通信原理实验报告

陈子熠　游笑权　向明义

2025 年 1 月 4 日

目录

1 实验设计

本实验系统由四个主要模块组成：Hamming 编解码、交织与解交织、QPSK 调制解调以及加性高斯白噪声（AWGN）信道。这些模块协同工作，模拟实际通信系统中的信号处理与传输过程。

1.1 Hamming 编解码

在通信系统中，通过在信源中添加冗余信息，可以使信道传输后的结果具有一定的检错与纠错能力。本系统中采用 (7,4) Hamming 信道编码，即将 4 位信息 bit 转换为 7 位编码 bit，经过信道传输后纠错并转换回 4 位信息 bit。具体原理表述如下：

设 $\boldsymbol{m}$ 为 k 位信息 bit 矢量，定义生成矩阵 $G_{k\times n} = [I_{k\times k}; P_{k\times(n-k)}]$ 为单位矩阵 I 与奇偶校验阵 P 的组合，编码后的结果为 $\boldsymbol{x} = \boldsymbol{m}G = [\boldsymbol{m}; \boldsymbol{m}P]$。记 $H^T = \begin{bmatrix} P \\ I \end{bmatrix}$ 为校验矩阵，由于 $GH^T = \begin{bmatrix} I; P \end{bmatrix} \begin{bmatrix} P \\ I \end{bmatrix} = P + P = 0$，故合法码字一定满足 $\boldsymbol{x}H^T = 0$。由于 Hamming 码的最小码距为 3，故最多纠一位错，设通过信道后的 bit 串为 $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{e}$，其中 $\boldsymbol{e}$ 为长度为 n 且最多只有一位为 1 的矢量，则校正子 $\boldsymbol{s} = \boldsymbol{y}H^T = \boldsymbol{x}H^T + \boldsymbol{e}H^T = \boldsymbol{e}H^T$ 为 H^T 矩阵中的某一行，将校正子与 H^T 比较即可得出错误 bit 位置，从而完成纠错。

1.2 交织与解交织

在通信系统中，由于无线信道的深衰落等原因，有可能使得系统中存在不可抗拒的连续突发错误，这使得一组码字中可能出现多于 1 位的错误，则 Hamming 码的纠错能力大大下降。为此，可在编码后引入一个交织器，典型的交织器让信息序列按行写入，按列读出，使得原本相邻的码元在时间上的距离最大化。如图 1 所示，交织矩阵的列数一般不小于分组码长，交织矩阵的行数不小于突发错最大长度。一个交织块的长度为交织矩阵的列数乘行数，略小于交织延时和解交织延时。交织块总长度应为整数个编码块长度，以避免同一个编码块分到前后两个交织块引起过大延时。

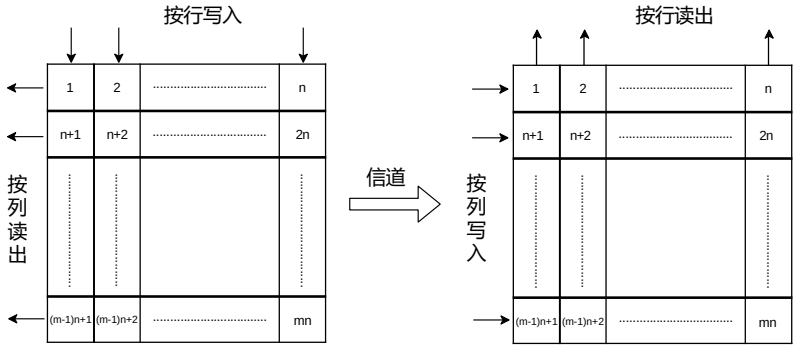


图 1: 交织与解交织过程

在本次实验中，我们取交织矩阵的列数等于 Hamming 码长 $n = 7$，交织矩阵的行数取 $m = 4$，交织块的长度 $mn = 28$.

1.3 QPSK 调制解调

在现代数字通信系统中，QPSK（Quadrature Phase Shift Keying，正交相位调制）是一种高效的调制方式，它通过在同一时间内传输两个比特来实现高数据传输率。如图 2 所示，QPSK 调制器将输入的二进制数据分成两部分，每部分表示一个信号的相位，分别控制正交的两个载波（I 路和 Q 路）。这种方式可以有效利用带宽，同时保持较高的抗干扰能力。

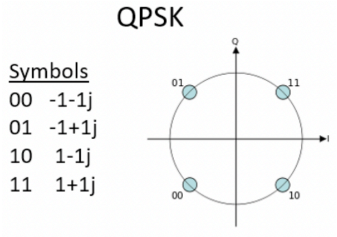


图 2: QPSK 调制解调模型

QPSK 调制与解调过程包括以下几个步骤：

- 调制过程：输入的数据每两位作为一个信号符号，映射到四个不同的相位上，通常是 0°，90°，180° 和 270°，分别对应不同的比特组合。
- 解调过程：接收端根据信号的相位变化来恢复发送的比特流。通过比较接收到的信号与阈值，判断其属于哪个相位，并输出对应的比特组合。

1.4 AWGN 信道模型

AWGN（Additive White Gaussian Noise，加性白噪声）信道模型广泛应用于无线通信领域，用于模拟理想情况下的噪声影响。在此信道模型中，噪声是加性、均匀分布的，且具有高斯分布特性。在本实验中，AWGN 信道的模拟过程分为两个步骤：首先，通过 LFSR（线性反馈移位寄存器）生成伪随机数序列；然后，利用 Box-Muller 变换将这些伪随机数转换为服从高斯分布的随机数，最终将这些噪声加到信号中，模拟实际通信中的噪声干扰。

1.4.1 LFSR

为了产生 Gaussian 白噪声，首先需要生成均匀分布的伪随机数序列。本实验中，我们采用线性反馈移位寄存器（LFSR），通过移位寄存器和特定的反馈函数来生成这些随机数。

图 3 展示了一个 16 位 Fibonacci LFSR。其采用的特征多项式为 $X^{16} + X^{14} + X^{13} + X^{11} + 1$。该多项式确保了生成的随机序列的周期最大（65535，不包括全零状态），以有效地模拟随机性。移位寄存器逐位输出随机序列的每一位，同时通过反馈机制决定下一个状态。这一过程不断重复，直到达到最大周期。例如，图中的状态为 0xACE1（十六进制），下一个状态是 0x5670.

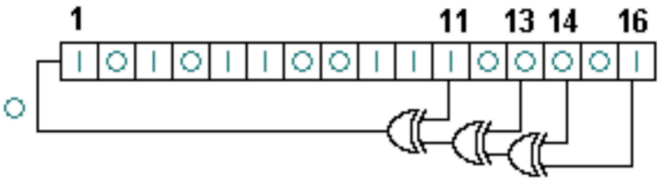


图 3: 一个 16-位 Fibonacci LFSR

1.4.2 Box-Muller

Box-Muller 变换是一种用于从均匀分布的随机数生成高斯分布随机数的方法。在本实验中，我们首先生成两个均匀分布的随机数 $U_1, U_2 \sim U(0,1)$，然后通过以下公式将其转换为符合标准正态分布的随机数：

$$X = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$
$$Y = \sqrt{-2\ln U_1}\sin(2\pi U_2)$$

其中，X 和 Y 分别为符合标准正态分布的随机变量。

2 实验过程

2.1 Hamming 编解码

编码器与解码器均采用组合逻辑编写，其中

$$\text{生成矩阵 } \boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}, \text{校验矩阵 } \boldsymbol{H}^T = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

于是编码器的 Verilog 关键代码为：

```
assign data_o[0] = data_i[0];
assign data_o[1] = data_i[1];
assign data_o[2] = data_i[2];
assign data_o[3] = data_i[3];
assign data_o[4] = data_i[0] ^ data_i[1] ^ data_i[2];
assign data_o[5] = data_i[0] ^ data_i[2] ^ data_i[3];
assign data_o[6] = data_i[0] ^ data_i[1] ^ data_i[3];
```

解码器的 Verilog 关键代码为：

```
// 先计算校正子
wire [2:0] parity;
assign parity[0] = data_i[4] ^ data_i[0] ^ data_i[1] ^ data_i[2];
assign parity[1] = data_i[5] ^ data_i[0] ^ data_i[2] ^ data_i[3];
assign parity[2] = data_i[6] ^ data_i[0] ^ data_i[1] ^ data_i[3];
// 比较校正子和校验矩阵
assign data_o = (parity==3'b111) ? data_i[3:0]^4'b0001 :
                (parity==3'b101) ? data_i[3:0]^4'b0010 :
                (parity==3'b011) ? data_i[3:0]^4'b0100 :
                (parity==3'b110) ? data_i[3:0]^4'b1000 :
                data_i[3:0];
```

2.2 交织与解交织

交织器：当检测到 Hamming 码编码器成功完成 $m = 4$ 次编码时，将 $mn = 28$ 位编码器输出直接映射为 28 位交织器输出。

Verilog 关键代码如下，通过采用直接映射的方式，有效提高了系统的效率。

```
always @(posedge clk or negedge rst) begin
    if (~rst) begin
        data_o <= 0;
```

```
        eno <= 0;
    end
    else if (en) begin
        data_o[0] <= data_i[0];
        data_o[1] <= data_i[7];
        data_o[2] <= data_i[14];
        data_o[3] <= data_i[21];
        data_o[4] <= data_i[1];
        data_o[5] <= data_i[8];
        data_o[6] <= data_i[15];
        data_o[7] <= data_i[22];
        data_o[8] <= data_i[2];
        data_o[9] <= data_i[9];
        data_o[10] <= data_i[16];
        data_o[11] <= data_i[23];
        data_o[12] <= data_i[3];
        data_o[13] <= data_i[10];
        data_o[14] <= data_i[17];
        data_o[15] <= data_i[24];
        data_o[16] <= data_i[4];
        data_o[17] <= data_i[11];
        data_o[18] <= data_i[18];
        data_o[19] <= data_i[25];
        data_o[20] <= data_i[5];
        data_o[21] <= data_i[12];
        data_o[22] <= data_i[19];
        data_o[23] <= data_i[26];
        data_o[24] <= data_i[6];
        data_o[25] <= data_i[13];
        data_o[26] <= data_i[20];
        data_o[27] <= data_i[27];
        eno <= 1;
    end
end
```

解交织器：当检测到 QPSK 解调器成功完成 14 次解调时，将 28 位解调器输出直接映射为 28 位解交织器输出。

Verilog 关键代码如下，与交织过程类似。

```
always @(posedge clk or negedge rst) begin
    if (~rst) begin
        r_data_o <= 0;
        r_eno <= 0;
    end
    else if (r_en) begin
        r_data_o[0] <= r_data_i[0];
        r_data_o[1] <= r_data_i[4];
```

```
        r_data_o[2] <= r_data_i[8];
        r_data_o[3] <= r_data_i[12];
        r_data_o[4] <= r_data_i[16];
        r_data_o[5] <= r_data_i[20];
        r_data_o[6] <= r_data_i[24];
        r_data_o[7] <= r_data_i[1];
        r_data_o[8] <= r_data_i[5];
        r_data_o[9] <= r_data_i[9];
        r_data_o[10] <= r_data_i[13];
        r_data_o[11] <= r_data_i[17];
        r_data_o[12] <= r_data_i[21];
        r_data_o[13] <= r_data_i[25];
        r_data_o[14] <= r_data_i[2];
        r_data_o[15] <= r_data_i[6];
        r_data_o[16] <= r_data_i[10];
        r_data_o[17] <= r_data_i[14];
        r_data_o[18] <= r_data_i[18];
        r_data_o[19] <= r_data_i[22];
        r_data_o[20] <= r_data_i[26];
        r_data_o[21] <= r_data_i[3];
        r_data_o[22] <= r_data_i[7];
        r_data_o[23] <= r_data_i[11];
        r_data_o[24] <= r_data_i[15];
        r_data_o[25] <= r_data_i[19];
        r_data_o[26] <= r_data_i[23];
        r_data_o[27] <= r_data_i[27];
        r_eno <= 1;
    end
end
```

2.3 QPSK 调制解调

QPSK 调制器：

- QPSK 调制器根据输入的 2 比特数据，将其分别映射为 I 路和 Q 路的符号。
- 由于 QPSK 调制是基于正交载波的，调制器的输出信号对应于一个复数的实部和虚部，分别通过 I 路和 Q 路输出。
- 具体实现中，我们通过将每个输入符号映射到 $\pm 1/\sqrt{2}$ 乘以常数值来实现 QPSK 调制。常数值为 $1/\sqrt{2}$，其作用是对输出信号进行归一化处理，避免过强的信号造成干扰。

Verilog 关键代码如下：

```
localparam signed [15:0] CONST_VAL = 16'b0101_1011;  // 1/sqrt(2)
always @(posedge clk or negedge rst) begin
    if (~rst) begin
        data_o_i <= 16'b0;  data_o_q <= 16'b0;
    end else begin
```

```
        case (data_i)
            2'b00: begin
                data_o_i <= CONST_VAL;  data_o_q <= CONST_VAL;
            end
            2'b01: begin
                data_o_i <= -CONST_VAL; data_o_q <= CONST_VAL;
            end
            2'b10: begin
                data_o_i <= CONST_VAL;  data_o_q <= -CONST_VAL;
            end
            2'b11: begin
                data_o_i <= -CONST_VAL; data_o_q <= -CONST_VAL;
            end
            default: begin
                data_o_i <= 16'b0;      data_o_q <= 16'b0;
            end
        endcase
    end
end
```

QPSK 解调器：

- 解调器的作用是根据接收到的 I 路和 Q 路信号，判断信号属于哪一个象限，进而恢复原始数据。
- 解调器通过比较 I 路和 Q 路信号与阈值的大小关系，来判断比特的值。阈值设置为零，即信号大于零则为 1，小于零则为 0。

Verilog 关键代码如下：

```
localparam signed [15:0] THRESHOLD = 16'b0;
always @(posedge clk or negedge rst) begin
    if (~rst) begin
        data_o <= 2'b00;
    end else begin
        if (data_i_i > THRESHOLD) begin
            if (data_i_q > THRESHOLD)   data_o <= 2'b00;
            else                        data_o <= 2'b10;
        end else begin
            if (data_i_q > THRESHOLD)   data_o <= 2'b01;
            else                        data_o <= 2'b11;
        end
    end
end
```

2.4 AWGN 信道模型

2.4.1 LFSR 生成均匀分布随机数

在本实验中，为保证生成的两组随机数序列的独立性，LFSR 的伪随机数序列通过不同的特征多项式、不同的种子生成。具体地，选用不同的 SEED 值（即种子），并根据其最低有效位（LSB）选择特定的反馈多项式：

- SEED[0] = 1 时，特征多项式为 $X^{16} + X^{14} + X^{13} + X^{11} + 1$
- SEED[0] = 0 时，特征多项式为 $X^{16} + X^{15} + X^{13} + X^4 + 1$

Verilog 代码实现如下：

```
reg [15:0] lfsr_state;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        lfsr_state <= SEED;
    end else begin
        if (SEED[0] == 1) begin
            lfsr_state <= {lfsr_state[14:0], lfsr_state[15] ^
            ↪  lfsr_state[13] ^ lfsr_state[12] ^ lfsr_state[10]};
        end else begin
            lfsr_state <= {lfsr_state[14:0], lfsr_state[15] ^
            ↪  lfsr_state[14] ^ lfsr_state[12] ^ lfsr_state[3]};
        end
    end
end
```

在此过程中，生成的伪随机数序列被用于模拟 AWGN 信道中的噪声影响。为了确保生成的随机数适应特定范围的需求，采用如下代码进行范围调整：

```
assign uniform_o = lfsr_state % (O_MAX - O_MIN + 1'b1) + O_MIN;
```

该过程通过调整输出值的范围，确保生成的随机数序列符合信道噪声的要求。

2.4.2 Box-Muller 生成高斯分布随机数

为了提高系统的效率，我们采用查表法来计算 Box-Muller 变换中的 $\ln U_1$ 以及 $\cos(2\pi U_2)$，以减少计算复杂度。具体来说，我们通过查表将 U_1 和 U_2 映射到预先计算好的值，从而实现快速变换。

特别地，对于 $\cos(2\pi U_2)$，我们利用对称性和奇偶性，仅对 $U_1 \in [0, 0.25]$ 的部分进行查表映射，从而降低了存储需求并提高了查表效率：

```
wire cos_sign;
assign cos_sign = (uniform_o_0 <= 256) ? 0 :
                  (uniform_o_0 <= 768) ? 1 :
                  0;
```

```
wire [15:0] cos_addr;
assign cos_addr = (uniform_o_0 <= 256) ? (uniform_o_0 - 1) :
                  (uniform_o_0 <= 512) ? (512 - uniform_o_0) :
                  (uniform_o_0 <= 768) ? (uniform_o_0 - 513) :
                  (1024 - uniform_o_0);
wire signed [7:0] cos_lut_o;
cos_lut cos_lut_0 (
    .addr(cos_addr),
    .cos_out(cos_lut_o)
);

wire signed [7:0] cos_o;
assign cos_o = cos_sign ? -cos_lut_o : cos_lut_o;
```

为了有效处理计算和存储，Box-Muller 变换的实现采用了定点数表示，通过算术左移和右移操作，在不溢出的情况下平衡了计算精度和速度。

3 仿真测试

本次实验使用 Vivado 2017.3 进行仿真测试。

3.1 Hamming 编解码

仿真文件如下：

```
`define PERIOD 10
module hamming_tb();
    reg [3:0] data_i;
    wire [6:0] data_enc;
    wire [3:0] data_dec;
    hamming_encode hamming_encode_0 (
        .data_i(data_i),
        .data_o(data_enc)
    );
    hamming_decode hamming_decode_0 (
        .data_i(data_enc),
        .data_o(data_dec)
    );
    integer i;
    initial begin
        for (i = 0; i < 16; i=i+1) begin
            data_i = i;
            #(`PERIOD)
            if (data_dec ! = data_i) begin
                $display("[Error]: %b -> %b -> %b", data_i, data_enc,
                ↪  data_dec);
            end
            else begin
                $display("Correct: %b -> %b -> %b", data_i, data_enc,
                ↪  data_dec);
            end
        end
    end
endmodule
```

输出如下：

```
Correct: 0000 -> 0000000 -> 0000
Correct: 0001 -> 1110001 -> 0001
Correct: 0010 -> 1010010 -> 0010
Correct: 0011 -> 0100011 -> 0011
Correct: 0100 -> 0110100 -> 0100
Correct: 0101 -> 1000101 -> 0101
Correct: 0110 -> 1100110 -> 0110
Correct: 0111 -> 0010111 -> 0111
```

```
Correct: 1000 -> 1101000 -> 1000
Correct: 1001 -> 0011001 -> 1001
Correct: 1010 -> 0111010 -> 1010
Correct: 1011 -> 1001011 -> 1011
Correct: 1100 -> 1011100 -> 1100
Correct: 1101 -> 0101101 -> 1101
Correct: 1110 -> 0001110 -> 1110
Correct: 1111 -> 1111111 -> 1111
```

这表明 Hamming 码编解码模块实现正确。

3.2 交织与解交织

仿真文件如下：

```
module test_IL();
    reg clk;
    reg rst;
    reg en;
    reg [28 -1:0] data = 28'b0011111000001111010111100101;
    wire en_IL;
    wire [28 -1:0] data_IL;
    interleaver il(clk, rst, en, data, en_IL, data_IL);
    wire [28 -1:0] data_DIL;
    wire en_DIL;
    deinterleaver dil(clk, rst, en_IL, data_IL, en_DIL, data_DIL);
    initial begin
        clk <= 1;
        rst <= 0;
        en <= 0;
        #10 en <= 1;
        rst <= 1;
    end
    always #5 clk = ~clk;
endmodule
```

仿真结果如图 4 所示。

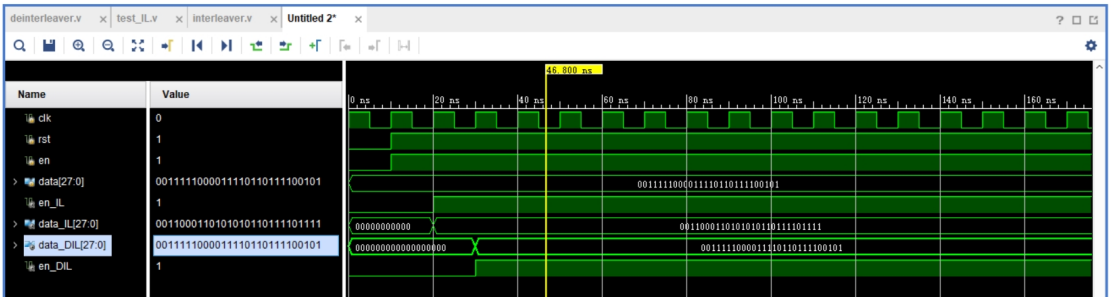


图 4: 交织与解交织仿真波形

如图 5 所示，按照“按行写入，按列读出”这一规则处理输入序列 0011111 0000111 1011011 1100101，得到的交织器输出序列应为 0011000 1101010 1011011 1101111，与图 4 中的 data_IL 波形一致，说明交织器设计正确，且 data_DIL 和输入序列 data 相同，说明解交织器设计正确。

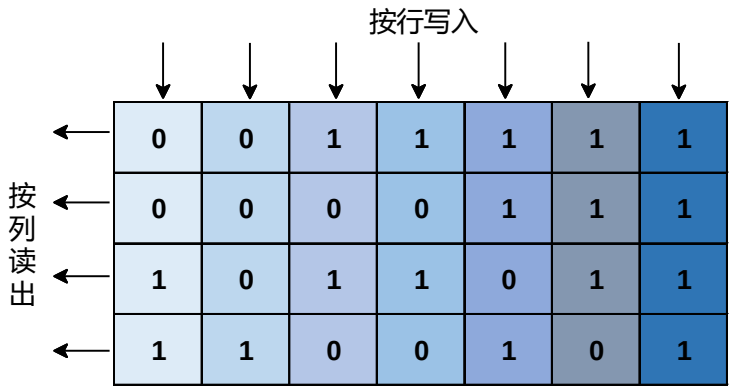


图 5: 交织过程演示

3.3 QPSK 调制解调

仿真文件如下：

```
module test();
    parameter clk_period_100M = 10; // f=100MHz <=> T=10ns
    reg rst;
    reg clk;
    reg [15:0] datas = 16'b1110110110001101;
    wire [1:0] data_i;
    assign data_i = datas[1:0];
    wire signed [15:0] data_o_i;
    wire signed [15:0] data_o_q;
    qpsk_modulator qpsk_modulator_0 (
        .clk(clk),
        .rst(rst),
        .data_i(data_i),
        .data_o_i(data_o_i),
        .data_o_q(data_o_q)
    );
    wire [1:0] data_o;
    qpsk_demodulator qpsk_demodulator_0 (
        .clk(clk),
        .rst(rst),
        .data_i_i(data_o_i),
        .data_i_q(data_o_q),
        .data_o(data_o)
    );
```

```
    initial begin
        rst <= 0;
        clk <= 1;
        #(clk_period_100M) rst <= 1;
        #(clk_period_100M/2)
        while (datas > 0) begin
            datas <= datas >> 2;
            #(clk_period_100M) ;
        end
    end
    always #(clk_period_100M/2) clk <= ~clk;
endmodule
```

仿真结果如图 6 所示。

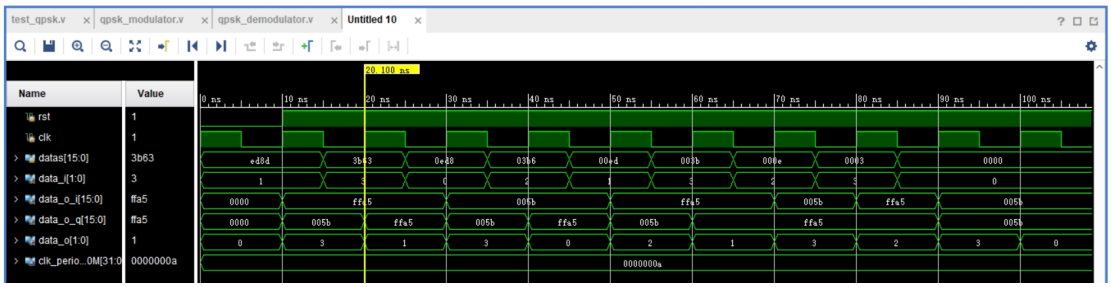


图 6: QPSK 调制解调器仿真波形

从第三个时钟周期开始，每个时钟周期的 QPSK 解调器的输出 data_o 都与前一个时钟周期的 QPSK 调制器的输入 data_i 相等，这说明 QPSK 调制/解调器的设计正确。

3.4 AWGN 信道模型

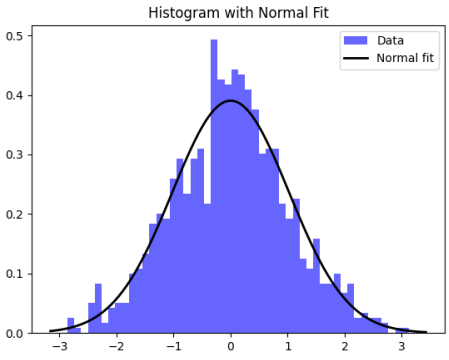
我们保存了仿真产生的高斯白噪声数值，并进行了正态分布假设检验

```
integer log_file;
initial log_file = \$fopen("simulation_log.txt", "w");
always @(posedge clk) begin
    \$fwrite(log_file, "%d\n",
    ↪  gaussian_noise_channel_0.gaussian_o_0);
    \$fflush(log_file);
end
```

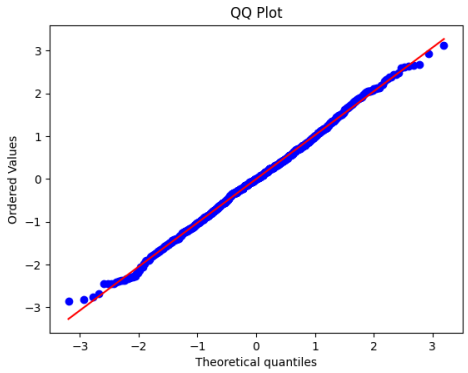
得到高斯白噪声序列（1000 条），并进行正态分布检验，结果如图 7 所示。可见，概率密度分布和 QQ 图均符合高斯分布的特性。

下面进行严格的正态性检验：

首先进行 Shapiro-Wilk 正态性检验，得到统计量 $W = 0.998$，$p = 0.310$，因此我们无法拒绝高斯分布的假设；同时进行 Kolmogorov-Smirnov 正态性检验，得到统计量 $D = 0.027$，$p = 0.462$，同样无法拒绝高斯分布的假设。



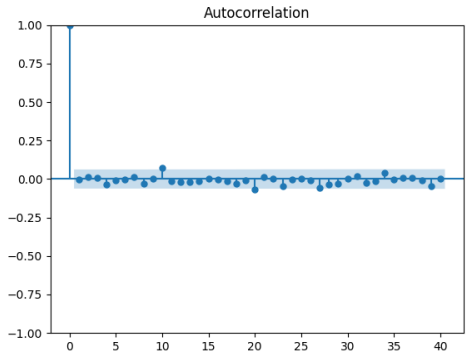
(a) 高斯白噪声概率密度分布可视化



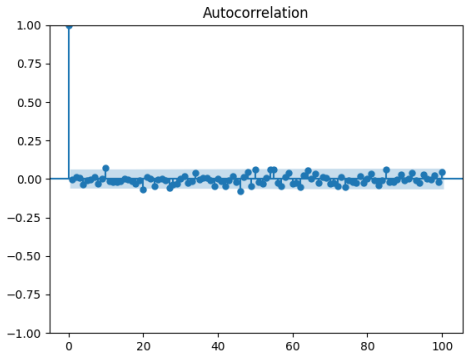
(b) 高斯白噪声 QQ 图

图 7: 高斯白噪声分布检验

最后检验自相关性，如图 8 所示，分别取 lag 为 20 和 40，自相关性函数均在 0 附近波动，且绝大部分位于 95% 置信区间内，因此我们可以认为噪声序列是独立同分布的。



(a) 高斯白噪声 ACF（lag=40）



(b) 高斯白噪声 ACF（lag=100）

图 8: 高斯白噪声自相关性检验

3.5 联合仿真

仿真文件如下：

```
module test_top();
    reg clk_origin;
    reg rst;
    reg [15:0] button;
    wire [15:0] hamming_dec;
    wire hamming_wave;
    wire interres_wave;
    initial begin
        clk_origin <= 1;
        button <= 16'b0001_0100_0111_1100;
```

```
        rst <= 0;
        #5
        rst <= 1;
    end
    always #10 clk_origin = ~clk_origin;
    top top_0 (
        .clk_origin(clk_origin),
        .rst(rst),
        .button(button),
        .hamming_dec(hamming_dec),
        .hamming_wave(hamming_wave),
        .interres_wave(interres_wave)
    );
endmodule
```

仿真结果如图 9 所示。

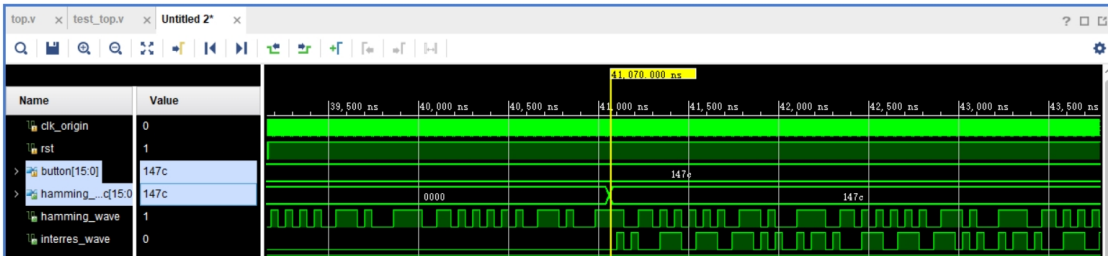


图 9: 联合仿真波形

读图可知，系统最终的输出 hamming_dec 与输入 button 一致，说明系统实现正确。

4 实验效果

本实验通过设计不同信道条件下的通信系统，评估了海明码编解码、交织解交织、QPSK调制方式以及高斯噪声信道对系统性能的影响。实验环境涵盖了无噪声理想信道、突发连续错误信道和高斯噪声干扰信道。

4.1 无噪声理想信道

首先我们设计了无噪声、无突发错误的理想信道条件，以验证系统基本功能的正确性。

通过实验设置中的开关（图 10），我们可以手动输入任意 16 比特的信号，其中每个开关的上拨表示 1，下拨表示 0。每次按下重置键后，输入信号通过系统进行处理，并通过 16 个 LED 灯输出，灯亮表示 1，熄灭表示 0；注意本实验没有用到数码管。

从实验结果可见，输入信号与输出信号完全一致，验证了系统在理想条件下能够完美实现信号传输。

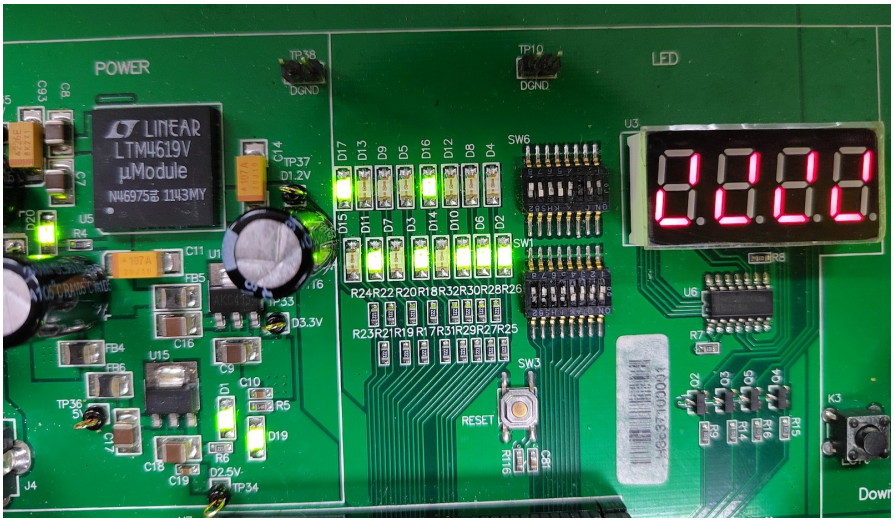


图 10: 无噪声系统传输结果

4.2 突发连续错误

接着，为了验证海明码、交织的功能，我们为输入信号加入了突发连续错误。

突发错误通常会导致多个连续比特发生错误，严重影响接收信号的准确性。为了模拟这一情况，我们通过编写代码引入连续错误，从而模拟信号中出现突发错误的情形。具体来说，突发错误通过以下代码实现：

```
wire [27:0] inter_res_error;
assign inter_res_error =  {!inter_res[27], !inter_res[26],
↪  !inter_res[25], !inter_res[24], inter_res[23:0]};

...

case (qp_cnt)
    1: qpsk_in <= inter_res_error[1:0];
```

```
    2: qpsk_in <= inter_res_error[3:2];
    3: qpsk_in <= inter_res_error[5:4];
    4: qpsk_in <= inter_res_error[7:6];
    5: qpsk_in <= inter_res_error[9:8];
    6: qpsk_in <= inter_res_error[11:10];
    7: qpsk_in <= inter_res_error[13:12];
    8: qpsk_in <= inter_res_error[15:14];
    9: qpsk_in <= inter_res_error[17:16];
    10: qpsk_in <= inter_res_error[19:18];
    11: qpsk_in <= inter_res_error[21:20];
    12: qpsk_in <= inter_res_error[23:22];
    13: qpsk_in <= inter_res_error[25:24];
endcase
```

代码中从第 24 位开始，连续 4 位的比特被反转，模拟了一个突发错误。

为了应对突发错误，通常可以采用海明码进行错误检测与纠正，并结合交织技术来增强系统的抗干扰能力。交织技术通过重新排列数据比特，将连续错误分布得更加均匀，从而减轻了多个连续错误对系统性能的影响。经过交织处理后，信号传输至接收端，接收端利用海明码对错误进行检测与纠正。海明码能够有效纠正单个比特错误，结合交织技术后，系统在处理多个连续错误时展现出了显著的恢复能力。

实验表明，经过海明码和交织技术的处理，接收端成功恢复了原始信号，证明了这两种技术在面对突发连续错误时具有良好的性能，显示了系统的可靠性。实验结果与图 10 相同，此处不再重复展示。

图 11 展示了信号在通过解交织器前后的波形图，可以看到 bit 串经过了解交织器的重新排列，使得连续突发错误被分散到多个待解码串中。

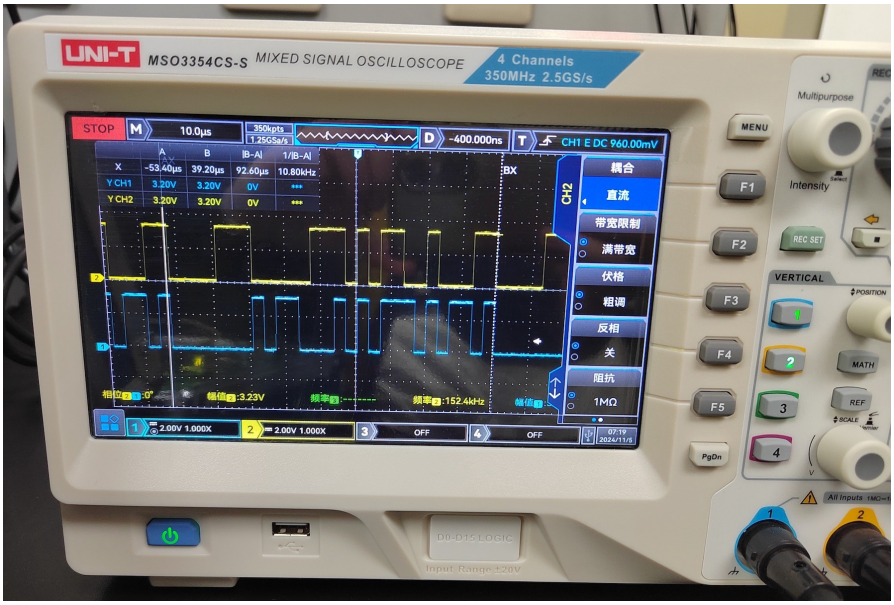


图 11: 通过解交织器前后的信号波形

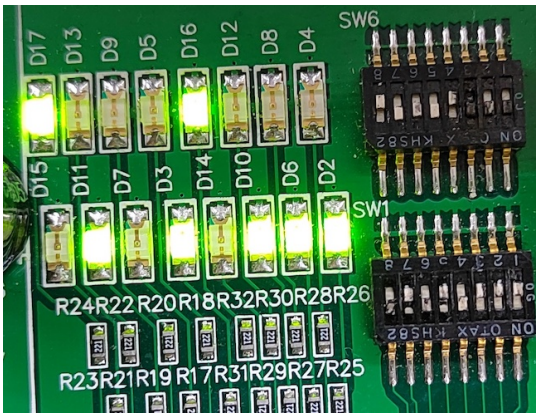
4.3 高斯噪声信道

为全面评估系统在不同信道条件下的性能，我们还模拟了高斯噪声信道。通过调整高斯噪声的功率，模拟噪声功率对误码率（BER）的影响。实验结果表明，在理想信道条件下，系统的误码率接近零；而在高斯噪声影响下，随着噪声功率的增大，误码率逐渐增高。

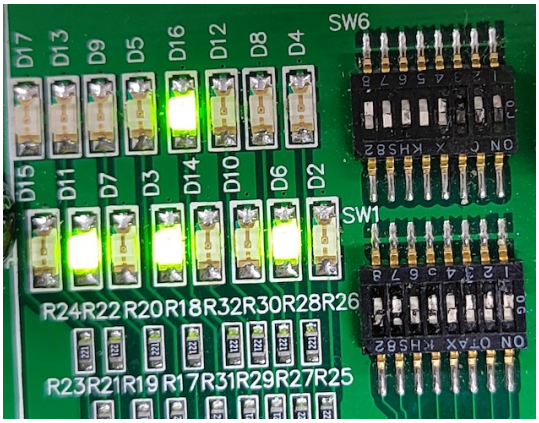
表 1 展示了不同信道条件下的误码率变化，表明噪声功率对系统性能具有重要影响，尤其在较低信噪比（SNR）下，系统的误码率显著增加。

噪声相对功率	误码比例	误码率百分比
1/4	0/16	0%
7/16	3/16	19%
1/2	4/16	25%
1	4/16	25%

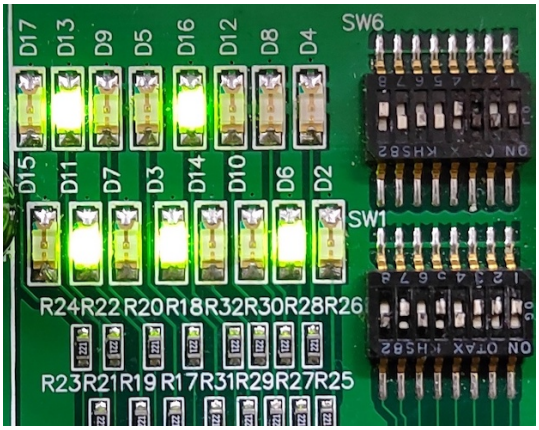
表 1: 不同噪声（相对）功率下的误码率



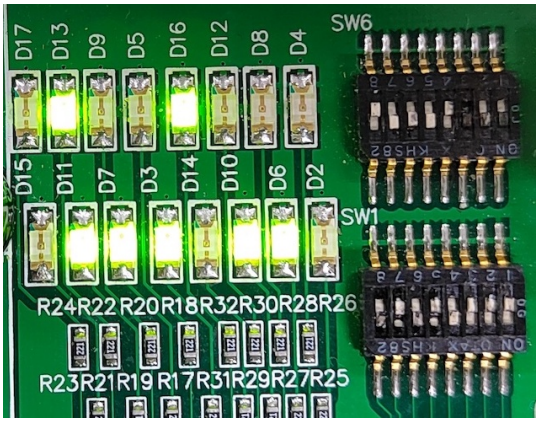
1/4 噪声，无错误



7/16 噪声，3 位误码



1/2 噪声，4 位误码



1 噪声，4 位误码

图 12: 不同噪声（相对）功率下的误码情况

4.4 总结

通过本实验，我们综合评估了海明码、交织技术、QPSK 调制和高斯噪声信道对通信系统性能的影响。在无噪声理想信道下，系统能够实现几乎完美的信号传输；在突发错误信道中，海明码和交织技术的结合有效降低了误码率，恢复了原始信号；而在高斯噪声信道下，增加信噪比是提高系统性能、降低误码率的关键因素。

实验结果表明，交织技术在面对突发错误时有显著的性能提升作用，而海明码和交织技术的结合能有效提高系统的可靠性。此外，高斯噪声对系统性能的影响较大，因此提高信噪比仍是应对噪声干扰的主要手段。

本次实验也加深了我们对通信实验技能的掌握，在动手实践的过程中我们逐步熟悉了 verilog 的语法细节、测试文件的编写、仿真和烧录的方法等等。在此感谢老师、助教和同学在我们实验过程中耐心的答疑解惑，让我们能够圆满完成此次实验。

5 成员分工

- Hamming 编码、解码：向明义 2021012760
- 交织、解交织：游笑权 2021012763
- QPSK 调制解调、AWGN 信道：陈子熠 2022010503
- 联调：全体成员