge clk)
begin
case(state)       //以 state 來選擇 case
    idle:begin    //是等待狀態
         if(load) //load 信號有效進入 compute 狀態
            state <= compute;
         else
            state <= idle;
    end
    compute:begin
            if(d_finish) //d_finish 信號有效進入 finish 狀態
               state <= finish;
            else
               state <= compute;
    end
    finish:begin
           if(count==2) //count 信號不等於 2 進入 finish 狀態
              state <= idle;
           else
```

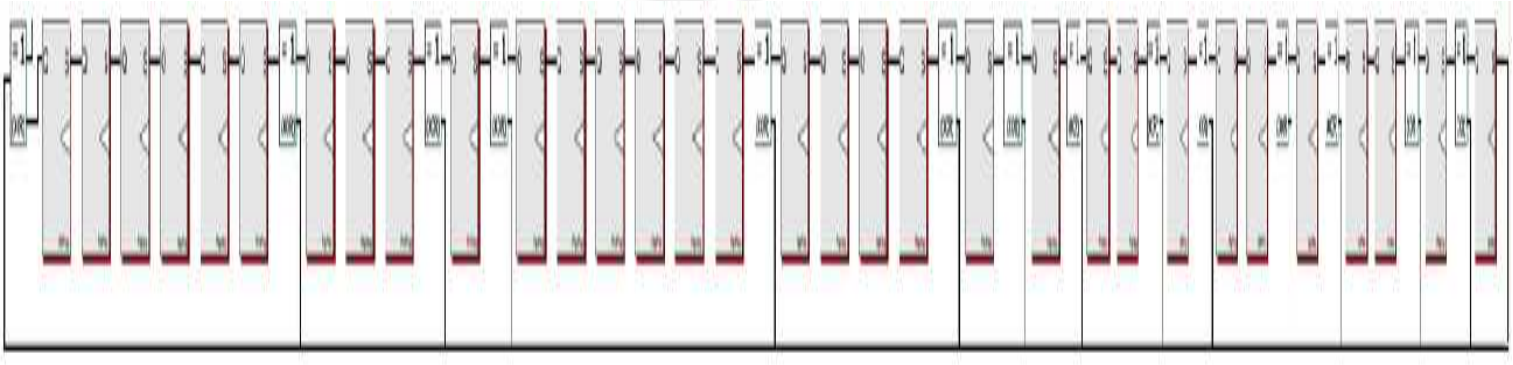
```
                state <= finish;
    end
endcase
end

always@(posedge clk or negedge rst) //每當 clk 正緣觸發或是 rst 負緣觸發
就執行
   if(rst)
   begin
      crc_reg[31:0] <= 32'b0000_0000_0000_0000_0000_0000_0000_0000; //
暫存器預裝初值
      state <= idle;
      count <= 2'b00;
   end
   else
   case(state)
       idle:begin //暫存器裝初值狀態
            crc_reg[31:0] <=
32'b0000_0000_0000_0000_0000_0000_0000_0000;
       end
       compute:begin //編碼計算狀態
               crc_reg[31:0]<= next_crc_reg[31:0];
               crc_out[7:0] <= crc_in[7:0];
       end
       finish:begin //編碼結束狀態
              crc_reg[31:0] <= {crc_reg[23:0],8'b0000_0000};
              crc_out[7:0] <= crc_reg[31:24];
       end
    endcase
endmodule


//模組功能:該模組是 crc_16 校驗碼的並行編碼電路,該編碼電路可以將並行輸
入的 8bits 資料進行編碼
module CRC_32_test;
reg clk;
reg rst;
reg load;
```

```
reg d_finish;
reg [7:0] crc_in;

wire [7:0] crc_out;

parameter clk_period = 40;

//初始值的設置
initial
begin
   #clk_period clk = 1;
   #clk_period rst = 1;
   #clk_period rst = 0;
   #clk_period crc_in[7:0] = 8'b1010_1010; //輸入待編碼資料
   #clk_period load = 1;
   #clk_period load = 0;
   #clk_period d_finish = 0;
   #(10*clk_period) d_finish = 1;
   #clk_period d_finish = 0;
end

always #(clk_period/2) clk = ~clk;
always #(clk_period) crc_in[7:0] = ~crc_in[7:0]; //輸入待編碼資料

//調用並行編碼模組
CRC_32_paraller
ul(.clk(clk), .rst(rst), .load(load), .d_finish(d_finish), .crc_in(cr
c_in), .crc_out(crc_out));
Endmodule
```

## （二）程式執行結果




## (三)邏輯電路圖



## 四. 心得

這次的 CRC-16 的課本兩個範例都有兜出來，但是 CRC-32 花了不少時間才用出來，原則上跟 CRC16 並行的方法大同小異，差別再於 CRC32 的處理多項式比較多，其餘都是不太難的。另外 CRC32 是用並列的方式完成，串列的方法也是可行但是對於要處理多位元以及高速執行就要用並列的方式才比較有效率，只是並列的方法要使用較多的邏輯閘。

# 參考文獻

1. Verilog HDL 的通信系統設計

作者:陳曦、邱志成、張鵬、何初冬 等編著 安亮 審校