应用笔记

NOVOSENSE 纳芯微电子

曼彻斯特编码的隔离ADC解码方法

AN-13-0020

作者：Jianan Shi

AN-13-0020

曼彻斯特编码的隔离ADC解码方法

NOVOSENSE 纳芯微电子

摘 要

曼彻斯特编码是一种同步编码技术，也称为相位编码，其通过电平的高低转换表示”0”或”1”。曼彻斯特编码将分离的数据和时钟合并成一个单一的、自同步的数据流，适用于串行信道上的数据传输。NSI1303E是采用了曼彻斯特编码的隔离ADC，其输出的解码方式与常规的Σ-Δ ADC有所不同，本文介绍了对曼彻斯特编码形式输出的Σ-Δ类型ADC的解码方式。

目 录

1. 曼彻斯特编码介绍

1.1.曼彻斯特编码的定义

曼彻斯特编码是一种典型的同步数组编码计数，常用与数据通信和数字信号处理领域。如图1.1所示，曼彻斯特编码会将一个完整的数据位周期分为两个子周期，每个周期都由一半的低电平子周期和一半高电平子周期组成。因此曼彻斯特编码中每个数据位周期内必然包含一次完整的电平跳变，在传输逻辑1时会在信号的中间部分由低电平变为高电平，而逻辑0则会在信号的中间部分由高电平变为低电平。在主流的“相位编码法”中，将一个周期内由低电平跳变为高电平定义为数据1，将高电平跳变为低电平定义为数据0（部分协议中定义相反，但物理本质一致）。

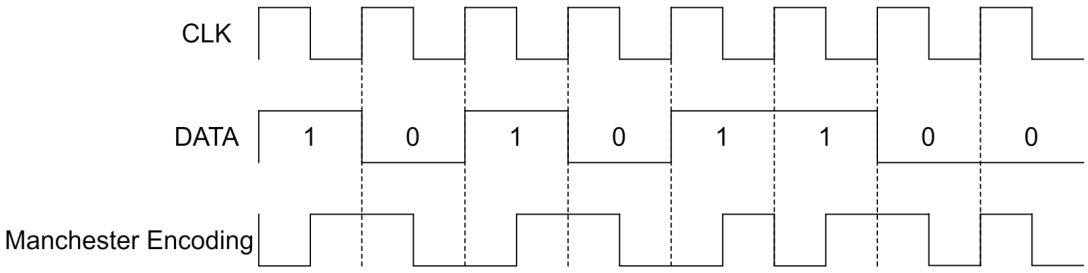

图1.1 曼彻斯特编码方式

1.2.曼彻斯特编码的特点及应用优势

曼彻斯特编码区别与传统的不归零编码（NRZ）相比的最主要特点是其在每一个数据位周期中间一定存在电平跳变，并通过这个电平跳变的极性来表示所传输的数据。这一特点使得曼彻斯特编码在实际应用中存在以下几点优势：

- **时钟自同步，减轻布线压力，优化EMI表现：**曼彻斯特编码是一种自同步的编码方式，通过将数据与时钟合并到一起，不需要额外的时钟信号，在接收端可以从编码中还原出时钟以及数据。利用曼彻斯特编码的这种特性可以减少板级的高速信号布线需求，简化系统设计；同时高频时钟走线的减少也可以直接减少电磁干扰发射源，优化系统整体的EMI表现。

- **边沿敏感信号，抗干扰性更优：**一方面，检测信号的“边沿”而非信号的“电平”，降低系统随机噪声导致的信号被“淹没”的风险。另一方面，编码后信号的理论上平均值是稳定的，实现了良好的直流平衡，不仅适合通过变压器耦合传输，还能减少信号在长距离传输中的失真概率。

- **实现成本可控，适配难度低：**相较于需要复杂同步机制的编码方式，曼彻斯特编码的编解码逻辑简洁高效，尤其适合资源有限的嵌入式设备与低成本系统。编码端可通过简单的2选1数字选择器实现核心功能，无需专用芯片；解码端仅需缓存器保存前后时钟周期的信号状态，通过对比“01”“10”等特征组合即可还原数据，无需复杂的相位锁定环路（PLL）。在FPGA应用中，曼彻斯特编解码模块仅需200-400个逻辑单元，就能实现超过20Mbps的传输速率，既满足了工业控制的实时性需求，又控制了硬件资源消耗。

综上，曼彻斯特编码的应用优势主要体现在自同步、抗干扰、低成本等技术特性。尽管其50%的编码效率在更高速的传输场景中存在局限，但在中低速、高可靠性、资源受限的应用场景中，曼彻斯特编码仍有独有的优势。

2.曼彻斯特编码的解码基础

由于曼彻斯特编码是一种自同步的编码，其解码过程就需要考虑跨时钟域的同步问题，本文给出了一个以过采样的方式实现的应用范例。

2.1.编码的状态

曼彻斯特编码的具体形式如下图所示：

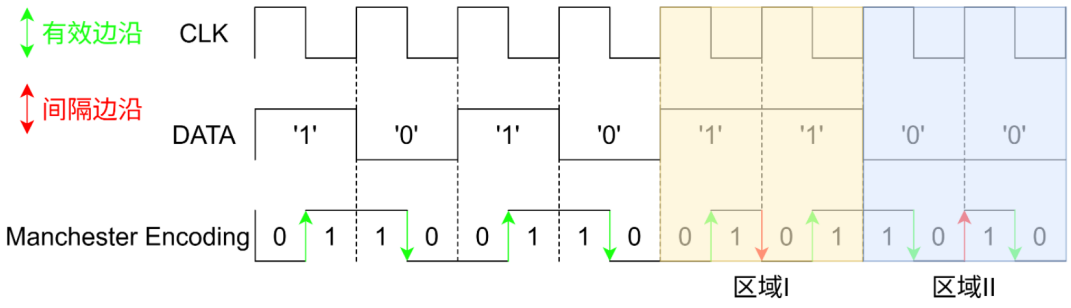

图 2.1曼彻斯特编码中的两种边沿

为了方便阐明解码方式，这里先定义一些基础概念的名称。首先曼彻斯特编码是一种边沿敏感的编码，其解码的本质就是识别出编码中代表了数据的有效边沿，但事实上在曼彻斯特编码中存在着两种类型的边沿。于是将曼彻斯特编码中存在的两种信号边沿进行了分类，如图 2.1所示，图中绿色的跳变边沿称为有效边沿，图中红色的边沿称为间隔边沿。进一步为了方便描述，本文中将编码前的数据记为带引号’0’和’1’，将编码后一个数据位周期内的两个子周期的电平记为不带引号的0和1。

在此基础之上，可以从图 2.1中发现曼彻斯特编码的以下几个特点：

1.曼彻斯特编码为了实现用不同极性的边沿对数据进行编码，需要在每一个数据位的起始位置将电平回复到特定的位置，这就会引入实际有效编码边沿以外的边沿。因此曼彻斯特编码中存在着两种跳变边沿，分别是代表着原始编码数据的有效边沿(对应图2.1中的绿色边沿)和连续数据之间的间隔边沿(对应图2.1中的红色边沿)；

2.在极端情况下，如果未编码的原始数据始终是’0’或’1’，得到的编码结果就如图2.1中的区域I或者区域II所示，始终是两种上升沿与下降沿以固定的延时（半个时钟周期）交替出现，也就是系统会进入一种状态未定的状态，无法区分那一种极性的边沿是有效的边沿；

3.但实际上，所编码的数据中一定会出现’0’变成’1’或者是’1’变成’0’的情况，当发生数据的变化时就可以发现上升沿与下降沿之间的时间间隔变为了完整的一个时钟周期，且这个完整时钟周期间隔的两个边沿一定都是有效的。

根据上面的这些结论可以得到曼彻斯特编码的解码实现思路，这里将曼彻斯特编码后的数据经过采样后得到的结果看作是一个输入序列，事实上实现解码的本质就是一个序列检测器，前文中所描述的不带引号的曼彻斯特编码中的0和1可以看作是这个序列检测器的以固定频率采样到的输入。

- 解码器刚开始工作时，事实上是无法判断第一个到来的边沿是否有效的。并且如果原始数据始终是’0’或’1’的话，曼彻斯特编码的结果就会是间隔为半个时钟周期的连续的上升与下降沿交替出现。此时依然无法哪些边沿是有效的；

- 而当出现两个边沿之间的间隔时间是一个完整周期时，就可以确定这时的边沿一定是有效的，解码电路随之进入到一个确定的状态；

- 当电路进入到确定状态之后，如何利用采样得到的输入序列进行解码，下面继续进行具体的讨论。这里首先列出所有可能的数据组合的情形，任意单bit的数据序列的连续两位数据都只可能有四种编码组合：’00’\’01’\’10’\’11’，如下图所示：

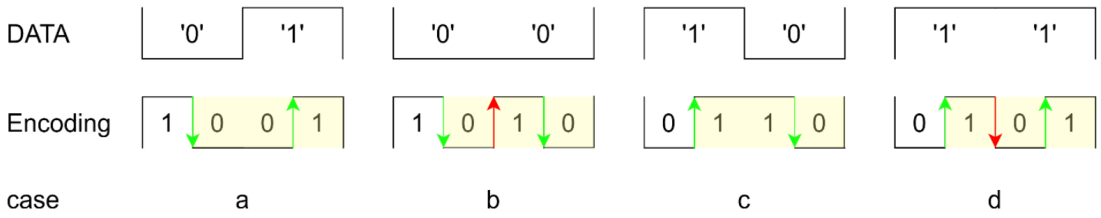

图2.2 曼彻斯特编码中连续编码的可能情形

根据图中的差异可以根据边沿之间的延时来对不同类型的边沿进行区分，具体而言：
a)在数据“0”之后的输入序列如果是00，则下一个数据是“1”；
b)在数据“0”之后的输入序列如果是01，则下一个数据是“0”；
c)在数据“1”之后的输入序列如果是10，则下一个数据是“1”；
d)在数据“1”之后的输入序列如果是11，则下一个数据是“0”。

据此可以画出状态机如下：

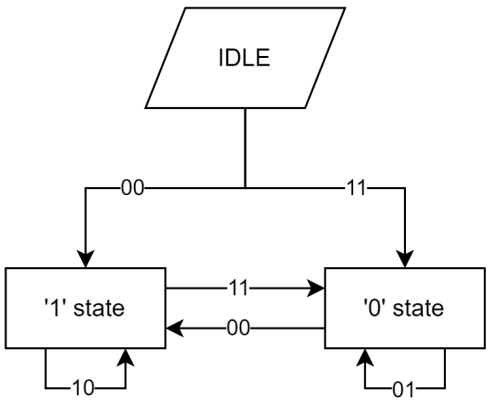

图2.2 曼彻斯特编码的解码状态机

如前所述，由于无法确定编码的初始位置，电路在初始状态下并不会进行解码，等到接收到第一个00或是11序列后，电路进入正常的连续解码状态。后续就会如上文的讨论电路进行连续的解码。

2.2.曼彻斯特编码的电路实现

基于上述状态机编写完整verilog代码见附录，这里给出整体代码的框图进行说明：

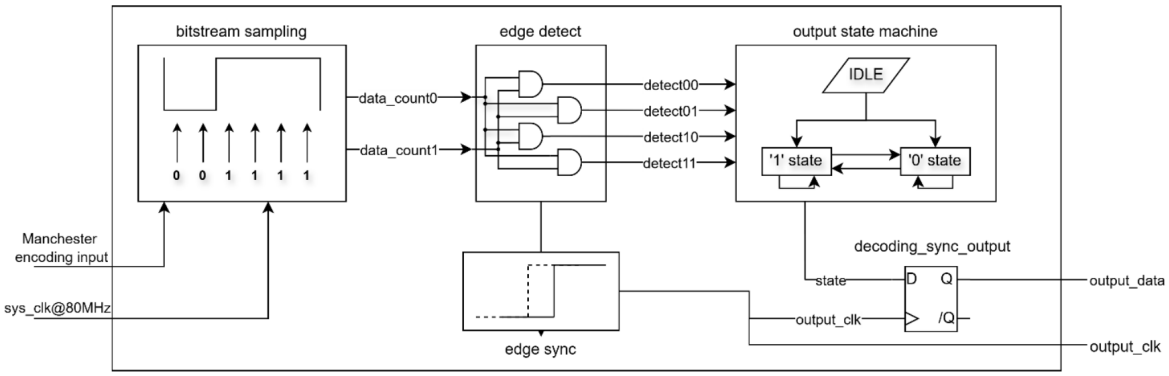

图2.3 曼彻斯特编码解码电路框图

为了避免时钟不同步带来的问题，在这个例子中用了4倍于ADC时钟(20MHz)的采样时钟sys_clk@80MHz，将曼彻斯特编码的ADC输出采样为频率更高的数字序列。对数字序列中连续的0或1的数量进行计数并记录在寄存器data_count0和data_count1中。

在计数结果的基础上，通过一些组合逻辑可以得到数字序列中的一些特征并提取为detect00/detect01/-detect10/detect11四个flag，通过这些flag控制解码状态机的状态就可以实现曼彻斯特编码的解码，最后将解码的结果通过边沿检测得到的结果同步输出即可。

3.曼彻斯特编码解码过程的仿真

仿真使用的bench文件见附录，仿真时给得原始数据序列是一个不断重复的存在，其对应的解码结果如表3.1所示：

表3.1 解码电路仿真结果

case	a	b	b
原始数据	0b 11100101	0b 00110101	0b 10100101
解码结果	0b 00101 11100101 111	0b 110101 00110101 00	0b 0100101 10100101 1

每一个case的具体分析如下：

a)原始数据是一个’111’开头的序列，实际解码结果中一直到第4位的’0’开始正常得到解码结果，随后就可以正常地连续解码；

b)原始数据是一个’00开头的序列，实际解码结果中一直到第3位的’1’开始正常得到解码结果，随后就可以正常地连续解码；

c)原始数据从第2位开始就发生了数据的变化，从这开始就可以正常地连续解码。

总得来说，就可以看到如之前得分析，解码电路无法识别初始连续的0或1序列，当输入第一次发生变化时，解码电路才开始正常工作。为了方便区分，表中的结果用空格进行了分割，解码结果均正确无误。下图展示了仿真的时序过程，可以看到电路如预期方式正常工作。

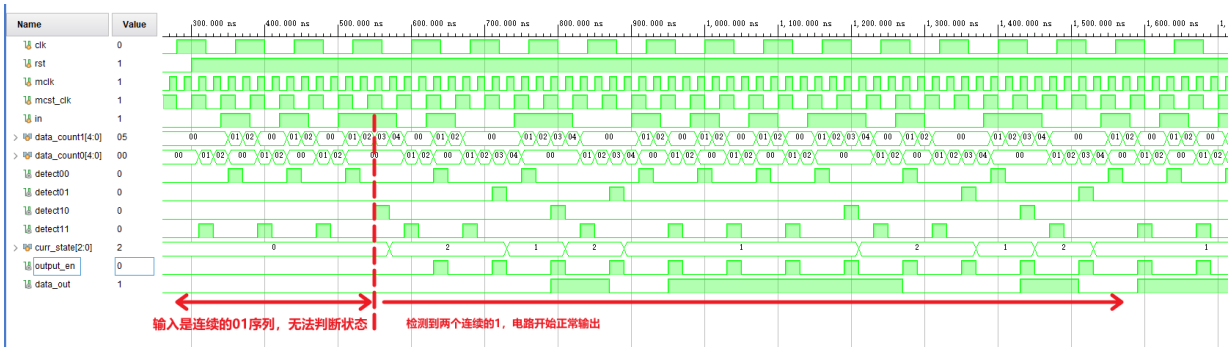

图3.1 曼彻斯特解码时序图

附录

仿真激励文件 manchester_coded_stream_decoding_sim_tb.v

```
`timescale 1ns / 1ps


module manchester_sim_tb();

reg clk;
reg rst;
reg mclk;
reg mcst_clk;
reg in;
reg [7:0] input_code;
reg [15:0] manchester_stream;


initial begin
    rst = 0;
    clk = 0;
    mclk = 0;
    mcst_clk = 0;
    input_code = 8'b11100101;
    in = 0;
    #300;
    rst = 1;
    manchester_generate(input_code);
    $display("input code is a repeat stream of %b", input_code);
    $display("manchester coded stream is %b", manchester_stream);
    repeat(3) begin
    manchester_generate(input_code);
    input_generate(manchester_stream);
    end
    $display("decoding result is %b", output_serial);
    #100 $stop;
end

always #20 mcst_clk = ~mcst_clk;
always #40 clk = ~clk;
always #10 mclk = ~mclk;


task manchester_generate;
    input [7:0] data;
    repeat(8)
    begin
        if (data[7] == 'b1)
        begin
            data = data << 1;
            manchester_stream = manchester_stream << 2;
            manchester_stream[1:0] = 2'b01;
        end
        else
        begin
            data = data << 1;
            manchester_stream = manchester_stream << 2;
            manchester_stream[1:0] = 2'b10;
        end
```

```
    end
endtask


task input_generate;
    input [15:0] data;

    repeat(16)
    begin
    @(posedge mcst_clk)begin
        in = manchester_stream[15];
        manchester_stream = manchester_stream<<1;
        end
    end
endtask


wire output_en;
wire data_out;

manchester_decoder man_u(
    .input_data(in),
    .sys_clk(mclk),
    .sys_rstn(rst),

    .output_en(output_en),
    .data_out(data_out)
);

reg [15:0] output_serial;
reg [5:0] output_count = 0;

always @(negedge output_en)
begin
    if(output_count < 16 && rst == 1)
    begin
        output_serial = output_serial << 1;
        output_serial[0] = data_out;
//      $display("output_count is %b, output_data is %b", output_count, data_out);
        output_count = output_count + 1;
    end
end

endmodule
```

manchester_decoder.v

```
1.  `timescale 1ns / 1ps
2.  //////////////////////////////////////////////////////////////////////////////////
3.  // Company: novosns
4.  // Engineer: jianan.shi
5.  //
6.  // Create Date: 2024/11/24 20:38:28
7.  // Module Name: manchester_decoder
8.  //
9.  // Revision 0.01 - File Created
10. // Additional Comments: For Manchester encoding adc
11. //
12. //////////////////////////////////////////////////////////////////////////////////
13.
14.
15. module manchester_decoder(
16.     input input_data,
17.     input sys_clk,
18.     input sys_rstn,
19.
20.     output reg output_en,
21.     output reg data_out
22.
23.     );
24.
25. // synchronize sampling data input
26. always @(posedge sys_clk or negedge sys_rstn)
27.     begin
28.         if(!sys_rstn)
29.         begin
30.             m_data0 <= 'b0;
31.             m_data1 <= 'b0;
32.         end
33.         else
34.         begin
35.             m_data0 <= input_data;
36.             m_data1 <= m_data0;
37.         end
38.     end
39.
40. // count the number of 1s and 0s
41. always @(posedge sys_clk or negedge sys_rstn)
42.     begin
43.         if(!sys_rstn)
44.         begin
45.             data_count1 <= 'd0;
46.             data_count0 <= 'd0;
47.         end
48.         else
49.         begin
50.             if(input_data == 'b1)
51.             begin
52.                 data_count1 <= data_count1 + 'b1;
53.                 data_count0 <= 'b0;
54.             end
55.             else
56.             begin
57.                 data_count0 <= data_count0 + 'b1;
58.                 data_count1 <= 'b0;
59.             end
60.         end
```

```
    end

wire detect00;
wire detect01;
wire detect10;
wire detect11;

// detect the serial pattern of input bitstream
assign detect00 = ((data_count0 == 0) && (data_count1 == 1)) ? 'b1 : 'b0;
assign detect01 = ((data_count0 == 3) && (data_count1 == 0)) ? 'b1 : 'b0;
assign detect10 = ((data_count0 == 0) && (data_count1 == 3)) ? 'b1 : 'b0;
assign detect11 = ((data_count0 == 1) && (data_count1 == 0)) ? 'b1 : 'b0;

// state machine of decoding state machine
reg [2:0] curr_state;
reg [2:0] next_state;

parameter s0_initial    = 3'b000;
parameter s1_read1      = 3'b001;
parameter s2_read0      = 3'b010;
parameter s3_end        = 3'b011;

reg [4:0] data_count1;
reg [4:0] data_count0;

reg m_data0;
reg m_data1;

reg output_flip0;
reg output_flip1;

always @(posedge sys_clk or negedge sys_rstn)
begin
    if(!sys_rstn)
        curr_state <= s0_initial;
    else
        curr_state <= next_state;
end

always @(*)
begin
    case(curr_state)
    s0_initial:
    begin
        if(detect01)
            next_state = s1_read1;
        else if(detect10)
            next_state = s2_read0;
        else
            next_state = s0_initial;
    end

    s1_read1:
    begin
        if(detect11)
            next_state = s1_read1;
        else if(detect10)
            next_state = s2_read0;
        else
            next_state = s1_read1;
    end

    s2_read0:
    begin
```

```
125.        if(detect01)
126.            next_state = s1_read1;
127.        else if(detect00)
128.            next_state = s2_read0;
129.        else
130.            next_state = s2_read0;
131.    end
132.
133.    default:
134.        next_state = s1_read1;
135.
136.    endcase
137. end
138.
139. always @(*)
140. begin
141.    case(curr_state)
142. //    s0_initial:
143. //    begin
144. //        if(detect01 || detect10)
145. //            output_en = 'b1;
146. //        else
147. //            output_en = 'b0;
148. //    end
149.    s1_read1:
150.    begin
151.        if(detect11 || detect10)
152.            output_en = 'b1;
153.        else
154.            output_en = 'b0;
155.    end
156.
157.    s2_read0:
158.    begin
159.        if(detect01 || detect00)
160.            output_en = 'b1;
161.        else
162.            output_en = 'b0;
163.    end
164.
165.    default:
166.        output_en = 'b0;
167.
168.    endcase
169. end
170.
171. always @(posedge output_en or negedge sys_rstn)
172. begin
173.    if(!sys_rstn)
174.        data_out <= 'b0;
175.    else
176.    begin
177.        data_out <= (curr_state == s1_read1) ? 'b1 : 'b0;
178.    end
179. end
180.
181. endmodule
```

4.修订历史

版本	描述	作者	日期
1.0	创建应用笔记	Jianan Shi	2024/12/3

销售联系方式：sales@novosns.com;　获取更多信息：www.novosns.com