北京航空航天大學

BEIHANG UNIVERSITY

实验四、应用设计——曼彻斯特码检测器

2018 年 12 月 05 日（版本 v01）

网络空间安全学院

目录

实验四、应用设计——曼彻斯特码检测器（实验指导书部分） 2

1 实验指导 2

1.1 背景知识 2

1.1.1 曼彻斯特码检测器的主要功能 3

1.1.2 自启动设计与软件功能仿真 4

1.1.3 曼彻斯特码检测过程中的问题 5

1.1.4 参考 MIL-STD-1553B 总线系统中的前导字 5

1.2 实验要求 6

1.2.1 曼彻斯特码检测器核心电路 6

1.2.2 前导字捕获电路 7

1.2.3 调试用曼彻斯特码序列信号生成电路 7

1.2.4 调试用曼彻斯特码序列数据设置程序 8

1.2.5 调试用曼彻斯特码检测器人机接口 8

1.4 考核说明 9

1.5 其它说明 9

实验四、应用设计——曼彻斯特码检测器（实验报告部分） 10

2 实验报告 10

2.1 需求分析 10

2.2 概要设计 10

2.3 详细设计 11

2.3.1 核心电路设计 11

2.3.2 含前导字捕获电路的检测器 13

2.3.3 序列信号生成电路 14

2.3.4 序列信号设置程序 15

2.3.5 人机接口设计 17

2.3 功能仿真测试 19

2.4 小结 21

实验四、应用设计——曼彻斯特码检测器（实验指导书部分）

1 实验指导

曼彻斯特码是通过跳变表示“0”、“1”信息的基带编码。设计一种曼彻斯特码检测器，能够把输入的曼彻斯特码还原为普通二进制编码数据；首先对于检测器中的核心组件进行分析、仿真（考察同步时序逻辑电路分析、状态机设计等知识点）；其次，特别考虑曼彻斯特码的起始字的捕获方法，以及捕获后合理地转入码转换操作的过程（考察序列滤波知识点，以及解决实际应用问题的能力）；随后，为了进行硬件调试，建议采用编程的方法生成调试用的序列编码（考察灵活使用各种开发工具的能力），生成相应的波形（考察序列生成知识点）；最后，设计人机接口合理显示代码转换的结果（考察应用设计系统集成的能力）。

1.1 背景知识

从一个数字电路源端模块向另一个数字电路目的端模块传递串行数据信号，如果不附加时钟信号，接收端根据已知的数据速率进行采样和整形（例如：RS-232 串口通信），但目的端和源端时钟的误差会造成不可靠的风险；如果附加时钟信号，不仅需要增加电缆条数，而且时钟信号和串行数据信号之间的偏差（skew）必须被控制在可接受的范围之内。

为了克服上述两种方案的缺点，解决的思路是将时钟信号和串行数据信号合并起来传送，在目的端同时恢复数据和数据脉冲（strobe）。曼彻斯特码就是其中最简单的解决方案。

曼彻斯特码的编码规则为，每个二进制码元对应一个符号，该符号的中心含有从高电平到低电平或低电平到高电平的跳变。

如果不进行曼彻斯特编码，串行传递地大段的“0”或大段的“1”的时候会造成目的端对于源端的时钟信号失去锁定。而曼彻斯特编码保证不论传递“0”、“1”码元，都含有频率为数据率 2 倍的频率分量。

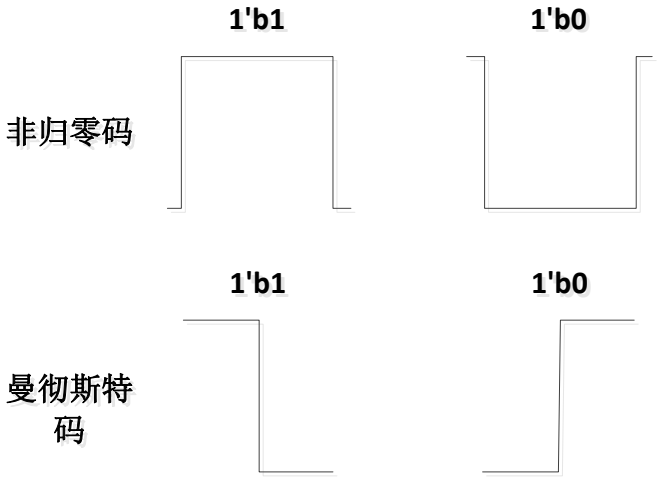

图 1 曼彻斯特码编码规则

习惯上，二进制“0”对应的曼彻斯特码先为低电平，然后跳变为高电平；而二进制“1”对应的曼彻斯特码先为高电平，然后跳变为低电平，如图 1 所示。

值得说明的是，曼彻斯特码占据的数据带宽为源数据速率的 2 倍，在宽带数据传输中，这种带宽资源的浪费也是不可容忍的，因此采用线路码型 4B/5B 或 8B/10B 编码，可以在保留串行信号时钟信息的条件下使波特率仅为码速率的 1.2 倍，但这两种解决方案的时钟恢复和时钟相位对准更加复杂一些，可使用锁相环电路进行时钟频率的提取，并通过特殊的前导字（preamble）设计达到符号同步，进而达到码元和时钟的同步。

1.1.1 曼彻斯特码检测器的主要功能

曼彻斯特码检测器首先要能检测电平的变化，确定电平跳变的边沿，在电平跳变之后采样，如果采样值为低电平，则判决为“1”，反之如果采样值为高电平，则判决为“0”。

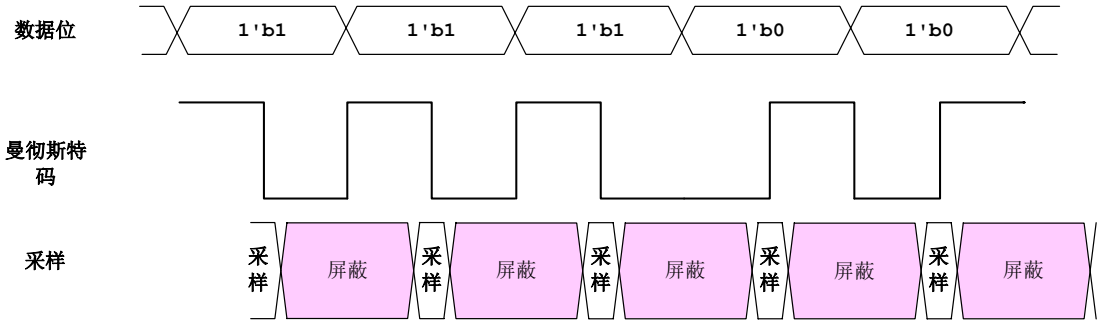

图 2 曼彻斯特码检测器的采样判决

然而对于曼彻斯特码，连续的 0 或者连续的 1 之间的（曼彻斯特码的符号之间）也会形成跳变，因此必须在作出有效的“1”或“0”判决之后，使检测器“屏蔽”一段时间，直到保证越过符号之间的电平变化，如图 2 所示。

另外，曼彻斯特码检测器还应输出数据脉冲（strobe）和串行数据（data），以保证在目的端可靠地获

得有同步于数据脉冲的可靠的采样数据。

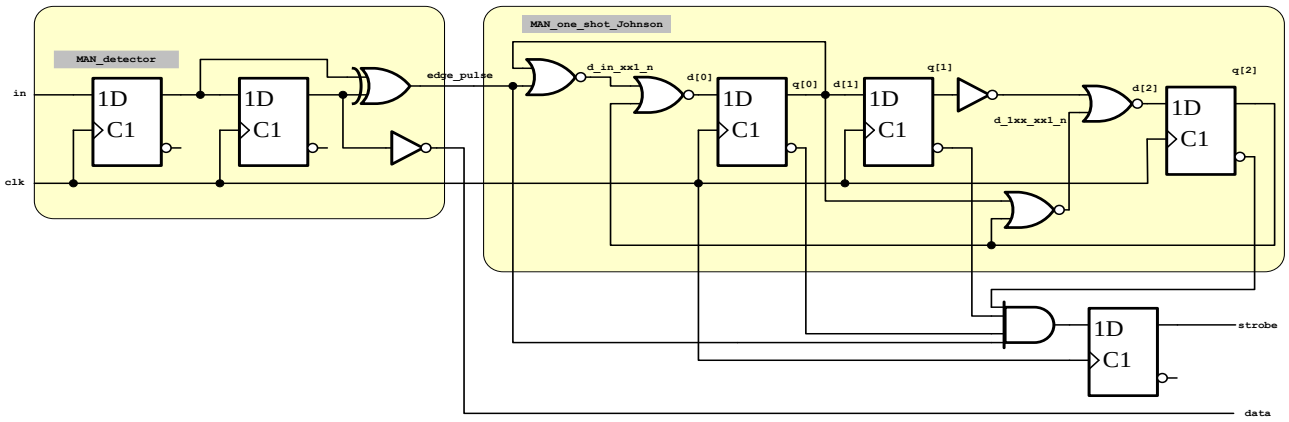

图 3 曼彻斯特码检测器电路原理图

图 3 展示了一种曼彻斯特检测器的电路实现，它包含边沿检测部分（MAN_detector 模块）和一个状态机（MAN_one_shot_Johnson 模块）。其中后者从 3 位约翰逊计数器（又叫作扭环计数器）改造而成，其状态转换图如图 4 所示。

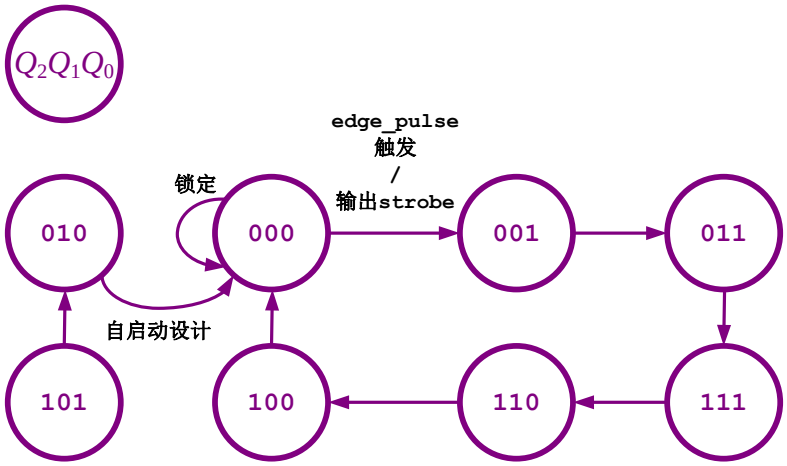

图 4 曼彻斯特码检测器中 MAN_one_shot_Johnson 的状态机

注：提供已经编写好的 MAN_detector 模块和 MAN_one_shot_Johnson 模块，以供参考。注意到：码元持续时间为采样时钟的周期的整倍数，这里倍数为 8。

1.1.2 自启动设计与软件功能仿真

曼彻斯特码检测器中 MAN_one_shot_Johnson 的状态机具有自启动的功能，所以在应用中不需要 rst_n 等复位信号。但功能仿真器往往将状态机的初始状态设置为未知（“3’bxxx”），导致仿真无法正常运行。因此在模块中添加一个 rst_state_n 的复位信号，满足软件功能仿真的要求。

而在模块真正实例化并硬件实现的时候，去除与有关的代码，或者使复位信号 rst_state_n 无效（如下所示的代码行），经过检验可以正常工作。

```
MAN_decoder uut_MAN_decoder ( .clk( clk ),
  .in( serial_data ), .data( data ) , .strobe( strobe ),
  .rst_state_n( 1'b1 ) );    /* self reset func.*/
```

1.1.3 曼彻斯特码检测过程中的问题

曼彻斯特码的每个符号之中都有跳变，而相同的符号之间有跳变，而且在第一个符号到达之前，如果串行信号一直保持高电平或低电平，则要么第一个跳变出现在符号之中（设计期望的结果），要么出现在符号的边界处。例如：如果信道休眠的时候为高电平，两种跳变的情况如图 5。

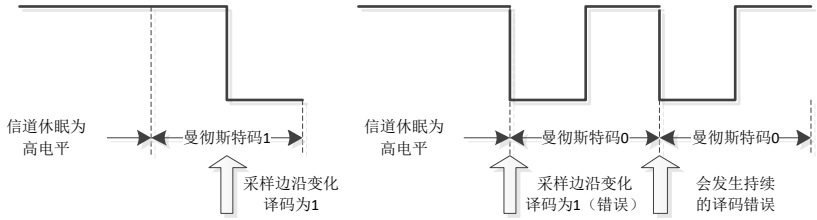

图 5 在符号起始的时候有可能发生译码错误

由于曼彻斯特码检测器本身无法识别何为有效地跳变，会造成暂时的或一长串的译码错误。例如（如所示，注意图中为低位串行先输出）：如果信道休眠的时候为高电平，如果数据码元为 8’b1011_1101（8’hBD，低位先串行输出），则经过曼彻斯特编码后再译码，会正确恢复原数据；但如果数据码元为 8’b0100_0010（8’h42，低位先串行输出），则经过曼彻斯特编码后再译码，恢复的数据为 8’b0100_0011（8’h43）。

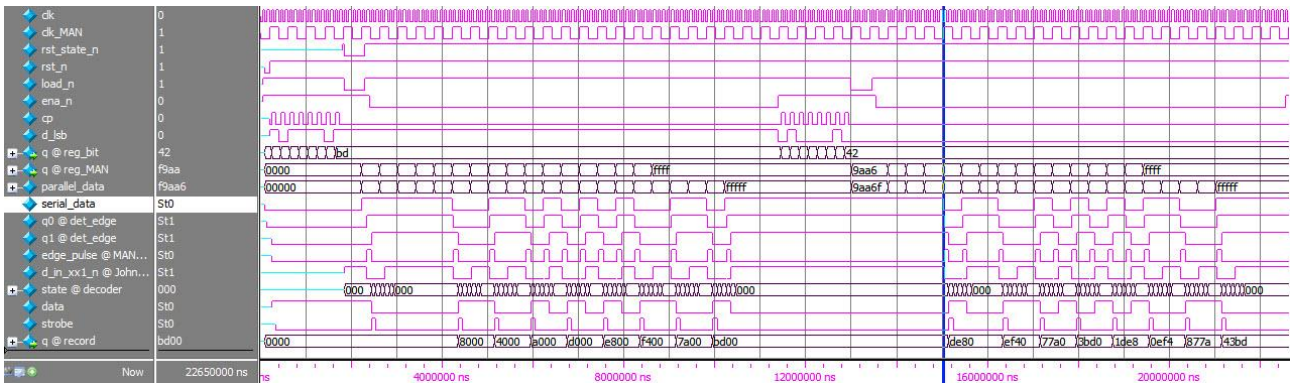

图 6 在符号起始的时候有可能发生译码错误（软件功能仿真）

为了克服这个问题，实用中往往在曼彻斯特码的头部添加前导字（preamble）。

1.1.4 参考 MIL-STD-1553B 总线系统中的前导字

在 F-16 战机上，使用了 MIL-STD-1553B 标准总线将自动驾驶、大气机、显示仪表等部件综合化地连接起来，就采用了曼彻斯特 II 型线路码形，而它的线路帧结构（以数据帧为例）如图 7 所示。一个完整的数据帧由前导字（标准上称为“Sync”）、数据字和奇校验位组成。

前导字的宽度为 3 个比特码元宽度，其中低电平和高电平各占一半（即：各占 1.5 个码元周期），数据帧的前导字是先为低电平，后为高电平。

由于曼彻斯特码的高、低电平各占 0.5 个码元周期，所以不论前导字后面紧接着是“0”的编码，还是“1”的编码，都能分辨出这个前导字符号，进而能够判明后续曼彻斯特码正确的跳变时刻。

由于两个相邻的不同比特（如：数据“10”或数据“01”）之间产生的脉冲宽度为 1 个码元周期，在合理地设计下 1 个码元周期和 1.5 个码元周期是可以分辨的，因此不会将传输中数据字的某段中误认为前导字。

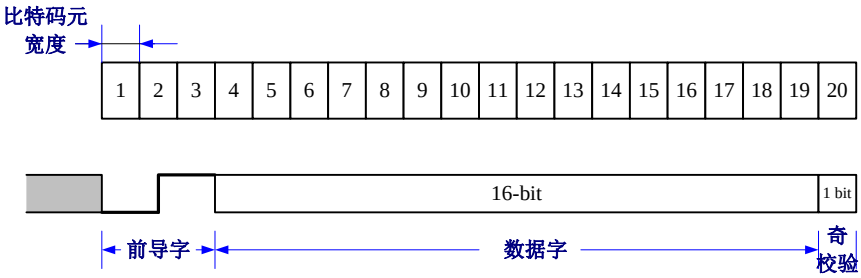

图 7 前导字

注：确切地说 MIL-STD-1553B 标准总线采用带有正、负、地的三电平信号，这里为了便于实验的讲解，只考虑高、低电平，叙述的和真实的总线信号不同，特此说明。

在前导字之前，可以是高电平，也可以是低电平，甚至有的系统为了建立码元同步（从曼彻斯特码中提取远程时钟相位），不发送数据的时候也会持续发送或定期发送一长串“111…11”等连续曼彻斯特编码，但奇妙的是，只要前导字前面不存在和它脉冲宽度相近的干扰，不论之前是持续的高、低电平或是曼彻斯特码连续变化的跳变，该前导字均能有效地标示数据字的起始。

更奇妙的是，在一定倍频的时钟采样下，仅依靠本课程在“同步时序逻辑电路设计”中的知识，即可方便地捕获该同步头。

1.2 实验要求

1.2.1 曼彻斯特码检测器核心电路（简称“核心电路”）

a) 针对边沿检测模块（MAN_detector）和状态机模块（MAN_one_shot_Johnson），进行时序逻辑电路分析，在《实验报告》中写出逻辑分析结果，判定电路功能；

b) 编写测试用例（testbench, test cases），对 MAN_detector 模块和 MAN_one_shot_Johnson 模块进行功能仿真测试，建议能够仿真出由于没有前导字而造成的码转换错误的情况；

c) 采用完全行为描述（即：不涉及触发器、门电路，但要求必须也是可综合的代码）的方法用

Verilog HDL 语言编写曼彻斯特码检测器核心电路模块，并用同样的测试用例进行仿真测试。

1.2.2 前导字捕获电路（简称“捕获电路”）

d) 仿照 MIL-STD-1553B 标准总线的数据帧格式，设置前导字的形状，针对该形状的前导字，用 Verilog HDL 语言编写前导字捕获电路模块；（提示：在时序逻辑电路课程中讲授过）

e) 编写测试用例（testbench, test cases），对前导字捕获电路模块进行功能仿真测试，要求在仿真测试中要体现采样时钟与码元周期边界不同步的一般情况；

f) 前导字捕获电路模块要充分避免错误的判决功能，例如图 8，要求在前导字之后必须跟随有效的曼彻斯特码，才能切换到码检测状态，否则，则认为当前不是有效的前导字，仍然进行捕获；

g) 将前导字捕获电路模块的实例与前述核心模块电路的实例相互组装，构成考虑前导字的曼彻斯特码检测器。

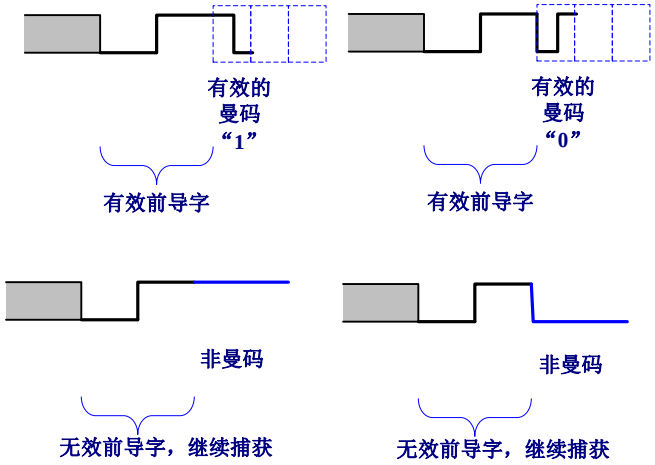

图 8 捕获后必须跟随有效的曼彻斯特码才为有效前导字

1.2.3 调试用曼彻斯特码序列信号生成电路（简称“序列信号生成”）

h) 为了实现硬件的测试，采用定义一个数组（如：reg ROM [220]）作为存储器，以 0、1 表示输出电平，由于要构造出与采样时钟相位不同的波形，建议使数组中的每个单元，仅用于表示码元持续时间的 1/10 时段内的逻辑电平（即：每连续 10 个比特，才表示 1 个码元持续时间内的波形）；

注：之所以例子中给出 220 个记录，是由于数据帧（含前导字和奇校验位）共 20 个码元持续时间，但相位如果在一个码元持续时间内变化，则数据帧的前后可以有无关的数据（其实前后无关的数据之和为一个码元持续时间，调试就可以正常进行，这里只是略微使存储资源富裕

一些）。

i) 设计并实现序列信号生成电路模块，在一定的时钟触发下连续输出存储的电平信号，并回答此时时钟的频率与曼彻斯特检测器的采样时钟频率的关系；

j) 该序列信号生成电路模块要求可综合，需进行实现和必要的调试。

1.2.4 调试用曼彻斯特码序列数据设置程序（简称“序列数据设置”）

k) 由于对数组赋值的时候不能像比特矢量那样对连续的几位进行（请查询 Veilog HDL 语言的资料，在《实验报告》中把该现象描述清楚），因此手工填写数据赋值代码非常不便；

l) 请用 C 语言（或其它语言，不限）编写一段序列数据设置程序，当用户给定 16 比特数据、初始相位后，能够自动生成 reg ROM [220]中的序列数据，并**必须**能够输出文本文件格式的 Verilog HDL 代码段，如字符串打印出“ROM [0] = 1’b1 ;”等，能够直接被“剪切+复制”到*.v 源代码中；

m) 毕竟使用数组时只能对一个元素进行操作很不方便，但比特矢量却不能利用变量性质的索引表示，请问有没有折衷的解决方案？此要求可以不用电路实现，仅在《实验报告》中说明即可。（提示：在时序逻辑电路课程中讲授过）

1.2.5 调试用曼彻斯特码检测器人机接口（简称“人机接口”）

n) 人机接口需求 1——用开关 Switch[0]的状态表示 MODE_RUN 和 MODE_DEMO，

- ◆ 在 MODE_RUN 状态，按钮复位键 Sys_RST 和按钮 Key[0]可用，Sys_RST 使检测器和调试用序列信号发生器恢复到初始设置，此时如果按动 Key[0]，则序列信号输出到检测器模块，检测器模块输出还原的数据并存储起来；
- ◆ 在 MODE_DEMO 状态，按钮 Key[1]可用，此时如果按动 Key[1]，通过七段数码管 SEG[0]~SEG[6]依次显示解码后的数据，为了便于观察，显示速度降至肉眼可以分辨的范围；

o) 人机接口需求 2——用 4 个 LED 灯表示整个电路系统所处的状态，如：LED[0]点亮表示系统复位就绪、LED[1]点亮表示进行完一次曼彻斯特码检测、LED[2]点亮表示正在进行解码后数据的输出展示、LED[3]点亮表示演示结束；

p) 人机接口需求 3——用开关 Switch[1]的状态表示 MODE_DISP_DATA 和 MODE_DISP_CNT，在 MODE_DISP_DATA 状态，数码管依次显示解码还原的数据，在 MODE_DISP_CNT 状态，数码管依次显示解码还原数据的序号。

1.4 考核说明

本实验将采取**实验报告和现场汇报演示相结合**的形式，实验报告上交的具体**截止期限**请注意任课教师的通知，实验报告格式请参考本文档；**现场汇报演示**的时间和分组情况请注意任课老师的通知。特此广而告之。

1.5 其它说明

作为一项应用设计，从实战出发不设置“基本要求”和“扩展要求”，而是从完成度的角度进行评价。根据知识点或技能的重要程度，以及工作量上的“边际效应”，将实验要求中的各项内容的权值设置如下：

	实验要求	权重（100%）	完成度自评（100%）
1	核心电路	25%	
2	捕获电路	25%	
3	序列信号生成	25%	
4	序列数据设置	5%	
5	人机接口	20%	

所有小组将慎重地将在《实验报告》中的总结部分填写自评结果，现场演示的时候需要公布自评的情况，并有意识地通过演示中的证据支持自评结果，说明应用设计完成的成熟度。

实验四、应用设计——曼彻斯特码检测器（实验报告部分）

2 实验报告

2.1 需求分析

在核心电路模块中，需要对所给的代码按要求进行分析，并且自行编写 RTL 级曼彻斯特码解码器。捕获电路是通过状态机实现的，状态机分为等待、高电平、低电平和接收数据四个状态，可以捕获前导字并且检验数据帧的校验位。再使用其他语言编写序列数据设置模块，这里用了较为容易的 Python 实现，在生成模块中建立起一个大小为 220 的寄存器数组，并在分频时钟信号触发下输出储存好的曼彻斯特编码序列给解码器。人机接口模块则利用之前的知识按要求实现即可。

2.2 概要设计

系统整体设计如图 9 所示。Top 模块输入去抖后的按键信号，将需要显示的状态和数字输出到相应显示模块，除此之外，将系统时钟和使能信号传递给序列信号生成模块，序列信号生成模块输出预先存储在寄存器数组中的数据以及分频后的时钟，传递给带有前导字捕获电路的曼彻斯特码解码器，这里系统使用了自己编写的 RTL 级解码器，最后解码器将解码后的数据以及数据脉冲信号传给 Top 模块，存在另一个寄存器数组中。

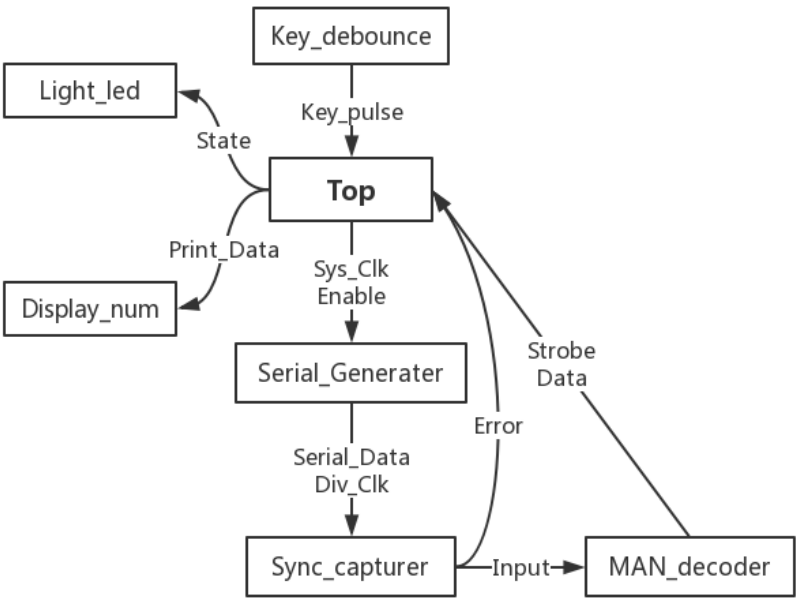

图 9 整体设计图

2.3 详细设计

2.3.1 核心电路设计

首先对提供的边沿检测模块和状态机模块进行时序逻辑电路分析：

对于 MAN_detector 模块，根据时序逻辑电路的分析方法，写出每个触发器的驱动方程

$$\begin{cases} D_1 = in \\ D_2 = Q_1 \end{cases}$$

其中Q_1为第一个触发器的输出，将上式代入 D 触发器的特性方程$Q^* = D$，于是得到电路的状态方程

$$\begin{cases} Q_1^* = in \\ Q_2^* = Q_1 \end{cases}$$

根据逻辑图写出输出方程

$$\begin{cases} edge\_pulse = Q_1 \oplus Q_2 \\ data = \overline{Q_2} \end{cases}$$

则根据输出方程，可以知道 edge_pulse 是电平跳变的判断信号。

对于 MAN_one_shot_Johnson 模块，写出每个触发器的驱动方程

$$d[0] = \overline{d\_in\_xx1\_n + q[2]}$$

$$d[1] = q[0]$$

$$d[2] = \overline{d\_1xx\_xx1\_n + \overline{q[1]}}$$

$$d\_strobe = \overline{edge\_pulse \cdot \overline{q[0]} \cdot \overline{q[1]} \cdot \overline{q[2]}}$$

将上式代入 D 触发器的特性方程$Q^* = D$，于是得到电路的状态方程

$$q[0]^* = \overline{\overline{edge\_pulse + q[0]} \cdot q[2]}$$

$$q[1]^* = q[0]$$

$$q[2]^* = \overline{\overline{q[1]} + \overline{q[2] + q[0]}}$$

$$d\_strobe^* = \overline{edge\_pulse \cdot \overline{q[0]} \cdot \overline{q[1]} \cdot \overline{q[2]}}$$

根据逻辑图写出输出方程

$$strobe = \overline{edge\_pulse \cdot \overline{q[0]} \cdot \overline{q[1]} \cdot \overline{q[2]}}$$

对于这种较为复杂的时序逻辑电路，不便于直接进行逻辑分析，则我们使用波形图来分析。下面是 ISE 下 edge_pulse 作为输入的波形图

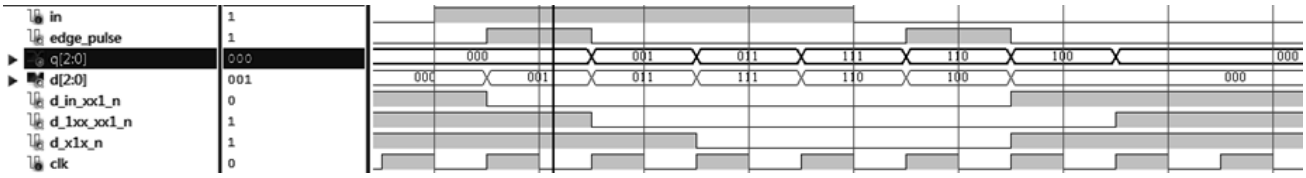

图 10 门级解码器仿真波形

可以看出来，当 edge_pulse 变为高电平且状态为初始状态时，整个模块开始变化，经过一个时钟信号后，信号传到第一个 D 触发器，而之后由于 q[0]、q[1]和 q[2]在不同时期至少一个为 1，edge_pulse 的改变不会影响 strobe，直到第六个时钟信号之后，又重新回到初始状态等待 edge_pulse 输入信号的改变。而只有在初始状态时，3 个 D 触发器都为 0，edge_pulse 的变化会影响到 strobe 信号。综上可以得出，这两个模块就是用来获取曼切斯特编码的序列信号和数据脉冲。

然后按照图 3 将两个模块拼装成一个完整的曼彻斯特码解码器：

```
not gaten1(nq[0], q[0]);
not gaten2(nq[1], q[1]);
not gaten3(nq[2], q[2]);
and gatea1(raise, nq[0], nq[1], nq[2], edge_pulse);

D_FF FF(strobe, clk, raise, rst);

MAN_detector man_detector (
   .clk(clk),
   .in(in),
   .data(data),
   .edge_pulse(edge_pulse),
   .rst_state_n(1'b1)
);

MAN_one_shot_Johnson man_one_shot_johnson (
   .clk(clk),
   .edge_pulse(edge_pulse),
   .q(q),
   .rst_state_n(1'b1)
);
```

在将门级实现的解码器电路烧录到实验板测试时候去掉了 reset 信号，我们观察到解码器仍然可以正常工作，说明该状态机可以自启动。

接着我们使用 RTL 级编写了功能相同的曼彻斯特吗解码器，利用一个寄存器来判断信号的跳变，并且保证 strobe 信号仅有一个时钟周期为高电平，模块核心代码如下：

```
if (strobe)
   strobe <= 1'b0;
```

```
if (in != old_in) begin
   old_in <= in;
   if (counter == 3'd0) begin
      strobe <= 1'b1;
      data <= ~in;
      counter <= counter + 1;
   end
end
else begin
   if (counter == 3'd0 || counter == 3'd5)
      counter <= 3'd0;
   else
      counter <= counter + 1;
end
```

2.3.2 含前导字捕获电路的检测器

我们设计的前导字形状与图 7 相同，即 1.5 个码元周期的低电平与 1.5 个码元周期的高电平，紧接着必须跟有效的曼彻斯特码才可以进入解码状态，经过对功能需求的仔细分析，我们决定使用一个状态机来描述前导字捕获电路。

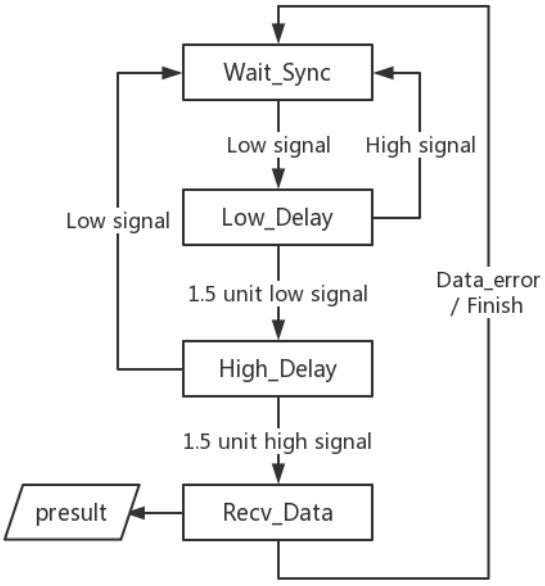

图 11 前导字捕获电路状态机

该状态机可以捕获对应形状的前导字，捕获后进入解码的状态，如果前导字后跟一个正确的曼彻斯特码才可以进入解码状态，所以我们在半个码元周期后检测输入信号是否有跳变，若存在跳变即为正确的曼彻斯特码，否则返回初始状态等待下一个前导字信号：

```
if (cnt < 4'd15) begin
   old_in <= in;
```

```
    if (cnt == 4'd3) begin
       if (in == old_in)
          State <= Wait_Sync;
    end
    cnt <= cnt + 1;
end
```

由于解码状态的寄存器编码是 2'b11，所以可以直接用“与”操作来处理解码器的输出，形成带有前导字捕获电路的曼彻斯特码解码器，其中 raw_strobe 和 raw_data 为不带前导字的解码器输出信号。

```
assign strobe = raw_strobe & State[0] & State[1];
assign data = raw_data & State[0] & State[1];
```

2.3.3 序列信号生成电路

序列信号生成电路模块需要在一定的时钟触发下输出连续的序列信号，不妨将其频率设为f_{data}，同时设分频时钟频率为f_{div}。模块内寄存器数组的每个单元仅用于表示一个码元持续时间的$\frac{1}{10}$时段内的逻辑电平，而每个单元的时间为$T_{data}=\frac{1}{f_{data}}$，则一个码元持续时间为 10 个单元时间，设为$T_{meta}=\frac{10}{f_{data}}$，而系统规定码元持续时间为采样时钟的周期的整倍数，这里倍数为 8，即一个码元持续时间为$T_{div}=\frac{8}{f_{div}}$。则很容易得到如下关系：

$$\frac{10}{f_{data}}=\frac{8}{f_{div}}\Leftrightarrow\frac{f_{data}}{f_{div}}=\frac{10}{8}=\frac{5}{4}$$

但是为了方便设计，我们保持它的频率作为偶数倍，即最后取

$$\frac{f_{data}}{f_{div}}=\frac{10}{8}$$

作为两个信号的频率比。

根据如上分析我们设计实现序列信号生成电路模块：

首先定义一个 220 大小的寄存器数组

```
reg ROM [0:219];
```

然后根据频率比设置两个计数器来控制输出信号

```
if (enable) begin
   if (cnt_a == 3'd7) begin // 8 sys clk
      cnt_a <= 3'd0;
      if (data_cnt < 8'd219)
         data_cnt <= data_cnt + 1;
   end
   else
      cnt_a <= cnt_a + 1;
   if (cnt_b == 3'd4) begin // 5 sys clk
```

```
        cnt_b <= 3'd0;
        div_clk <= ~div_clk;
    end
    else
        cnt_b <= cnt_b + 1;
    serial_data <= ROM[data_cnt];
end
```

2.3.4 序列信号设置程序

在对寄存器数组赋值时不能像对比特矢量一样连续几位进行赋值，例如对比特矢量，这样的赋值方式是允许的：

```
reg [3:0] vector;
vector <= 4'b0101;
```

而对于寄存器数组，只能分别对其每一个寄存器赋值：

```
reg array[0:3];
array[0] <= 1'b0; array[1] <= 1'b1; array[2] <= 1'b0; array[3] <= 1'b1;
```

我们使用 Python 编写了序列信号设置程序，可以打印出能复制到*.v 文件中的 Verilog 程序，并且存储一个 rom.patt，便于仿真时读取序列数据到寄存器数组中。

该程序分为三个部分，输出前导字，输出曼彻斯特码和输出校验位，核心代码如下：

```
for d in data:
    if(d == 0):
        for i in range(5):
            print("ROM[{:d}] <= 1'b0;".format(j))
            output.append("0\n")
            j+=1
        for i in range(6):
            print("ROM[{:d}] <= 1'b1;".format(j))
            output.append("1\n")
            j+=1
    else:
        cnt += 1
        for i in range(5):
            print("ROM[{:d}] <= 1'b1;".format(j))
            output.append("1\n")
            j+=1
        for i in range(6):
            print("ROM[{:d}] <= 1'b0;".format(j))
            output.append("0\n")
            j+=1
with open('rom.patt', 'w') as f:
```

```
    f.writelines(output)
    f.close()
```

除此之外，我们还想到了一种折衷的解决方案，既可以使用变量性质的索引，而且还可以进行连续赋值。根据要求，若是全部采用数组时，仅能对一个元素进行操作；而若是全部采用比特矢量，不能利用变量性质的索引表示。那么很自然的，我们会想起数字电路课程中所讲到的 ROM 的结构——存储器的容量=（字数）*（每个字的位数）。同样的，在 Verilog 中，位数相等于比特矢量，字数相当于数组。我们可以定义一个数组，但其每一个数组单元都是比特矢量，这样我们可以根据元素所在的数组单元进行索引访问，而对于多个数据，则找到对应的比特矢量进行数据选择即可。

以上面的 reg ROM [0:219]为例，本来需要定义一个大小为 220 的数组，若是根据 10 个数组单位表示一个码元持续时间的特性定义为 reg [9:0] ROM [0:21]，则可以设计时序电路来实现与之前相同的功能。

对于此例中的序列信号，我们可以使用数据选择器和 10 进制计数器来实现操作如上定义的存储结构。即计时器的值作为数据选择器的输入，通过数据选择器输出的常量来决定 ROM 中的比特矢量的输出，而计时器每循环一次，则把 ROM 的数组下标加 1，则可以方便的操作所有元素。

```
reg [4:0] index;
reg [3:0] counter;
reg [9:0] ROM[0:21];

always @(posedge clk or negedge rst) begin
   if (!rst) begin
      index <= 5'b0;
      counter <= 4'b0;
   end
   else begin
      if (counter == 4'd9) begin
         counter <= 4'd0;
         if (index < 5'd21)
            index <= index + 1;
      end
      else
         counter <= counter + 1;
   end
end

always @(posedge clk or negedge rst) begin
   if (!rst) begin
      data <= 1'b0;
   end
   else begin
```

```
        case (counter)
           4'd0: data <= ROM[index][0];
           4'd1: data <= ROM[index][1];
           4'd2: data <= ROM[index][2];
           4'd3: data <= ROM[index][3];
           4'd4: data <= ROM[index][4];
           4'd5: data <= ROM[index][5];
           4'd6: data <= ROM[index][6];
           4'd7: data <= ROM[index][7];
           4'd8: data <= ROM[index][8];
           4'd9: data <= ROM[index][9];
        endcase
     end
  end
```

2.3.5 人机接口设计

我们对人机接口需求进行了一部分修改，开关 Switch 信号其实并没有实际用处，因为两个 Key 信号并不冲突；其次我们将系统的状态修改为：就绪、解码中、解码完成、输出、错误五种状态，去除原先无用的几个电路状态；再次，我们还将解码后的数据和数据序号一起打印出来，可以使效果更为直观，因为我们一个数据帧只有 16 位数据，而一个数码管可以显示 16 进制的数，故两个数码管可以同时显示。

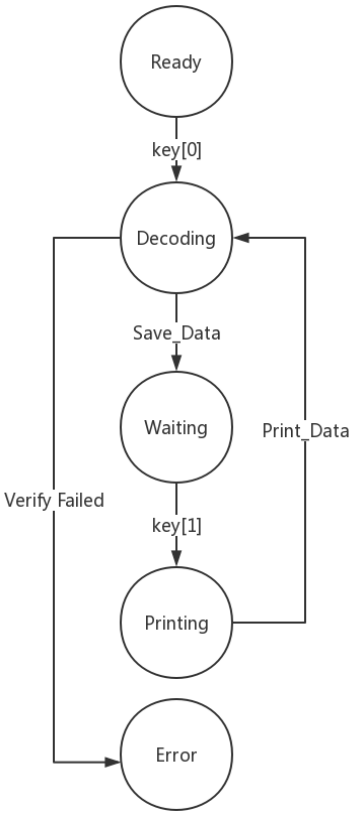

图 12 人机接口状态机

根据分析需求，我们认为人机接口也可以设置为一个状态机，状态转换图如图 12 所示，放在顶层模块中，起始为就绪状态，按动 key[0]后进入解码状态，由于解码器时钟和系统时钟不同，所以需要模拟一个检测上升沿的电路，在每个 strobe 信号的上升沿将解码器传出的数据存到解码后的寄存器中：

```
if (strobe != prestrobe) begin
   prestrobe <= strobe;
   if (strobe) begin // posedge
      if (Data_Cnt == 5'd16) begin
         Data_Cnt <= 5'd0;
         if (error)
            State <= Error;
         else
            State <= Waiting;
      end
      else begin
         Data_Reg[Data_Cnt] <= data;
         Data_Cnt <= Data_Cnt + 1;
      end
   end
end
```

解码完成后会校验数据的校验位是否正确，若不正确电路会进入错误状态，直到 Reset 信号恢复，否则进入等待状态，按动 key[1]进入打印输出状态，这时每位解码后的数据以 1 秒的间隔输出到数码管显示：

```
parameter interval = 50000000; // 1s
if (cnt == interval) begin
   cnt <= 0;
   if (Print_Cnt == 5'd16) begin
      Print_Cnt <= 5'd0;
      State <= Waiting;
   end
   else begin
      Print <= Data_Reg[Print_Cnt];
      Print_Cnt <= Print_Cnt + 1;
   end
end
else
   cnt <= cnt + 1;
```

同时，将状态机的状态数据传入 LED 显示模块，可以显示当前电路处于什么状态：

```
always @(*)
   case (Light)
      2'b00: Led <= 4'b0001;
```

```
        2'b01: Led <= 4'b0010;
        2'b10: Led <= 4'b0100;
        2'b11: Led <= 4'b1000;
    endcase
```

2.3 功能仿真测试

首先我们对门级的核心电路进行仿真测试，注意一个时钟周期为 2ns,因此一个码元周期为 16ns,这样在 testbench 中建立 8ns 低电平和 8ns 高电平表示 0，8ns 高电平和 8ns 低电平表示 1。首先仿真 011 的情况。需要使 in 分别为 011010（各延迟 8ns），通过 ISIM 仿真得到图 13。

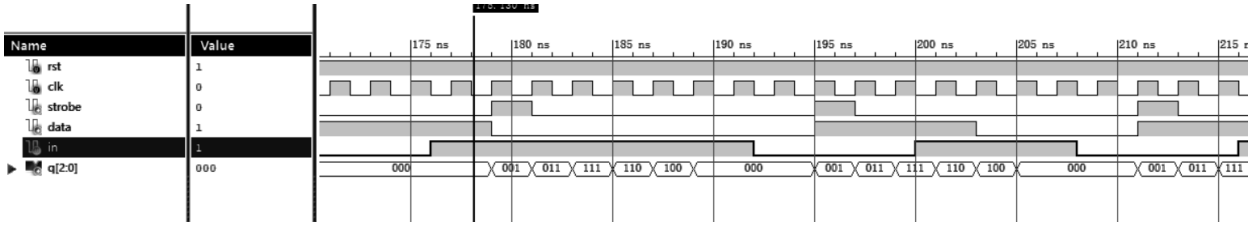

图 13 门级解码器仿真波形

从图 13 可以看到，当一个 strobe 信号到来时，data 识别到的是一个 01 形式的曼彻斯特码，输出为 0，第二个 strobe 信号时 10 形式的曼彻斯特码，输出的 data 为 1，同样的，第三个 strobe 时 data 为 1.

接下来仿真信号 100，需要将 in 设置为 100101（各延迟 8ns），再进行 ISIM 仿真，返回结果如图 14 所示。

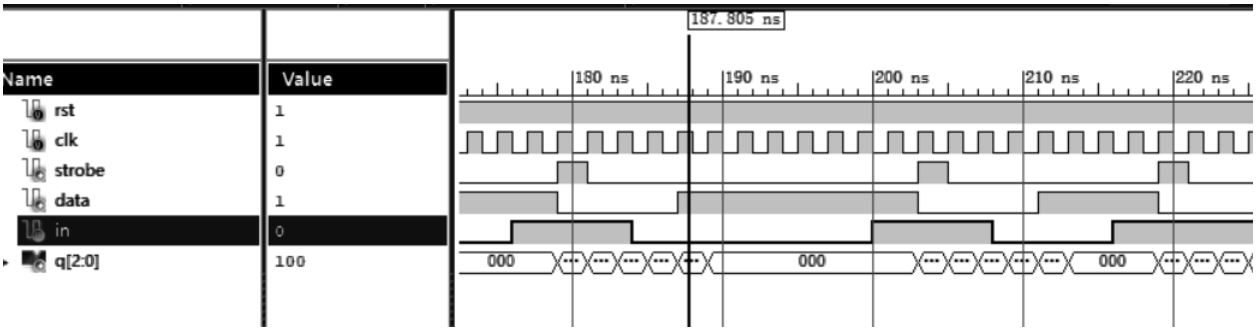

图 14 门级解码器仿真波形（无前导字造成转换错误）

发现本应该在 in 第一次变为 1 后开始检测，第一次 in 由 1 变为 0 时捕捉，进而产生 strobe 信号采样 in，但是实际上这里的检测器在 in 由 0 变为 1 时就捕捉了这个电平的跳变，然后产生了错误的 strobe,得到错误的 data。

接着对 RTL 级解码器做相同的测试，得到结果如图 15、图 16 所示，其中图 16 为无前导字造成的转换错误情况。

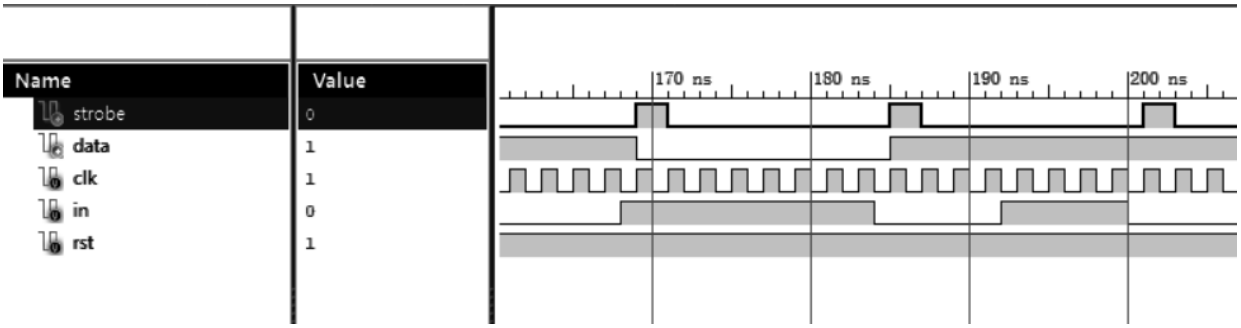

图 15 RTL 级解码器仿真波形

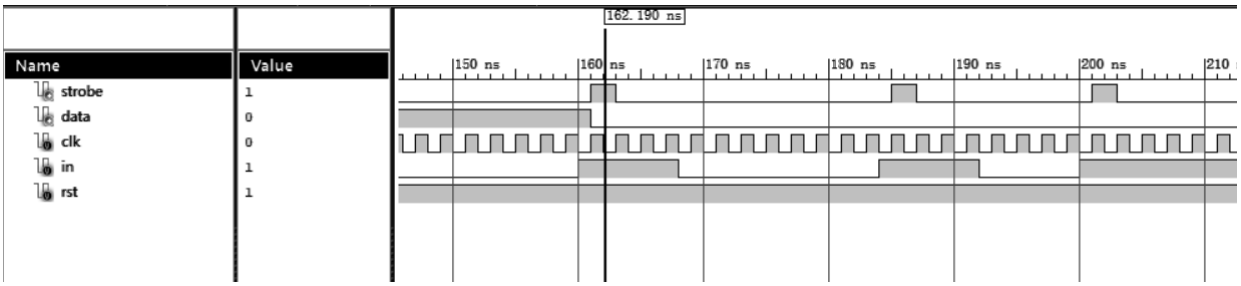

图 16 RTL 级解码器仿真波形（无前导字造成转换错误）

接下来我们对前导字捕获电路进行仿真。

首先，仿真一个前导字不正确（即高电平持续时间不足）的情况。这时候将 testbench 文件中的 in 的高电平持续时间设定为 18ns（只要小于 24ns 都可以），那么就不是一个合格的前导字，这样就不会对后面的曼彻斯特码进行译码操作。如图 17 所示。

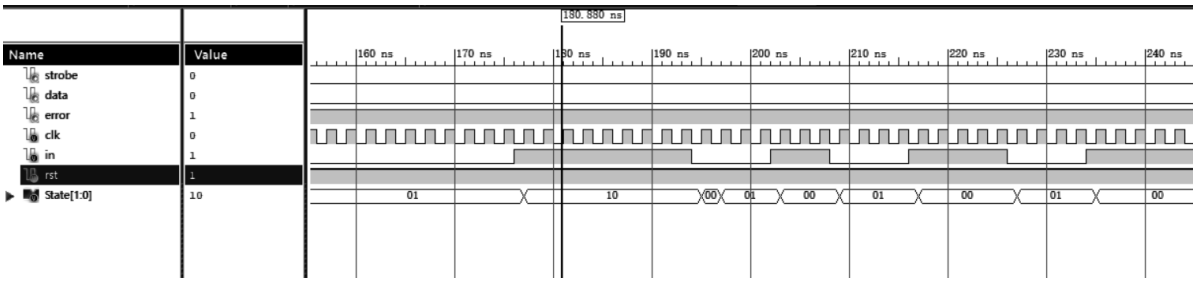

图 17 错误前导字

接下来，构建一个正确的前导字，使之能译码之后的曼彻斯特码。为了做到采样时钟与码元周期边界不同步，只需要使部分输入数据 in 的延迟时间不严格为 8ns 就好，这里我们将之设为 6ns 和 10ns。仿真之后显示的结果如图 18。这里输入的信号为 011，可以看到出现正确的 strobe 数据脉冲信号。

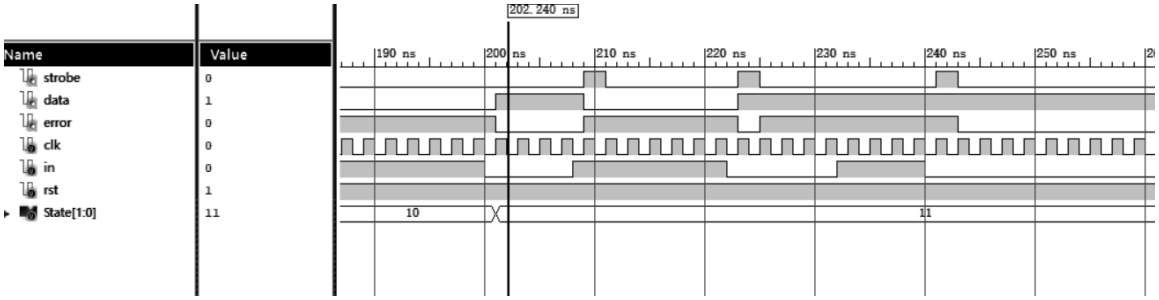

图 18 正确前导字

从图 18 总可以看出，即使采样时钟与码元周期边界不同步，也不影响对于曼彻斯特码的判断。输出的 data 仍然是 011。

完整的前导字捕获电路需要判断前导字之后是否跟随有效的曼彻斯特码，这里仿真时，在正确的前导字之后将 in 保持 32ns 的低电平，这样就是一个无效的曼彻斯特码。仿真结果如图 19 所示。

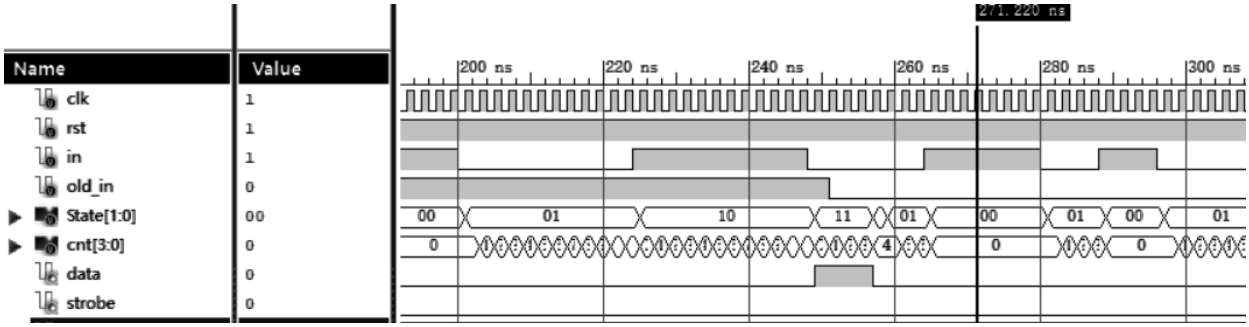

图 19 无效曼彻斯特码导致前导字无效

从图 19 可以看出，当前导字之后为无效曼彻斯特码时不会进行解码。

对序列生成模块进行仿真，在 testbench 中当 rst 为 1 的同时让 enable 使能端为 1，开始把之前存储好的 220bit 数据通过 serial_data 逐个地输出出来。如图 20 所示。

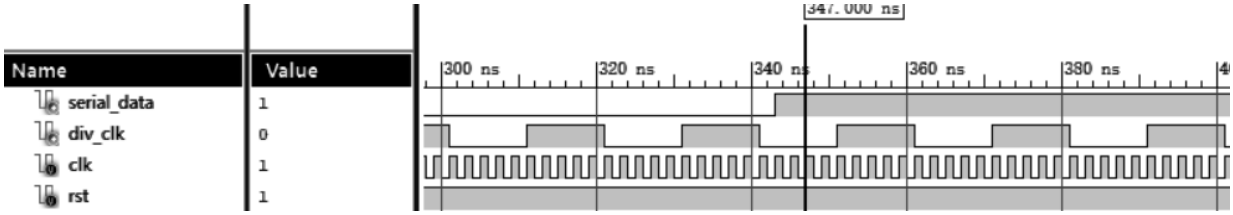

图 20 序列信号生成模块仿真波形

可以看出，此时的分频时钟和序列信号的边界并不一定是同步的。

2.4 小结

通过此次实验，我们学习到的知识有：

1. 曼彻斯特编码的定义和原理。
2. 曼彻斯特编码的解码。
3. 信息传输的基本知识。
4. 两个或多个模块的综合设计。
5. 如何合理设置 testbench 对已有模块进行仿真。

我们的以下能力得到了提升：

1. 用 Verilog 语言描述设计一个状态机。

2. 信息传输的序列滤波操作。
3. 如何用 Verilog 进行序列生成模块的开发。
4. 设计集成系统的能力。

完成度自评如下：

	实验要求	权重（100%）	完成度自评（100%）
1	核心电路	25%	25%
2	捕获电路	25%	22%
3	序列信号生成	25%	25%
4	序列数据设置	5%	4%
5	人机接口	20%	19%

附注：本次实验所用代码已经上传到 Github 仓库：

https://github.com/hiyouga/digiC-experiment/tree/master/ex4