浙江工业大学

通信系统课程设计报告

题目：HDB3 编解码系统

姓名学号 凌智城 201806061211 解码、时钟恢复、报告整合

姓名学号 林程浩 201806061210 编码、顶层连接

姓名学号 钟志鸿 201806061227 分频、M 序列产生

专业班级 通信工程 1803 班

学　　院 信息工程学院

提交日期 2021 年 9 月 25 日

报告正文

一、 设计目的

1. 了解常用的数字基带信号特征和作用；
2. 掌握 HDB3 码的编码和译码规则；
3. 熟悉 VerilogHDL 程序编写和 quartus II 及 modelsim 联合仿真；
4. 熟悉数模转换、模数转换，FPGA 芯片引脚绑定及烧写等。

二、 设计原理

1. HDB3 码为三阶高密度双极性码，是 AMI 码的一种改进型，保持了 AMI 的优点而克服了其的缺点，使连“0”个数不超过三个；
2. 编码规则如下：
   i. 先检查消息码的连“0”个数。当连“0”数目小于等于 3 时，则于 AMI 码的编码规则一样。
   ii. 当连“0”数目超过 3 个时，则将每 4 个连“0”化作一小节，用“000V”替代。（V 取值+1 或-1）应与其前一个相邻的非“0”脉冲的极性相同（因为这破坏了极性交替的规则，所以 V 称为破坏脉冲）。
   iii. 相邻的 V 码极性必须交替。当 V 码取值能满足之前的条件时但不能满足此要求时，则将“0000”用“B00V”替代。B 的取值与后面的 V 脉冲一致，用于解决此问题。因此，B 称为调节脉冲。
   iv. V 码后面的传号码极性也要交替。
3. 编码比较复杂，但解码却比较简单，从编码规则上看，每一个破坏脉冲 V 总是与前一非“0”脉冲同极性（包括 B 在内），也就是说从收到的符号序列中可以容易地找到破坏点 V，于是也判定 V 符号以及前面的三个符号必是连“0”符号，从而恢复 4 个连“0”码，再将所有-1 变成+1 后便得到原始消息码。
4. m 序列是最长线性反馈移位寄存器序列的简称。它是由线性反馈的移存器产生的周期最长的序列。在这种反馈移存器中避免出现全“0”状态，否则移存器的状态将不会改变，n 级移存器共有 2^n 种可能的状态，除去全“0”状态外，还有 2^n-1状态可用。产生 m 序列的充要条件就是移位寄存器的特征多项式为本原多项式，可根据 n 阶的 m 序列本原多项式的反复移位来产生我们需要的 m 序列。

三、 设计方案选择与经济决策

HDB3 码将 4 个连续的“0”位元取代成“000V”或“B00V”

消息代码	1	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1	1
AMI码	+1	0	0	0	0	-1	0	0	0	0	+1	-1	0	0	0	0	+1	-1
HDB3码	+1	0	0	0	+V	-1	0	0	0	-V	+1	-1	+B	0	0	+V	-1	+1

由实验箱产生的 32.768MHz 时钟，经过第一次 256 分频用于后续的码元定时恢复，继续经过 16 分频产生 8KHz 的码元时钟，由 M 序列产生模块生成随机的 M 序列来模拟传输过程中的随机信号，根据 AMI 码和 HDB3 的编码规则对 M 序列进行调整，产生单极性的 HDB3 码，但由于存在 0、+1、-1 三种情况，至少

需要两位数据才能存储，所以进行一次双极性变换，对应转化成八位数据，至此完成编码过程。

模拟数据传输过程在实验箱上用一条信号连接线来表示，用 DA 转换器将双极性的数字信号转换成模拟信号，经过信号连接线至 AD 转换器，转换成双极性的 0 和 1 输出到解码模块。由于使用 Quartus II 时没有 AD 和 DA 转换器，所以直接连接了双极性输出和输入断，跳过了模拟传输过程，但实际上的 AD 转换器输出并不是完美的双极性数据，需要根据实验箱的调整而对代码进行多次调试。

由 AD 转换器输出的八位数据，由于模拟传输过程和转化过程存在时延差，需要设计一个码元定时恢复模块，用最初 256 分频产生的时钟信号不断检测二维数据，根据实际的数据特征来产生与 16 分频信号相同的 8KHz 信号，并且判断码元的起始时刻，此处码元定时恢复模块输出的码元时钟将用于整个解码模块。解码模块根据 HDB3 码的特征，找到相应的+1 码和-1 码，并且判断 V 码和 B 码并清零，实现解码。

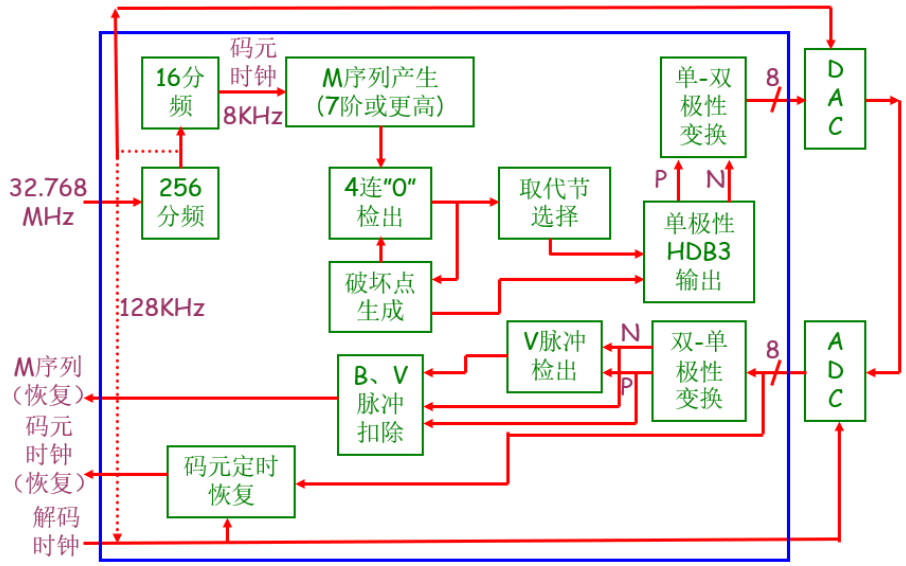

图 1.设计方案

实际情况的通信系统设计还需要考虑到经济等因素，关系到 FPGA 芯片的选型，既需要芯片有足够多的功能和响应速度以满足系统需求，又不宜过多导致性能过剩，导致经济成本过高，以满足系统需求略有性能溢出为最佳。

FPGA 芯片选择 ALTREA EP2C8T144C8N

DA 转换器选择 DAC0832

AD 转换器选择高速 CMOS 八位 AD 转换器 MX7820

四、 设计仪器及设备

1. Quartus II 13.0 (64-bit)
2. Modelsim SE-64 10.4
3. 可编程通信系统教学实验箱 TJ-FPGA-03
4. Tektronix TDS 1012C-EDU 双通道数字示波器

五、 设计项目管理与分工

根据设计方案将整个项目大致分为三部分，钟志鸿负责最开始的 32.768MHz 系统时钟进行 256 分频和 16 分频以及原始的 M 序列产生模块，其中 256 分频后

提供给后续模块进行码元定时恢复使用，16 分频时钟作为码元时钟提供 M 序列生成模块使用；

林程浩负责将产生的 M 序列模块进行 AMI 码转换、连 0 检测、插 V 码和 B 码以及单双极性转换模块，将双极性数据转化为八位数据传送给 DA 转换器输入；

模拟传输部分和 DA 转换器、AD 转换器在实验箱上，仿真时暂时跳过；

凌智城负责 AD 转换器输出的八位数据使用 256 分频的时钟进行边缘检测，根据实际数据特征恢复 8KHz 的码元时钟提供后续使用，同时将双极性数据转换成双极性数据，然后进行极性转换变成单极性，检测 V 码，清除多余的+1 码和-1 码，恢复得到原始的 M 序列，实际过程中会有一定的延时。

三个部分的分工较为独立，不影响其他部分的工作，因此可以比较高效地同步进行独立设计工作。

六、 各模块设计与仿真

1. 分频和 m 序列产生部分一共设计了 3 个模块，分别为 even256_div 模块、even16_div 模块、m_sequence 模块。

1) 模块 even256_div 的作用是将最开始的 32.768MHz 系统时钟进行偶数 256 分频，产生新的时钟信号，供给后续 16 分频模块使用，同时用于解码部分的码元定时恢复模块判决时钟使用；
2) 模块 even16_div 的作用是将 256 分频输出的时钟再次进行偶数分频，得到的时钟就是用于 m 序列产生的码元时钟；
3) 模块 m_sequence 的作用是根据 7 阶 m 序列的本原多项式，循环位移，使用 16 分频的码元时钟产生伪随机序列模拟基带传输信号。

2. HDB3 编码部分一共设计了 4 个模块，分别为 add_v 模块、add_b 模块、polar 模块、change 模块。

1) 模块 add_v 的作用是找到 m 序列中 4 个连续的 0，并将第四个连 0 变成 v。
   a) 输入信号为“0”码和“1”码，输出信号用两位二进制数“00”“01”，“11”分别表示信号为“0”、“1”、“V”；
   b) 设置了一个寄存器用于存储当前码元位置处连“0”个的个数；
   c) 对输入信号进行判断，若输入高电平，则计数器复位，输出为“01”；若输入低电平，则将计数器加一，并判断此时计数器的值是否为“4”，若计数器的值为“4”，则表示出现四个连“0”，将该“0”信号变为“V”且计数器复位，输出为“11”，否则，输出为“00”。
2) 模块 add_b 的作用是为了保证在添加“V”码后的序列中，相邻的“V”码极性必须交替，当“V”码取值不能满足此要求时，则将“000V”用“B00V”替代。
   a) 输入信号为 add_v 模块的输出，输出信号用两位二进制数“00”、“01”、“10”、“11”分别表示信号为“0”、“1”、“B”、“V”；
   b) 设置两个寄存器分别用于存储“1”码和“V”码的个数；
   c) 设置四位移位寄存器方便实现用“B00V”替代“000V”的功能；
   d) 对输入信号进行判断，若输入为“00”，将“1”码计数器加上“V”码计数器的值，并复位“V”码计数器；若输入为“01”，先将“1”码计数器加一，然后加上“V”码计数器的值，最后复位“V”码计数器；若输入为“11”，则“V”码计

数器加一；

e) 用“1”码计数器和“V”码计数器的值判断最终的输出信号，若输入不为“11”，则输出不变；若输入为“11”，判断此时“1”码计数器的值，当此前“1”码的个数为奇数时，输出不变，当此前“1”码的个数为偶数时，输出“10”。

3) 模块 polar 的作用是单极性信号变为双极性信号。

a) 输入信号为 add_b 模块的输出，输出信号用“00”表示“0”码，用“10”表示“+1、+B、+V”码，用“01”表示“-1、-B、-V”；

b) 设置极性判断标志位，当其为“1”时，表示“+1”和“-V”，当其为“0”时，表示“-1”和“+V”；

c) 对输入信号进行判断，若输入为“11”（V 码），利用极性判断标志位判断该码正负；若输入为“01”（1 码）或者“10”（B 码），利用极性判断标志位判断该码正负，同时将极性判断标志位翻转；若输入为“00”（0 码），则输出为“00”。

4) 模块 change 的作用是将双极性信号变更为 8 位二进制数。

a) 输入信号为 polar 模块的输出，输出信号则为 8 位二进制数；

b) 对输入信号进行判断，若输入信号为“00”，则输出信号为“10000000”；若输入信号为“10”，则输出信号为“11111111”，若输入信号为“01”，则输出信号为“00000000”。

3. HDB3 解码和码元时钟定时恢复部分一共设计了 4 个模块，分别为 recover 模块、trans8to1 模块、findv 模块、delvb 模块。

1) 模块 recover 的作用是将八位二进制信号变更为一位的单极性信号。

a) 对输入信号进行判断，八位输出信号有三种情况，8'h00、8'h80、8'hFF，分别代表-1，0 和+1 这三种情况，仿真时可以用三种情况的数字特征来进行边缘检测和时钟定时恢复；

b) 在实际调试过程中，AD 转换输出的三种信号并不是确定的值，会存在毛刺现象，所以需要根据实际情况调整，我们采集到的数据是大于 8'h58 为+1，8'h30~8'h3f 为 0，小于 8'h0f 为-1，故根据三个区间，用 256 分频的时钟去采样，初始置 flag 为 0，如果检测到区间发生跃变，则置 1，同时进行定时恢复，连续八个时钟翻转一次，得到码元定时恢复的时钟提供后续模块使用。

2) 模块 trans8to1 的作用是将八位二进制数据转换为 P 和 N 双极性信号。

a) 输入信号为 AD 转换的 8 位输出，输出信号则为 P 和 N 各一位的二进制数代表双极性；

b) 对输入信号进行判断，若输入信号为“3'h00”，则输出信号为“P=0,N=1”；若输入信号为“3'h80”，则输出信号为“P=0,N=0”，若输入信号为“3'hFF”，则输出信号为“P=1,N=0”；

c) 但实际情况是，真实的 AD 转换输出不会是刚好的 3'h00、3'h80 和 3'hFF，经过实际检测，输出的范围分别是 3'h00~8'h0f、8'h30~8'h3f、大于 8'h58，和理想的情况相差较大，如果实际操作时用理想的三种情况检测，将不能恢复出双极性信号。

3) 模块 findv 的作用是根据 P 和 N 两位数据进行逻辑判定，找到 V 码，输出只包含 0 和 1 的序列。

a) 设置两个 flag 来表示+1 和-1 的个数状态，根据 HDB3 码中的连续同号的 1 来找到 V 码；

b) 如果 PN 是 10，则 flag2 置 0，flag1 自加 1，如果 flag1 自加+后为 2 则表示有两个连续的+1；

c) 如果 PN 是 01，则 flag1 置 0，flag2 自加 1，如果 flag2 自加+后为 2 则表示有两

个连续的-1；

d) 无论是找到连续的+1 还是-1 都全部置 11 表示连续的 1，正常不连续的+1 和-1 用 01 表示，其他状况都清 00 表示 0 基本还原原始序列。

4) 模块 delvb 的作用是将序列进行最后的处理，根据 HDB3 的连 0 规则将多余的 1 清零，恢复出原始的 m 序列完成最终的解码。

a) 设置一个 buffer 来暂存前四位数据，HDB3 的编解码规则是对连续的四个 0 进行处理，之前的模块已经找到了 V 码，这里需要对 V 码以及 B 码清零；

b) 但实际上只要找到了 V 码，将 V 码本身和之前的三位一共四位全部清零，就可以不用考虑 B 码的影响；

c) 检测 indata 的输入若为 11 则表示这是一个 V 码，则使用 buffer 将四位数据清零，输出接到 buffer 的前三位，用 counterv 的 0 和 1 状况来判定清零还是直接 buffer 输出，完成最终的解码。

4. 顶层模块的作用是将所有的功能模块进行连接，并将部分功能模块的输出端口拉出，以方便最终的仿真及 FPGA 引脚的绑定。

1) 分频功能的连接：将系统时钟作为 even256_div 模块的输入，其输出为系统时钟的 256 分频时钟，将其作为 even_div16 模块的输入，其输出为 256 分频时钟的 16 分频时钟；

2) M 序列生成功能的连接：用 16 分频时钟作为 m_sequence 模块的驱动时钟，输入使能信号，输出为 m 序列；

3) HDB3 编码功能的连接：该功能下模块的驱动时钟都为 16 分频时钟，将 m_sequence 模块的输出作为 add_v 模块的输入，将 add_v 模块的输出作为 add_b 模块的输入，将 add_b 模块的输出作为 polar 模块的输入，将 polar 模块的输出作为 change 模块的输入，输出 8 位二进制数；

4) 码元定时恢复功能的连接：用 256 分频时钟作为 recover 模块的驱动时钟，ad 转换器的输出作为 recover 模块的输入，输出为码元定时恢复时钟；

5) HDB3 译码功能的连接:该功能下模块的驱动时钟都为码元定时恢复时钟，ad 转换器的输出作为 trans8to1 模块的输入，将 trans8to1 模块的输出作为 findv 模块的输入，将 findv 模块的输出作为 delvb 模块的输入，delvb 模块的输出即为译码恢复的 m 序列。

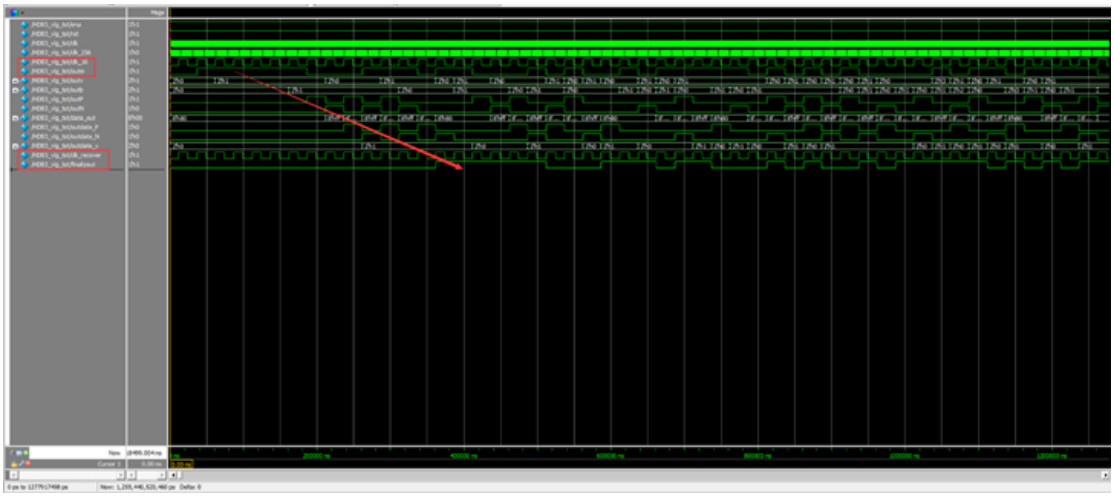

图 2.整体功能仿真

七、 FPGA 验证与调试

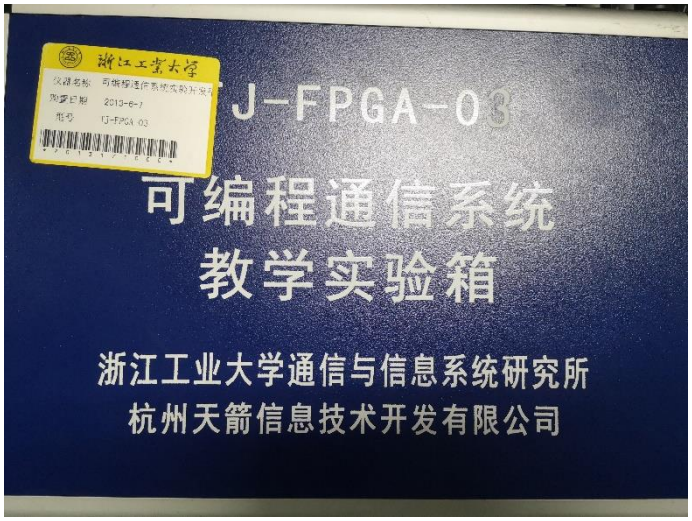

图 3.实验箱

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Differential Pair
clk	Input	PIN_17	1	B1_N0	PIN_17	3.3-V L...efault)		24mA (default)	
clk_16	Output	PIN_71	4	B4_N0	PIN_71	3.3-V L...efault)		24mA (default)	
clk_256	Output	PIN_73	3	B3_N1	PIN_73	3.3-V L...efault)		24mA (default)	
clk_256_ad	Output	PIN_144	2	B2_N1	PIN_144	3.3-V L...efault)		24mA (default)	
clk_256_da	Output				PIN_86	3.3-V L...efault)		24mA (default)	
clk_recover	Output	PIN_70	4	B4_N0	PIN_70	3.3-V L...efault)		24mA (default)	
data_out[7]	Output	PIN_100	3	B3_N0	PIN_100	3.3-V L...efault)		24mA (default)	
data_out[6]	Output	PIN_99	3	B3_N0	PIN_99	3.3-V L...efault)		24mA (default)	
data_out[5]	Output	PIN_97	3	B3_N0	PIN_97	3.3-V L...efault)		24mA (default)	
data_out[4]	Output	PIN_96	3	B3_N0	PIN_96	3.3-V L...efault)		24mA (default)	
data_out[3]	Output	PIN_94	3	B3_N0	PIN_94	3.3-V L...efault)		24mA (default)	
data_out[2]	Output	PIN_93	3	B3_N0	PIN_93	3.3-V L...efault)		24mA (default)	
data_out[1]	Output	PIN_92	3	B3_N0	PIN_92	3.3-V L...efault)		24mA (default)	
data_out[0]	Output	PIN_87	3	B3_N1	PIN_87	3.3-V L...efault)		24mA (default)	
data_out_ad[7]	Input	PIN_7	1	B1_N0	PIN_7	3.3-V L...efault)		24mA (default)	
data_out_ad[6]	Input	PIN_8	1	B1_N0	PIN_8	3.3-V L...efault)		24mA (default)	
data_out_ad[5]	Input	PIN_9	1	B1_N0	PIN_9	3.3-V L...efault)		24mA (default)	
data_out_ad[4]	Input	PIN_24	1	B1_N1	PIN_24	3.3-V L...efault)		24mA (default)	
data_out_ad[3]	Input	PIN_25	1	B1_N1	PIN_25	3.3-V L...efault)		24mA (default)	
data_out_ad[2]	Input	PIN_28	1	B1_N1	PIN_28	3.3-V L...efault)		24mA (default)	
data_out_ad[1]	Input	PIN_30	1	B1_N1	PIN_30	3.3-V L...efault)		24mA (default)	
data_out_ad[0]	Input	PIN_31	1	B1_N1	PIN_31	3.3-V L...efault)		24mA (default)	
data_out_check[7]	Output	PIN_119	2	B2_N0	PIN_119	3.3-V L...efault)		24mA (default)	
data_out_check[6]	Output	PIN_120	2	B2_N0	PIN_120	3.3-V L...efault)		24mA (default)	
data_out_check[5]	Output	PIN_121	2	B2_N0	PIN_121	3.3-V L...efault)		24mA (default)	
data_out_check[4]	Output	PIN_122	2	B2_N0	PIN_122	3.3-V L...efault)		24mA (default)	
data_out_check[3]	Output	PIN_125	2	B2_N0	PIN_125	3.3-V L...efault)		24mA (default)	
data_out_check[2]	Output	PIN_126	2	B2_N0	PIN_126	3.3-V L...efault)		24mA (default)	
data_out_check[1]	Output	PIN_129	2	B2_N1	PIN_129	3.3-V L...efault)		24mA (default)	
data_out_check[0]	Output	PIN_132	2	B2_N1	PIN_132	3.3-V L...efault)		24mA (default)	
ena	Input	PIN_74	3	B3_N1	PIN_74	3.3-V L...efault)		24mA (default)	
finallyout	Output	PIN_103	3	B3_N0	PIN_103	3.3-V L...efault)		24mA (default)	
outN	Output				PIN_75	3.3-V L...efault)		24mA (default)	
outP	Output				PIN_64	3.3-V L...efault)		24mA (default)	
outb[1]	Output				PIN_65	3.3-V L...efault)		24mA (default)	
outb[0]	Output				PIN_79	3.3-V L...efault)		24mA (default)	
outdata_N	Output				PIN_40	3.3-V L...efault)		24mA (default)	
outdata_P	Output				PIN_41	3.3-V L...efault)		24mA (default)	
outdata_v[1]	Output				PIN_43	3.3-V L...efault)		24mA (default)	
outdata_v[0]	Output				PIN_42	3.3-V L...efault)		24mA (default)	
outm	Output	PIN_101	3	B3_N0	PIN_101	3.3-V L...efault)		24mA (default)	
outv[1]	Output				PIN_69	3.3-V L...efault)		24mA (default)	
outv[0]	Output				PIN_67	3.3-V L...efault)		24mA (default)	
rst	Input	PIN_72	4	B4_N0	PIN_72	3.3-V L...efault)		24mA (default)	
<<new node>>									

图 4.引脚分配图

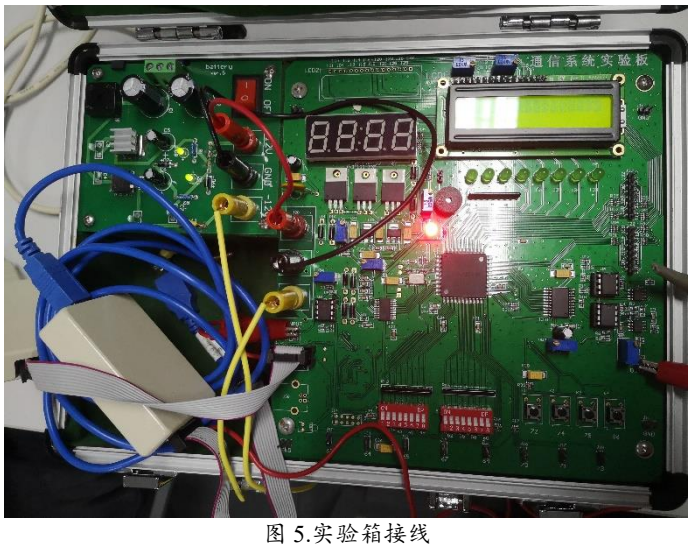

图 5.实验箱接线

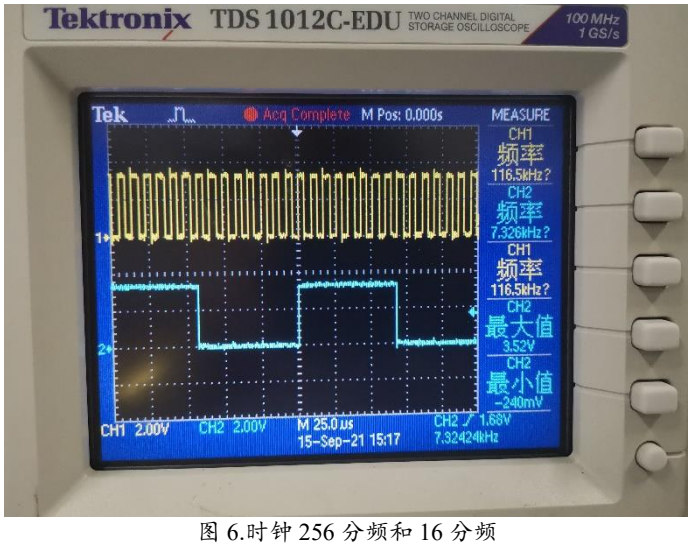

图 6.时钟 256 分频和 16 分频

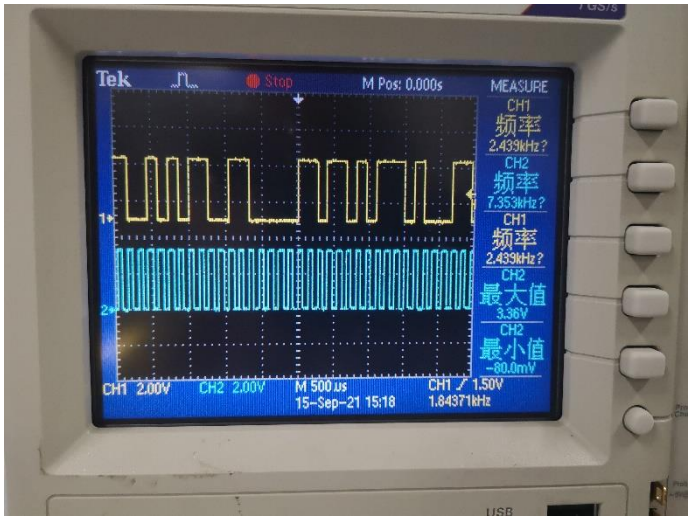

图 7. 时钟 16 分频和 m 序列

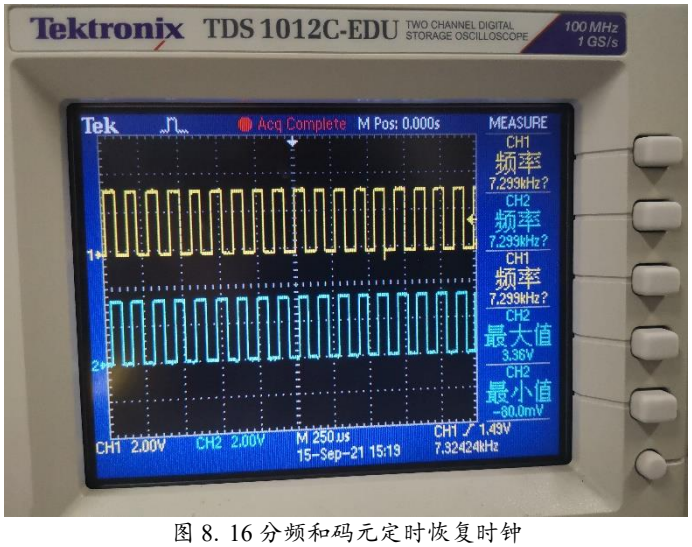

图 8. 16 分频和码元定时恢复时钟

码元定时恢复比较准确并且延时很小

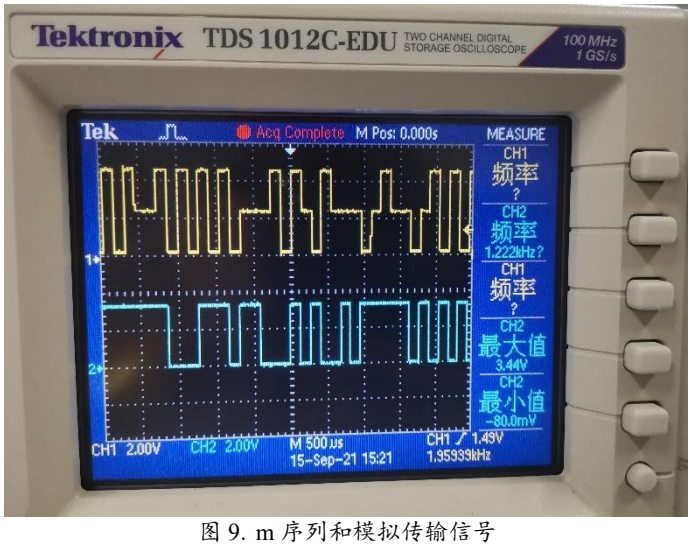

图 9. m 序列和模拟传输信号

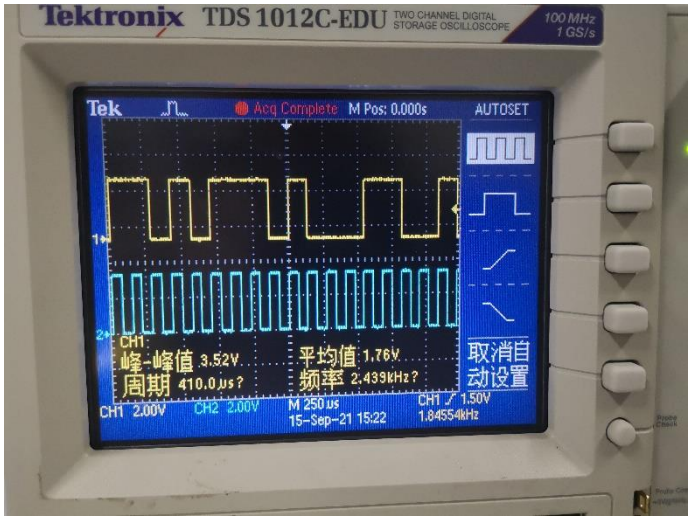

图 10.恢复的 m 序列和码元定时恢复时钟

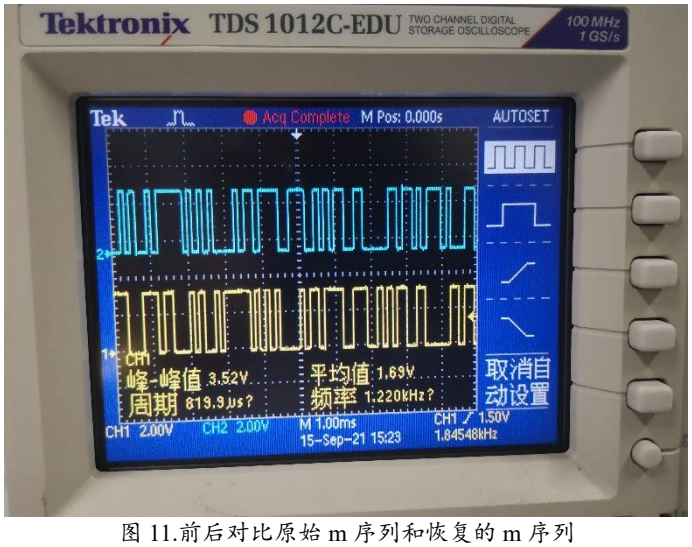

图 11.前后对比原始 m 序列和恢复的 m 序列

前后 m 序列对比成功解码，存在小段时间的时延差由于模拟传输导致，检查长段解码序列，发现与原始的 m 序列都相同。

八、 实验问题总结与心得

问题 1：在 FPGA 仿真时，ad 转换器输入的八位二进制数在一开始无法在示波器上检测到。

解决方法：一开始我们认为是因为 ad 转换器自身的问题，导致输出的 8 位二进制数只在较低位发生变化，但经过多次检测后，我们认为实际 8 位二进制数并没有输入进我们的顶层模块，于是我们通过对顶层模块代码进行检查，发现产生该问题的原因是输入 8 位二进制数的端口本应设置为输入端口吗，但我们设置为了输出端口，所以导致信号并没有传输进来，所以无法检测到。

问题 2：码元定时恢复不理解是什么意思，导致并不能准确的恢复出时钟。

解决办法：请教老师后发现原来是用 256 分频所得的时钟来对八位信号进行采样，根据采集到的信号变化确定码元的起始位置，再根据数字特征定时翻转，起到恢复出 16 分频时钟的作用。

问题 3：实际的 AD 转换器和理想状态不同，用理想的条件去检测无法正常将 8 位 AD 输出转成双极性数据。

解决办法：这个问题困扰的时间最久，由于 AD 转换输出的 8 位数字信号并不是非常完美的，存在跳变以及根据实验箱的实际情况并不会根据 0~5V 输出 00~FF，于是将 AD 输出的八位信号分别绑定到 FPGA 的其他空余引脚，对每个引脚上的数据进行分析，判定-1，0 和+1 分别对应该位的 0 状态还是 1 状态，最终得到我们需要的三种状态区间，高四位可以比较准确的判定，而第四位的 0 和 1 状态反复跳变，最终确定的区间为：

3’h00~8'h0f 代表-1，8'h30~8'h3f 代表 0，大于 8'h58 代表+1

九、 附原代码（或电路图），要求有注释。

even256_div.v

```
module even256_div(clk,rst,clk_out);

input clk,rst;
output clk_out;
reg clk_out;
reg[7:0] count;
// 偶数分频
parameter N=256;

always @(posedge clk or negedge rst)
    if(!rst) begin
        // 复位初始化
        count<=1'b0;
        clk_out<=1'b0;
    end
    else if(N%2==0)begin
        if(count<N/2-1) count <=count+1'b1;
        else begin
            // 计数到一半，时钟翻转实现分频
            count<=1'b0;
            clk_out<=~clk_out;
        end
    end

endmodule
```

even16_div.v

```
module even_div16(clk,rst,clk_out);
```

```
input clk,rst;
output clk_out;
reg clk_out;
reg[3:0] count;
// 偶数分频
parameter N=16;

always @(posedge clk or negedge rst)
    if(!rst) begin
        // 复位初始化
        count<=1'b0;
        clk_out<=1'b0;
    end
    else if(N%2==0)begin
        if(count<N/2-1) count <=count+1'b1;
        else begin
            // 计数到一半，时钟翻转实现分频
            count<=1'b0;
            clk_out<=~clk_out;
        end
    end

endmodule
```

m_sequence.v

```
module m_sequence(clk,rst,ena,data_out,load);

input clk;                          //时钟信号
input rst;                          //复位信号，低电平有效
input ena;                          //控制信号，高电平时序列发生器开始工作
output data_out;
output load;                        //控制信号，为高电平时表示伪随机序列开始
reg data_out,load;
reg [6:0]temp;                      //7 级移位寄存器产生周期为 127 的 m 序列

always @(posedge clk or negedge rst) begin
    if(!rst) begin
            data_out <= 0;
            load <= 1'b0;               //控制信号设为无效
            temp <= 7'b1111_111;        //移位寄存器初始状态设为全 1
    end
    else begin                          //开始产生序列信号
        if(ena) begin                   //判断序列发生器的控制信号是否有效
                //若控制信号有效
                load <= 1'b1;           //将控制伪随机序列产生的信号设为有效
                temp <= {temp[5:0],temp[2]^temp[6]};        //对应 f(x)=x^7+x^3+1
                //当控制伪随机数列产生的信号使能时将移位寄存器的最高位做为 m 序列进行输出
            if(load)    data_out <= temp[6];
        end
        else load <= 1'b0;      //若控制信号无效，则不开始产生伪随机序列
    end
end

endmodule
```

add_v.v

```
module add_v(rst,data_in,data_out,clk);

input data_in,rst,clk;
output [1:0]data_out;
reg [1:0]data_out;                  //用 00、01、11 分别表示输入信号为“0、1、V”
reg [1:0]counter;                   //设置连 0 计数器
```

```
always@(posedge clk or negedge rst) begin
    if(!rst) begin
        counter <= 0;
        data_out <= 0;
    end
    else if(data_in == 1'b1) begin   //判断输入信号是否为 1
        counter <= 0;                    //若为 1 则计数器复位，输出“01”
        data_out <= 2'b01;
    end
    else if(data_in == 1'b0) begin                              //若输入信号为 0，计数器+1
        counter <= counter + 1;
        if(counter == 2'b11) begin      //判断连 0 个数是否达到 4 个，因为非阻塞赋值，此时计数器值
应为 3 时，表示出现 4 个连 0
            data_out <= 2'b11;           //将 0 的输出变为 V
            counter <= 0;                //计数器复位
        end
        else data_out <= 2'b00;          //若连 0 数不为 4，输出“00”
    end
end

endmodule
```

add_b.v

```
module add_b(rst,data_in,data_out,clk);

input clk,rst;
input [1:0]data_in;
output [1:0]data_out;          //用 00、01、10、11 分别表示输入信号为“0、1、B、V”
reg counter1;                  //设置“01”的计数器
reg counterv;                  //设置“11”的计数器
reg [1:0]buffer[3:0];          //取代节选择

always@(posedge clk or negedge rst) begin
//设置四位移位寄存器方便插“B”的实现
    if(!rst) begin
        buffer[3]<=0;
        buffer[2]<=0;
        buffer[1]<=0;
        buffer[0]<=0;
    end
    else begin
        buffer[3]<=buffer[2];
        buffer[2]<=buffer[1];
        buffer[1]<=buffer[0];
        buffer[0]<=data_in;
    end
end

always@(posedge clk or negedge rst) begin
//对输入进行判断
    if(!rst) begin
        counter1 = 0;
        counterv = 0;
    end
else if(data_in == 2'b11) //如果输入为“11”，则 counterv 加 1
        counterv <= counterv + 1'b1;
    else if(data_in == 2'b01) begin //如果输入为“01”，则 counter1 加 1
        counter1 <= counter1 + 1'b1 + counterv;
        counterv <= 1'b0;
    end
    else begin
        counter1 <= counter1 + counterv;
        counterv <= 1'b0;
    end
```

```
end

//若输入为“11”,如果此前 1 的个数为奇数，则输出不变，若为偶数，则输出“10”
//若输入不为“11”,则输出不变
assign data_out = (counter1%2 == 0) && (counterv == 1)? 2'b10 : buffer[3];

endmodule
```

polar.v

```
module polar(rst,data_in,data_outP,data_outN,clk);

input [1:0]data_in;
input rst,clk;
output data_outP,data_outN;                 //用 P 与 N 表示正负
reg [1:0] polar_out;                        //“00”表示 0，“10”表示+1，“01”表示-1
reg data_outN,data_outP;
reg even;                                   //设置极性判断标志,1 表示+1 和-V，0 表示—1 和+V

always@(posedge clk or negedge rst) begin
    if(!rst) even <= 1;
    else if(data_in == 2'b11) begin         //若输入为“11（V）”
        if(even == 1)                       //判断极性标志，若 even 为 1
            polar_out <= 2'b01;             //输出为“01（-1）”
        else                                //若 even 为 0
            polar_out <= 2'b10;             //输出为“10（+1）”
    end
    //若输入为“01（1）”或者“10（B）”
    else if(data_in == 2'b01 || data_in == 2'b10) begin
        if(even == 1) begin                 //判断极性标志，若 even 为 1
            even <= 0;                      //将 even 翻转
            polar_out <= 2'b10;             //输出为“10（+1）”
        end
        else begin                          //若 even 为 0
            even <= 1;                      //将 even 翻转
            polar_out <= 2'b01;             //输出为“01（-1）”
        end
    end
    else if(data_in == 2'b00)               //若输入为“00（0）”
        polar_out <= 2'b00;                 //输出为“00（0）”
end

always@(polar_out or rst) begin
//将输出寄存器的两位数分别赋值给输出端口 data_outP 和 data_outN
    if(!rst) begin
        data_outP <= 0;
        data_outN <= 0;
    end
    else if(polar_out == 2'b01) begin
        data_outP <= 0;                     //PN=01 表示-1
        data_outN <= 1;
    end
    else if(polar_out == 2'b10) begin
        data_outP <= 1;                     //PN=10 表示+1
        data_outN <= 0;
    end
    else begin
        data_outP <= 0;                     //PN=00 表示 0
        data_outN <= 0;
    end
end
endmodule
```

change.v

```
module change(rst,data_inP,data_inN,data_out,clk);

//二输入八输出模块，将 PN 信号转换为八位输出信号作为 DAC 的输入
input rst,clk,data_inP,data_inN;
output [7:0]data_out;
reg [7:0]data_out;

always@(posedge clk or negedge rst)        begin
      if(!rst)
            data  out <= 8'b10000000;
      else if(data_inP == 1 && data_inN == 0) begin              //PN=10 对应输出 HFF
            data_out <= 8'b11111111;
      end
      else if(data_inP == 0 && data_inN ==1) begin              //PN=01 对应输出 H00
            data_out <= 8'b00000000;
      end
      else if(data_inP == 0 && data_inN ==0) begin              //PN=00 对应输出 H80
            data_out <= 8'b10000000;
      end
end

endmodule
```

recover.v

```
module recover(clk_in, rst_n, indata_8, clk_out);

input clk_in, rst_n;
input [7:0]indata_8;
output reg clk_out;
reg [3:0]counter;
reg [7:0]buffer;
reg flag;

always @(posedge clk_in or negedge rst_n) begin
      // 如果复位则 buffer 出清零，否则就将输入的八位信号给 buffer 处理
      if(!rst_n) buffer<=8'b0;
      else buffer<=indata_8;
end


always @(posedge clk_in or negedge rst_n) begin
      if(!rst_n) begin
      // 复位初始化
            clk  out<=0;
            flag <=0;
            counter <=1;
      end
      else begin
            /*if ((buffer[7] == 1'b0&&buffer[6] == 1'b1&&(indata_8 ==1'b1 || indata_8 ==1'b0)) ||
                  (buffer[7] == 1'b1 &&indata_8[7]!=1'b1)||
                  (buffer[7] == 1'b0&&buffer[6]==1'b0&&(indata_8[7]==1'b1||indata_8[6]==1'b1))) flag<= 1;
*/
                  // 根据实际 AD 输出调整，由于 AD 输出不是刚好的 8'h80,8'hff 和 8'h00 三种情况，而是
三个不确定的区间
                  // 用这三个实际的变化区间来进行码元的边缘检测
            if ((buffer >= 8'h58 && indata  8 <=8'h3f) || (buffer <=8'h3f && buffer >= 8'h30 && (indata  8 >=
8'h58 || indata_8 <= 8'h0f)) ||
                        (buffer <= 8'h0f && indata_8 >= 8'h30)) flag<= 1;
            if(flag == 1) begin
                  if (counter==4'h8) begin
                  // 如果计数到 8 则翻转时钟
                        clk_out=~clk_out;
                        counter<=1;
```

```
                end
                else begin
                // 计数没到 8 则继续
                    counter <= counter+1;
                end
            end
        end
end

endmodule
```

trans8to1.v

```
module trans8to1(rst_n, indata_8, outdata_P, outdata_N,clk);

input rst_n,clk;
input [7:0] indata_8;
output outdata_P, outdata_N;
reg outdata_P, outdata_N;

always @(posedge clk or negedge rst_n) begin
        if(!rst  n) begin
            // 复位初始化
            outdata_P <= 0;
            outdata_N <= 0;
        end
        else begin
        // 三个区间分别对应三种双极性 P 和 N 的值来表示+1、0 和-1
        if(indata_8 >= 8'h58) begin
                outdata_P<=1;
                outdata_N<=0;
            end
            else if(indata_8 <= 8'h3f && indata_8 >= 8'h30) begin
                outdata_P<=0;
                outdata_N<=0;
            end
            else if(indata_8 <= 8'h0f) begin
                outdata_P<=0;
                outdata_N<=1;
            end
        end
end

endmodule
```

findv.v

```
module trans8to1(rst_n, indata_8, outdata_P, outdata_N,clk);

input rst_n,clk;
input [7:0] indata_8;
output outdata_P, outdata_N;
reg outdata_P, outdata_N;

always @(posedge clk or negedge rst_n) begin
        if(!rst  n) begin
            // 复位初始化
            outdata_P <= 0;
            outdata_N <= 0;
        end
        else begin
        // 三个区间分别对应三种双极性 P 和 N 的值来表示+1、0 和-1
        if(indata_8 >= 8'h58) begin
                outdata_P<=1;
                outdata_N<=0;
            end
            else if(indata_8 <= 8'h3f && indata_8 >= 8'h30) begin
```

```
                outdata_P<=0;
                outdata_N<=0;
            end
            else if(indata_8 <= 8'h0f) begin
                outdata_P<=0;
                outdata_N<=1;
            end
        end
end

endmodule
```

delvb.v

```
module delvb(rst_n, indata, outdata,clk);

input rst_n,clk;
input [1:0]indata;
output outdata;
reg [3:0]buffer;
reg bufferdata;
reg counterv;

always@(posedge clk or negedge rst_n)
begin// 设置四位 buffer 暂存前四位数据
if (!rst_n) begin
        buffer[3]<=0;
        buffer[2]<=0;
        buffer[1]<=0;
        buffer[0]<=0;
end
else begin
        buffer[3]<=buffer[2];
        buffer[2]<=buffer[1];
        buffer[1]<=buffer[0];
        buffer[0]<=bufferdata;
end
end


always @(posedge clk or negedge rst  n) begin
        // 复位初始化
        if(!rst_n) bufferdata <= 0;
        else begin
            if(indata==2'b01) begin
            // 如果是正常的 01，则数据还是 1
                counterv <= 0;
                bufferdata<=1;
            end
            else if(indata==2'b00) begin
            // 如果是正常的 00，则数据还是 0
                counterv <= 0;
                bufferdata<=0;
            end
            else if(indata==2'b11) begin
            // 如果是的 11，则表示该位为 V 码，将 counterv 标记 1
                counterv <= 1;
                bufferdata<=0;
            end
        end
end

assign outdata = (counterv == 1)    ? 0 : buffer[2];

endmodule
```

HDB3.v

```
module HDB3(rst,
                        clk,clk_256,clk_16,clk_recover,clk_256_ad,clk_256_da,
                        ena,
                        data_out,data_out_ad,outm,outv,outb,outP,outN,data_out_check,
                        finallyout,outdata_P,outdata_N,outdata_v
                        );


input    rst,clk,ena;
input [7:0]data_out_ad;
output clk_256,clk_16,clk_recover,clk_256_ad,clk_256_da;
output [7:0]data_out_check;
output [7:0]data_out;
output [1:0]outv;
output [1:0]outb;
output outm,outP,outN;
output outdata_P,outdata_N;
output [1:0]outdata_v;
output finallyout;
//分频
even256_div div256(clk,rst,clk_256);
even_div16 div16(clk_256,rst,clk_16);
//m 序列
m_sequence m(clk_16,rst,ena,outm,load);
//编码
add_v u1(rst,outm,outv,clk_16);
add_b u2(rst,outv,outb,clk_16);
polar u3(rst,outb,outP,outN,clk_16);
change u4(rst,outP,outN,data_out,clk_16);
//码元定时恢复
recover u8(clk_256, rst, data_out_ad, clk_recover);
//译码
trans8to1 u5(rst, data_out_ad, outdata_P, outdata_N,clk_recover);
findv u6(rst, outdata_P, outdata_N, outdata_v,clk_recover);
delvb u7(rst, outdata_v, finallyout,clk_recover);

assign clk_256_ad = clk_256;
assign clk_256_da = clk_256;
assign data_out_check = data_out_ad;

endmodule
```