通信原理实验

无 45 孙径舟 2014011153
无 45 黄天昊 2014011157
无 46 黄秀峰 2014011193
无 47 刘 畅 2014011196

1 实验目的

- 根据所学习的现代通信原理知识，自己设计实验题目，并根据题目的需要在实验平台上选用所需要的器件，完成设计；
- 提高对现代通信原理中一些基础理论的理解和运用；
- 锻炼自学能力，提高设计能力，培养团队精神。

2 实验平台

实验平台主要包括以下几部分：FPGA（FLEX10K30RC240）、FLASH、锁相环、压控振荡器、单片机、直接数字频率综合器、码型变换、存储器、带低通滤波器的模数转换、带低通滤波器的数模转换、拨码开关、晶体振荡器和二极管显示等部分。我们主要使用了：FPGA、D/A 转换器、拨码开关和晶体振荡器。

3 实验内容

设计一个包含信道编解码、交织、调制解调等模块的通信系统，并前其在 FPGA 平台上实现

3.1 系统功能

系统包含信道编解码、交织、调制解调等模块，可实现比特流的传输

3.2 系统结构

系统按顺序包括如下模块：

- 信道编码
- 交织
- 调制
- 信道模拟
- 解调
- 解交织
- 信道解码

3.2.1 信道编码模块

卷积码将 k 个信息比特编码成 n 个比特，适合以串行形式传。其编码器一般表示为 (n,k,m) 的形式，其中 k 为每次输入到卷积编码器的比特数，n 为每次输入 k 位比特后输出的编码长度，m 为编码存储长度。

本次实验中，我们使用的是 $(2,1,3)$ 卷积编码，每输入一个信息比特，编码产生两位比特的输出，其框图如图.1所示。

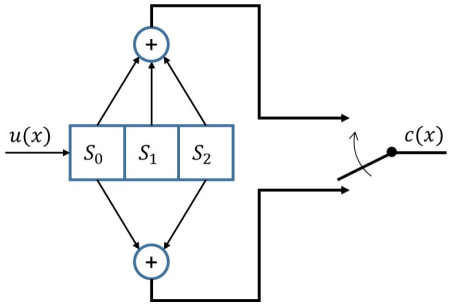


图 1: (2,1,3) 编码器

编码器的输入端口信号为：

- clk　输入控制时钟
- clk2　输出控制时钟

- data_in 编码数据输入
- reset 重置位

其中 clk2 用于对编码后数据进行控制输出，由于我们采用 $(2,1,3)$ 编码器，故而 clk2 的频率是 clk 的两倍。

编码器的输出端口信号为：

- code_out 编码数据输出
- valid 编码有效标志位
- valid_wave 编码有效信息流

由于我们设计了交织和调制解调模块，需要保证信号同步才能使系统正确工作，所以我们设计了 valid 以及 valid_wave 这两个控制信号来进行各模块间的信号同步。

我们还定义了一个寄存器数组 state，用于存储之前时刻的输入。由框图可知，code_out 和 state 的关系可以表示为：

code_out[0] = data_in + state[0] + state[1]

code_out[1] = data_in + state[1]

每次输入一位比特，编码两位输出，并将 valid 置为有效。当 reset 有效时，state 清空，并将 valid 置为无效。

仿真波形如图.2所示。其中 data_send 为原始序列，code_send 为编码后序列；二者对应的时钟分别为 clk_div2 和 clk。从图中可以看出，data_send 每字符占用周期是 code_send 两倍。下面用 Python 检验了一下输出结果，如图.3，可见我们的编码是正确的。

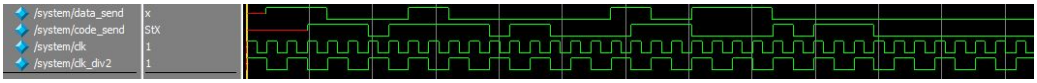


图 2: (2,1,3) 编码器仿真波形

3.2.2 交织模块

交织器的原理及作用 在通信中，传输信息比特差错经常是成串发生的。然而，信道编码仅在检测和校正单个差错和不太长的差错串时才有效。为了解决这一问题，希望能找到把一条消息中的相继比特分散开的方法，即一条消息中的相继比特以非相继方式被发送。这样，在传输过程中即使发生了成串差错，恢复成一条相继比特串的消息时，差错也就变成单个 (或长度很短), 这时再用信道编码纠错功能纠正差错，恢复原消息。简单来说，交织就是将信道传输过程中的连续差错分散化。

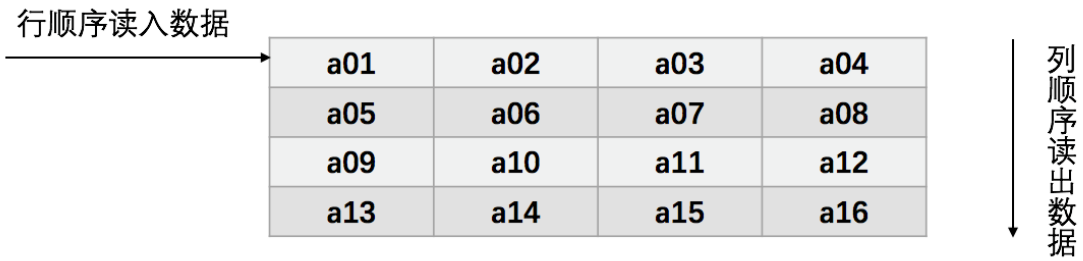
```
In [1]: def conv(a):
            b = list()
            for i in range(2,len(a)):
                b.append((a[i]+a[i-1]+a[i-2])%2)
                b.append((a[i]+a[i-2])%2)
            return b

        a = '0010010000101100'
        a = list(map(int,a))
        b = conv(a)
        print(b)

        [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1]
```

图 3: (2,1,3) 编码器输出

交织器的实现 本实验中采用最常见的交织方法——行列交织。行列交织基本的原理如下图所示：

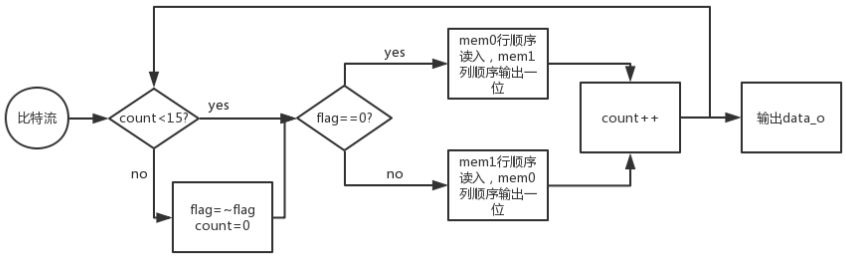


即行顺序写入寄存器，列顺序读出。上图交织结果为 a01,a05,a09,a13,a02,a06,a10,a14, a03,a07,a11,a15,a04,a08,a12,a16.

本实验中实现行列交织的具体算法为：

使用两个寄存器数组，一个的作用为行顺序接收当前输入数据并存储时另一个负责按照列顺序输出数组中存储数据。每周期读入一个比特，每 16 个周期两数组作用互换。

每一周期读入比特的流程图如下：



初始时：count=0；flag=0;mem0[15:0],mem1[15:0]=0;data_o=0;

3.2.3 调制、信道模拟和解调模块

本调制解调模块只包括数字部分。整个通信系统构建在 FPGA 平台上，模拟调制模块无法使用 Verilog 描述的数字电路实现。本小节将详细解释采用的数字调制解调方式和实现方法。为了测试调制解调模块的抗干扰能力，在解调模块的输入部分人为加入一定幅度的伪随机信号作为噪声干扰，该功能由信道模拟模块完成。另外，为模拟真实工作场景，解调模块具有简单的载波同步功能。

(1) QPSK 调制和解调

数字调制模块使用 QPSK 调制。QPSK 的星座图见图 X。QPSK 的输入符号由 2bit 数据构成，对应 4 种不同的输出波形。将输入符号记为 b_0b_1，输入符号和星座图坐标 (a_I, a_Q) 的对应关系如表格 1。脉冲成形采用简单的矩形窗，于是 QPSK 发送信号波形为 $Y_{b_0b_1}(t) = A'a_I sin(\omega \cdot t + \phi) + A'a_Q cos(\omega \cdot t + \phi)$ $(N \cdot T < t \leq (N+1) \cdot T)$ 。经过化简，$Y_{b_0b_1}(t) = sin(\omega \cdot t + \phi + \theta_{b_0b_1})$，$A$ 是一个常数，$\theta_{b_0b_1}$ 的取值见表格 1。因此，QPSK 调制可以视作为将输入符号映射到正弦波相位，然后输出一定频率正弦波的过程。为了方便起见，实际操作中取 $\phi = -\pi/4$，4 个符号依次对应 $sin(\omega t), cos(\omega t), -sin(\omega t), -cos(\omega t)$ 输出波形。

输入符号	星座图坐标	$\theta_{b_0b_1}$	$\phi + \theta_{b_0b_1}$
0 0	(+1, +1)	$\pi/4$	0
0 1	(-1, +1)	$3\pi/4$	$\pi/2$
1 1	(-1, -1)	$5\pi/4$	π
1 0	(+1, -1)	$7\pi/4$	$3\pi/2$

表 1: 输入符号，星座图坐标和相位的对应关系

为了将调制后的波形进行解调，解调模块需要把接收信号和相移相差 $\pi/2$ 的两路信号 $sin(\omega t + \phi), cos(\omega t + \phi)$ 做相关操作，根据相关结果，做门限判定，可还原 b_0b_1。本系统对 QPSK 解调方式做一定修改，使用 $sin(\omega t), cos(\omega t)$ 做相关。假设相关结果分别是 R_{sin}, R_{cos}，判决门限有两个：HIGH 和 LOW，则符号 b_0b_1 的判定规则见下表。表格 2 中的 x 表示 don't care。

(2) 信道加噪和载波同步

为了模拟信道真实情况，在 QPSK 调制输出端，我们人为地加入一定幅度的噪声。噪声生成源是一个伪随机数发生器，无法保证生成白噪声，但这足以演示解调系统的抗噪性能。

(R_{sin}, R_{cos})	b_0b_1
(>HIGH, x)	**00**
(x, >HIGH)	**01**
(<LOW, x)	**11**
(x, <LOW)	**10**

表 2: QPSK 解调判定规则

另外，实际通信系统的解调模块需要探测输入信号的开始时刻，以确保用来做相关判别的正弦信号相位准确。每次开始进行数据传输时，通信系统会自动加入 Nbit 接收方已知的数据作为帧头，用来同步载波相位。解调模块会不断对输入信号做相关操作。当一段时间内相关输出满足一定要求时，解调模块视作帧头同步完成。因为帧头数据已知，对应的信号相位也已知，解调模块随后可以生成相位正确的 $sin(\omega t), cos(\omega t)$ 信号进行数据解调工作。

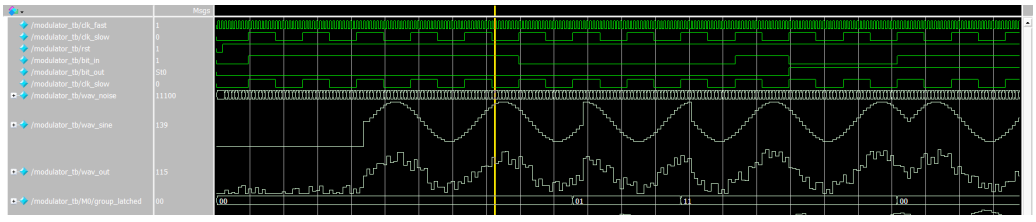
(3) 系统实现和仿真结果

本系统的调制解调模块用到两个时钟，其中快时钟频率是慢时钟的 16 倍。FPGA 无法生成模拟正弦信号，本系统采用查找表 + 快时钟的方法生成每 2 个慢周期 32 个采样点的数字正弦信号。为了避免之后计算相关出现负数需要额外处理的麻烦，此正弦信号被加上一定幅度直流信号，输出均大于 0。

对于**调制模块**（详细代码见 `modulation/modulator.v`），输入输出为

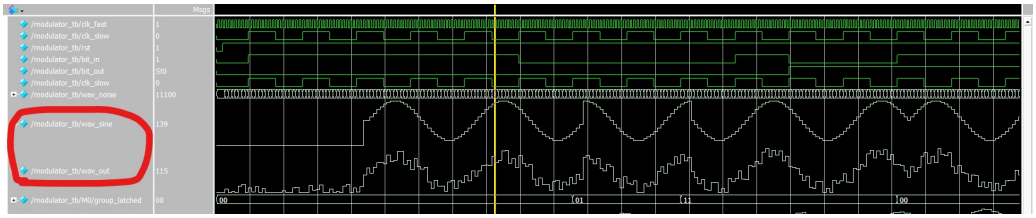
- `input clk_fast, clk_slow, rst, valid; input bit_in;`
- `output reg [7:0] wav_out;`

之前提到，调制模块开始传输数据时，需要额外附加帧头数据，方便解调模块进行相位同步。本系统选用长度 N=4，数据固定为 0000 的帧头。于是对于任何输入比特流，解调模块首先输出的调制波形都会是相移为 0 正弦信号的 2 个完整周期。输入端每两个周期会锁存 2bit 数据 b_0b_1，然后根据表 1，得到对应相位（$\phi + \theta_{b_0b_1}$ ）偏移值，换算成正弦波形起始采样点偏移数，输入到正弦波生成模块中。正弦波生成模块的输出就是调制信号输出。调制模块的仿真结果如下：



使用 5bit 的移位寄存器完成附加帧头的功能。可以观察到，初始波形输出的确是相移为 0 正弦信号的 2 个完整周期。随后 b_0b_1 数据存到 `group_latched[1:0]` 中，生成不同相位的正弦波形。附加起始帧头导致 `group_latched[1:0]` 中的数据是 5–6 个慢时钟周期之前的输入比特流 `bit_in`。

对于**信道加噪模块**（详细代码见 `modulation/pseudo_random.v`，我们从网上找到了一个伪随机数发生器实现，并加入到系统实现中。伪随机数发生器模块输出位宽是 5bit，使用原码，最高位是符号位。输出乘以 2 后和调制输出数据 `wav_out[7:0]` 相加。如果结果小于 0，则相加结果置为 0；如果结果大于 8bit 表示最大值，结果直接置为 255。于是噪声信号的最大值是有用信号最大值的 1/8。正弦波信号和加入噪声后信号的仿真波形如下：



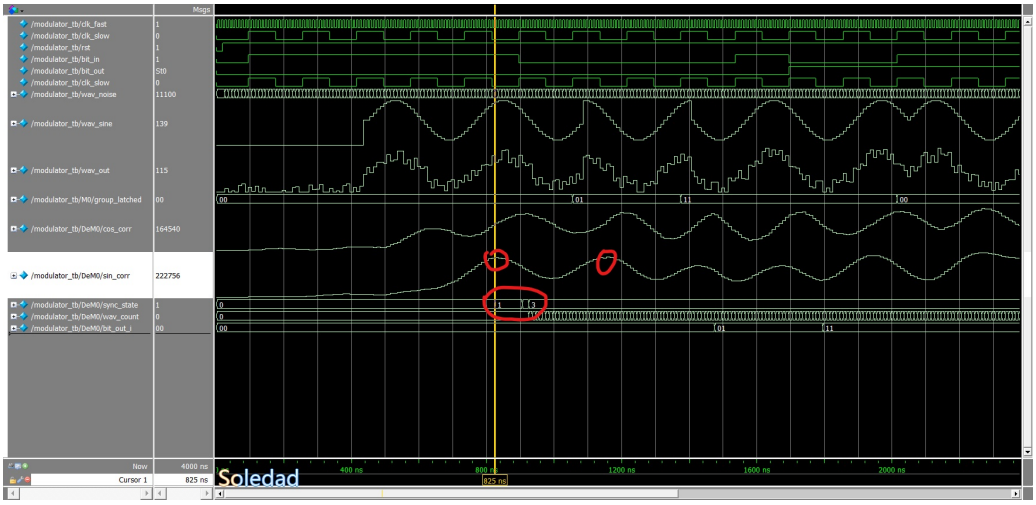
对于**解调模块**（详细代码见 `modulation/demodulator.v`），输入输出为

- `input clk_fast, clk_slow, rst; input [7:0] wav_in;`
- `output reg bit_out; output valid;`

功能可以分成两部分：帧头同步和符号判决。两个部分核心都是相关运算。相关运算单元由 32 个乘法器和 5 级加法树组成，有 6 个快时钟周期的延迟。解调模块会使用移位寄存器保存 32 个周期的输入数据，用来和零相位正弦波/余弦波做相关。因此，当输入信号也是一个零相移的正弦波信号时，正弦波相关结果是一个峰值。每个快时钟周期，相关运算结果都会更新。

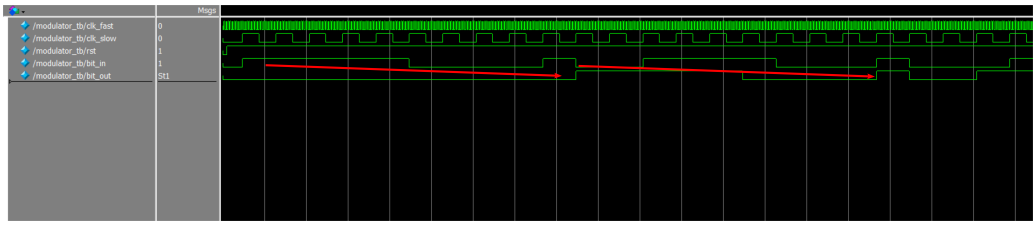
输入只是噪声时，这些相关结果近似为 0。当有用信号开始输入时，因为帧头的存在，最开始的有效波形一定是 2 个慢时钟周期的零相位偏移正弦波 + 噪声，显示在相关的输出上就是两个与正弦波做相关的峰。帧头同步的功能就是捕捉这些峰值的时刻。为了排除其他值较小的峰值影响，定义一个 `HIGH_TH`，只有相关结果大于这个门限值时，我们才捕捉峰值的存在。为简单起见，我们只捕捉输入第一个帧头对应正弦波的相关峰。一旦捕捉到一个大于 `HIGH_TH` 的相关峰，相位匹配就完成了，因为我们知道了第一个符号对应波形的起始时刻，就可以推测之后任意一个符号波形的起始时刻。峰值的捕捉由一个简单的状态机完成。

完成同步后，系统跳过第二个相关峰（仍由帧头数据产生），进行真实数据符号的判决。每两个慢时钟周期，系统会读取正弦波相关数据和余弦波相关数据。可以把它们分别记作 R_{sin}, R_{cos}，根据表格 2 定义的规则进行符号判决。实际系统中 HIGH, LOW 门限的值分别为 200000，120000。仿真结果如下：



帧头带来的两个相关峰被圈注出来了。下方用红笔圈注表示用于捕捉峰值状态机状态的变化，第一个峰值被捕捉后，状态就不再改变。最下方的 `bit_out_i[1:0]`是判决的符号结果，可以和 `group_latched[1:0]`做对比。除了有约 2 个慢时钟周期的延迟外，数据没有出错，而且解调输出中没有帧头数据。

下图是略去中间实现细节，整个调制解调系统的输入输出仿真结果：



红色箭头标示了输入输出信号的延迟。

3.2.4 解交织模块

解交织模块与交织模块相同

3.2.5 信道解码模块

信道解码使用维特比译码器，判决方式为硬判决。设计思路如下：

变量设计

- **codebuff**：接收的码流，存于缓冲区中，数量达到一组后送入译码区
- **code**：译码区码流，即当然需要译码的码流
- **possible_codex**：维特比译码图中，当前阶段第 x 个状态（共有四个状态）所对应的最优路径
- **errorx**：第 x 个最优路径（共有 4 条路径）对应的判决误差
- **decodex**：维特比译码中，当前阶段第 x 状态的最优路径所对应的输出译码
- **decode**：前一组码流（已译码结束）所对应的输出译码

译码流程

1. 接收码流至 codebuff，当接收完一组后，送入 code 中等待译码
2. 每一个时钟周期，译码往前推进两位，求出当前状态下的最优路径并计算硬判决误差，并记录四个状态所对应的最优路径
3. 完成一组译码后，将译码结果存入 decode，之后每一个时钟周期输出一位译码结果

3.3 软件仿真

软件仿真具体情况如下：

- 仿真平台：ModelSim-Altera 10.4b
- 输入序列：[1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0]
- 编码方式：(2,1,3) 卷积码
- 调制方式：QPSK
- 是否交织：是
- 是否加噪声：是

编码输入及输出波形已在原理部分有所展示，在此不予赘述。我们将考虑的内容包括：(1) 通过噪声信道前后发送序列；(2) 编码后序列与解码输入；(3) 解码后序列与原始序列。

通过噪声信道前后发送序列

通过噪声信道前序列即交织后序列，记作 bit_send。而通过噪声信道后的序列记作 bit_recv。如.4 中所示，由于交织需要一定的缓冲，从而两个信号之间存在延迟。仔细观察可以看出，这两个序列存在一些差异，这是由于我们添加的噪声较大，正好用来检验卷积码的纠错能力。

图 4: 通过噪声信道前后序列

编码后序列与解码输入

编码后序列及 code_send，而解码输入记作 code_recv，分别如图.5，图.6所示。同样的，由于信道噪声较强，二者并不一样。

图 5: 编码后序列

图 6: 解码输入序列

解码后序列与原始序列

解码后序列记为 data_recv, 如图.7所示，和输入序列完全相同，体现了卷积码的纠错能力。我们还可以观察一下 Viterbi 算法给出的最大似然输入，记作 code_prob，如如图.8所示，和 code_send 完全相同。

单独看起来，每个模块都不算复杂，无非是对输入数据做一些处理。但当我们试图把这些模块组合成一个系统是，同步成了最为关键的问题。举例来说，对于交织模块，如果我们交织的部分和实际输入存在 1bit 偏差，这样相当于在编码后结果前加一位 0，并减去末尾一比特。这对于卷积码来说是灾难性的错误，我们将不可能恢复出正确的序列。为了解决同步的问题，我们为每个模块都设置了同步信号。只有当下一级模块收到上一级模块的有效同步信号时，才开始进行相关的操作。牺牲一部分延时性能来换取同步的准确性。

图 7: 解码后序列

图 8: 最大似然输入

4 实验总结

在本次实验中，我们设计了基于 (2,1,3) 卷积码和 QPKS 调制的通信系统，使用 LED 灯作为输出，在有噪声的情况下检验了该系统的性能。实验证明，卷积码在一定噪声情况下依然可以有较好的性能。

由于我们对于卷积码有着较为充分的了解，且团队成员中有人非常擅长编写 Verilog 程序，因此总体上实验进行的较为顺利。虽然如此，各模块之间的同步依然是一个比较困扰我们的问题。在最开始设计的时候我们没有充分考虑到模块同步性的问题，仿真中出现了颇令人费解的现象，在逐级排查解决问题之后，我们终于使得系统能够正常工作。不过此时在边界情况下仍然存在一些小问题，我们尝试了不同的输入情况，尽可能充分地解决了这些问题。

最后，感谢老师和助教们这一学期的悉心指导和辛苦付出，祝二零一八年一切顺利！

分工情况：

- 黄天昊：信道调制解调/噪声模块编写，以及相应部分报告撰写。
- 孙径舟：卷积编码，模块联调与测试，以及相应部分报告撰写。**软件仿真**与**总结**撰写。
- 黄秀峰：卷积解码，模块联调与测试，以及相应部分报告撰写。**实验情况介绍**撰写。
- 刘畅：交织/解交织模块编写，以及相应部分报告撰写。