第1章　数据表示实验

1.1 补码表示实验

1.1.1 实验目的

学生理解补码基本特性，能分析简单 C 语言程序中与数据表示相关的执行行为，并通过实际代码执行验证真实计算机中数据表示的编码体系，掌握获取 C 语言变量机器码的方法，培养学生数据表示层面基本的软硬协同系统观。

1.1.2 背景知识

人类历史上第一台数字计算机 ENIAC 采用十进制进行运算，目前计算机普遍采用二进制进行数据表示和运算，二进制可以表示任何数据和信息，且状态数最少，运算简单，便于物理实现。二进制数与十进制数一样有正负之分，日常生活中采用“＋”或“－”表示的数据称为真值，在计算机中，通常将数的符号和数值一起编码来表示数据，这种用“0”或“1”表示数据符号位的编码称为机器码，常用的机器码分为原码、反码、补码、移码。

设 x 为 n 位定点数（不含符号位），则 x 的原码、反码、补码和移码的定义如表 1.1 所示：

表 1.1 四种机器码定义

	（定点整数） $-2^n < x \le 0$	（定点小数） $-1 < x \le 0$	$0 \le x \le 2^n - 1$ $0 \le x \le 1 - 2^{-n}$	备注
原 码	$2^n + \lvert x \rvert$	$2 + \lvert x \rvert$	x	最直观，增加一个符号位
反 码	$2^{n+1} + x - 1$	$2 + x - 2^{-n}$	x	符号位同原码，为负数时数值位逐位取反
补 码	$2^{n+1} + x$	$2 + x$	x	反码末位加一得到补码，多表示一个负数
移 码	$2^n + x$	无	$2^n + x$(整数时)	补码符号位取反得到移码

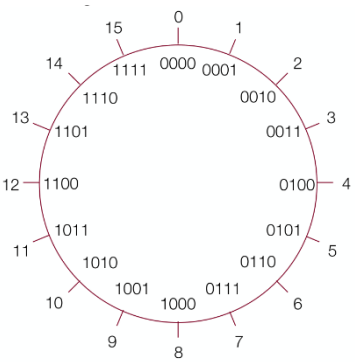
(1) 原码表示最为直观，仅仅是在数据位前面增加一个符号位，“0”表示正数，“1”表示负数，其在数轴上表示区间对称，存在正零和负零两个零，运算比较复杂，目前在计算机中主要用于表示浮点数的尾码部分，所以浮点数存在正零和负零两个不同的编码。

(2) 反码符号位与原码相同，数值表示范围也相同。当真值为正数时，数值部分与真值相同，当真值为负数时，数值部分可以通过对真值逐位取反得到，反码末位加一即可得到补码，而反码在二进制电路中非常容易得到，所以反码通常用于补码减法运算中减数的求补过程。

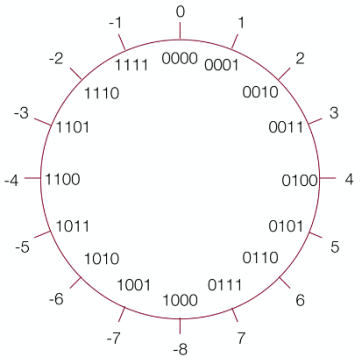
(3) 补码数据表示建立在“模”的概念基础上,模的值即为符号位进位所在位的权值。对于定点小数而言，模的值为 2^1 即 2,对于定点整数而言,模的值为 2^{n+1},其中 n 为不包含符号位的整数位数。当真值为负数时，补码等于真值加上模，补码这种特性使得补码符号位可以直接参与加减运算，运算电路实现方便，另外补码只有唯一的零，所以在表示范围上比其它几种机器码

多表示一个最小负数。目前计算机中整数采用补码进行存储、表示和运算。

(4) 移码主要用于表示浮点数的阶码，IEEE754 浮点数标准中使用的移码和表 1.1 中略有差异。



4 位无符号数



4 位有符号数(补码)（图另外截获）

图 1.1 四位整数表示循环圈

在 C 语言中，整型数据分为无符号和有符号两种，为便于描述，图 1.1 给出了 4 位整数表示的循环圈，实际上 32 位整型变量原理也完全相同。左侧图为无符号数循环圈，从图中可以看出，真值和二进制机器码相等，当两个数相加结果超过 2^4-1=15 时，结果将在循环圈顶部沿顺时针方向循环，运算结果将小于相加数，产生溢出。同理当两个数据相减结果小于零时，结果将在循环圈顶部沿逆时针方向循环，运算结果大于被减数，产生溢出。

右图是采用补码表示的 4 位有符号数表示循环圈，当真值为正数时，二进制的机器码和十进制的真值相同，-1 的机器码是全 1，补码公式为 2^4-1=1111（模为 2^4），-2 的机器码为 2^4-2=1110。最大正数是 2^4-1=0111, 两个正数相加如果超过这个数值，则会在循环圈底部顺时针进入负数区域，运算结果不正确，产生了正溢出。最小负数是-2^3=-8，两个负数相加结果如果小于最小负数，则会在循环圈顶部顺时针进入正数区域，结果会变成正数，运算结果不正确，产生了负溢出。实际上 C 语言程序并不检测这种加、减、乘等运算的溢出现象，程序员应尽量避免出现这种情况，对溢出应通过程序进行判断。

表 1.2 整形变量取值范围

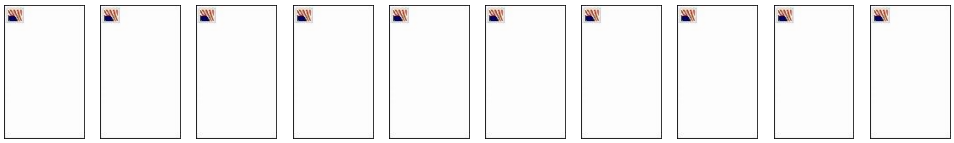
Value		char(8 位)	short(16 位)	int(32 位)	long(64 位)
无符号最大值	Hex	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	Dec	255	65,535	4,294,967,295	18,446,744,073,709,551,615
有符号最小值	Hex	0x80	0x8000	0x80000000	0x8000000000000000
	Dec	－128	－32,768	－2,147,483,648	－9,223,372,036,854,775,808
有符号最大值	Hex	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	Dec	127	32,767	2,147,483,647	9,223,372,036,854,775,807
-1	Hex	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	Hex	0x00	0x0000	0x00000000	0x0000000000000000

8 位、16 位、32 位、64 位的整型变量基本原理与上面的 4 位整数完全一样，其对应取值

范围见表 1.2。

1.1.3 实验内容

（1） 小练习：如果给你 10 个信封和 1000 张一元的纸币，如何将 1000 元钱分装到 10 个信封中，才能保证不管给出 1-1000 的任何一个数字，都可以直接从 10 个信封中直接拿走若干信封，而拿走信封里面的钱的总数正好等于这个数字？请在下面 10 个信封上写上具体钱数？



（2）仔细阅读如下 C 语言代码，结合数据表示相关知识，分析程序功能。

```
//file: machine_code_output.c
  #include "stdio.h"

  //输出字符的十六进制编码
  void char_hex_out(char a)
  {
    const char HEX[]="0123456789ABCDEF";
    int index=a&0x0F;
    printf("%c%c", HEX[(a&0xF0)>>4],HEX[a&0x0F]);
    }

  //输出4个字节数据的十六进制编码，可用于输出4字节变量的机器码
  void four_byte_out (char *addr)
  {
    char_hex_out (*(addr +3));     //输出指针变量的值，指针本质上是内存地址，无符号数
    char_hex_out (*(addr +2));
    char_hex_out (*(addr +1));
    char_hex_out (*(addr +0));
    printf("\n");
  }

  main()
  {
    int a= -1;
```

```
  int b=2147483648;                  //2147483648=2^31       4294967296=2^32
  int c= -b;
  unsigned int d = -2147483648;
  printf ("a = %u = %d = 0x%x    \n",a,a,a);
  printf ("b = %u = %d = 0x%x    \n",b,b,b);
  printf ("c = %u = %d = 0x%x    \n",c,c,c);
  printf ("d = %u = %d = 0x%x    \n",d,d,d);

  printf("\nd's memory addr    = 0x%x",&ui);   //输出变量d的虚存地址
  printf("\nd's machine code = 0x");
  four_byte_out (&ui);          //输出变量d的机器码

  return;
}
```

上述代码输出如下，请根据自己的分析和理解完成程序的输出。

a = _____ = _____ = 0x_____
b = _____ = _____ = 0x_____
c = _____ = _____ = 0x_____
d = _____ = _____ = 0x_____

d's memory addr   = 0x_____
d's machine code   = 0x_____

在 C 程序开发环境中编译运行以上代码，验证你自己的分析结果，如有不同，请思考原因，并找出依据。

（3）仔细阅读如下 C 语言代码，结合数据表示相关知识，分析程序功能。

```
//file:    int_add_overflow.c
#include "stdio.h"
main()
{
  int a= 1; int b=2147483647;          //2147483648=2^31       4294967296=2^32
  int d=-1; int e=-2147483648;
  int c= a+b;
  int f=e+d;
  printf ("c = a + b = %d + %d = %d    \n",a,b,c);
  printf ("f = e + d = %d + %d = %d    \n",d,e,f);
```

```
    return;
}
```

上述代码输出如下，请根据自己的分析和理解完成程序的输出。

c = a + b = _____ + _____ = _____
f = e + d = _____ + _____ = _____

在 C 程序开发环境中编译运行以上代码，验证你自己的分析结果，如有不同，请思考原因，并找出依据。

（4）仔细阅读如下 C 语言代码，结合数据表示相关知识，分析程序功能。

```
//file:   get_var_size.c
#include "stdio.h"
# include "stdlib.h"
int main ()
{
  int iarray [] = {5 , 4 , 3 , 2 , 1};
  char * iarr = ( char *) iarray ;
  printf (" sizeof ( char ): %d\n", ( int) sizeof ( char ) ) ;
  printf (" sizeof ( short ): %d\n", ( int ) sizeof ( short ) ) ;
  printf (" sizeof (int ): %d\n", ( int) sizeof (int ) ) ;
  printf (" sizeof ( unsigned int ): %d\n", (int ) sizeof ( unsigned int ) ) ;
  printf (" sizeof ( long ): %d\n", ( int) sizeof ( long ) ) ;
  printf (" sizeof ( long long ): %d\n", (int ) sizeof ( long long ) ) ;
  printf (" sizeof ( size_t ): %d\n", (int ) sizeof ( size_t ) ) ;
  printf (" sizeof ( void *): %d\n", (int ) sizeof ( void *) ) ;
  printf (" sizeof ( iarray ): %d\n", (int ) sizeof ( iarray ) ) ;
  printf (" sizeof ( iarr ): %d\n", ( int) sizeof ( iarr ) ) ;
  return 0;
}
```

上述代码输出如下，请根据自己的分析和理解完成程序的输出。

sizeof (char): _____
sizeof (short): _____
sizeof (int): _____
sizeof (unsigned int): _____

sizeof (long): _____

sizeof (long long): _____

sizeof (size_t): _____

sizeof (void *): _____

sizeof (iarray): _____

sizeof (iarr): _____

在 C 程序开发环境中编译运行以上代码，验证你自己的分析结果，如有不同，请思考原因，并找出依据。

1.1.4 实验思考

(1) 程序 1 中变量 b 和 c 是否相等，为什么？
(2) 程序 1 中变量 d 的内存地址输出的值是否是一个确定的值，其值能否被 4 整除，为什么？
(3) C 语言中 int，short，char，long 型变量在内存中存储的机内码是采用什么编码？
(4) C 语言中 int 变量是采用小端还是大端方式存储？小端大端存储方式由什么来决定？

1.2 浮点数表示实验

1.2.1 实验目的

学生理解 IEEE754 编码基本规则和特点，能分析程序中浮点运算的执行结果，并利用程序实际执行验证真实计算机中浮点数据的表示规则，了解 IEEE754 表示容易引起的一些问题。

1.2.2 背景知识

IEEE754 浮点数标准是目前主流计算机所采用的浮点数标准，主要包括 32 位单精度浮点数和 64 位双精度浮点数，分别对应 C 语言中的 float 型和 double 型。IEEE754 浮点数据由符号位 S、阶码 E 和尾数 M 组成，具体形式如图 1.2 所示：

图 1.2 IEEE754 浮点数格式

单精度浮点数共 32 位，其中 S 占 1 位，E 占 8 位，M 占 23 位；双精度浮点数共 64 位，其中其中 S 占 1 位，E 占 11 位，M 占 52 位。

阶码字段 E，单精度浮点数采用偏移值为 127 的移码表示，双精度浮点数阶码偏移值为 1023。尾数字段 M 小数点左边包含一位隐藏位 1，即尾数的实际有效位数为 24/53 位，完整的尾数形式为 1.M，但在进行浮点数据表示时只保存 M。正是由于要将位数变成 1.M，阶码的偏

移值才为 127/1023 而不是 128/1024。

随 E 和 M 的取值不同，IEEE754 浮点数据表示具有不同的意义，表 1.3 给出了 IEEE754 单精度浮点数在不同 S，E，M 取值下的具体表示意义。

表 1.3 IEEE754 单精度浮点数规范

符号位 S(1 位)	阶码 E（8 位）	尾数 M（23 位）	表示
0/1	255	1XXXX	NaN 非数
0/1	255	0XXXX	sNaN 发信号的非数(尾数不全为零)
0	255	*0*	+∞
1	255	*0*	-∞
0/1	1~254	*m*	$(-1)^s\times1.m\times2^{e-127}$ 规格化数
0/1	0	*m* （非 0）	$(-1)^s\times0.m\times2^{-126}$ 非规格化
0/1	0	0	+0/-0

从表 1.3 可以看出，浮点数可以表示正无穷和负无穷，另外还可以表示非数 NaN，为进一步提升浮点数表示精度，还引入了非规格化数，浮点数尾码采用原码表示，所以存在正零和负零两个零，单精度双精度浮点数表示范围见表 1.4。

表 1.4 IEEE754 单精度双精度浮点数表示范围

格式	最小值	最大值
单精度规格化 $(-1)^s\times1.m\times2^{e-127}$	$E_{min}=1$, M=0, $1.0\times2^{1-127}=2^{-126}$	$E_{max}=254$, M=1.1111…1×$2^{254-127}$ $=2^{127}\times(2-2^{-23})\approx+3.4\times10^{38}$
单精度非规格化 $(-1)^s\times0.m\times2^{-126}$	E=0, M=2^{-23}, $2^{-23}\times2^{-126}=2^{-149}$	E=0, f=0.1111…1×2^{-126} $=2^{-126}\times(1-2^{-23})$
双精度规格化 $(-1)^s\times1.m\times2^{e-1023}$	$E_{min}=1$, M=0, $1.0\times2^{1-1023}=2^{-1022}$	$E_{max}=2046$, f=1.1111…1×$2^{2046-1023}$ $=2^{1023}\times(2-2^{-52})\approx+1.8\times10^{308}$
双精度非规格化 $(-1)^s\times0.m\times2^{-1022}$	E=0, M=2^{-52}, $2^{-52}\times2^{-1022}=2^{-1079}$	E=0, M=0.11111…1×2^{-1022} $=2^{-1022}\times(1-2^{-52})$

IEEE754 单精度浮点数与对应真值之间的变换流程如图 1.3 所示。

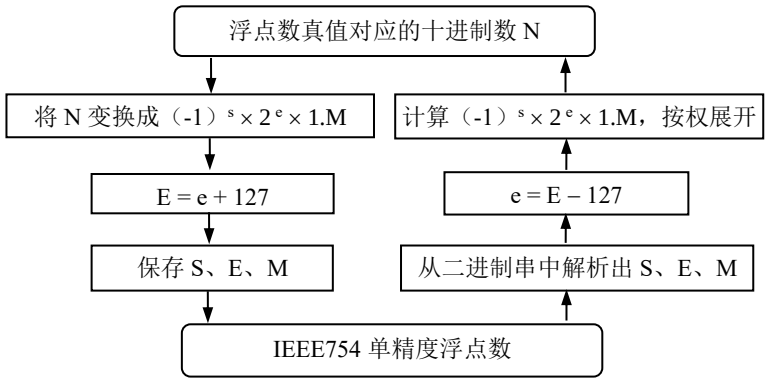


图 1.3 IEEE754 单精度浮点数与真值间的变换流程

值得注意的是，十进制小数大多不能精确转换成二进制，如 0.1、0.2、0.3、0.4 等在转换成二进制小数时都会变成循环小数，所以即使有再多的尾数位也无法精确表示这些十进制数，浮点数并不能精确表示所有十进制数。另外由于 IEEE 754 浮点数采用阶码和尾码的形式表示浮点数，所以浮点数在数轴上的刻度分布并不像定点整数一样是均匀的，图 1.4 是单精度浮点数在数轴上的刻度分布，零右边的阴影区是非规格化数区域，越往右，浮点数的分布越稀疏，这也导致浮点数的加法运算不满足结合律，小数和大数相加时小数可能会被吸收。

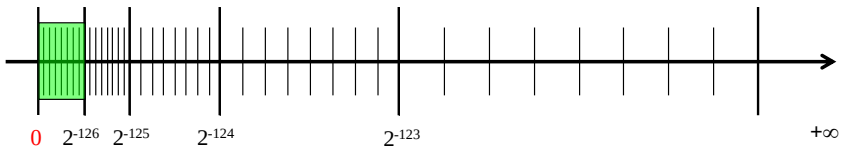


图 1.4 IEEE754 单精度浮点数在数轴上的刻度分布

1.2.3 实验内容

（1） **小练习**：假设你已经身家千万，请问你最少有多少钱（元为单位的整数）才能导致 32 位的单精度浮点数无法精确表示，请问你最少有多少钱才能导致 64 位的双精度浮点数无法精确表示，写一个小程序将这两个数字用十进制打印出来，同时请精确输出 32 位单精度浮点数的最小非规格化数、最大非规格化数最小非规格化数的十进制值。

（2） **无法表示的整数**：运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
// file : unrepresentable .c
# include < stdio .h>
# include < stdlib .h>
int main ( int argc , char ** argv )
{
    float f1 = 16777216.0;
    float f2 = 16777217.0;
```

```
    float f3 = 16777218.0;
    printf (" 16 ,777 ,216: %f\n",f1);
    printf (" 16 ,777 ,217: %f\n",f2);
    printf (" 16 ,777 ,218: %f\n",f3);
    printf ("f1 == f2? %s",f1 == f2 ? " true " : " false ");
    return 0;
}
```

（3）**整数除零问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
//file: int_div_by_zero.c
#include "stdio.h"
main()
{
  int a=1; a=a/0;
  printf("a=%d",a);
  return;
}
```

（4）**浮点数除零问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
//file: float_div_by_zero.c
#include "stdio.h"
main()
{
   float a=0.0, b;
   a=a/0;    b=-sqrt(-1);
   printf ("a=%f b=%f",a,b);          //输出值是什么意思？
   return;
}
```

（5）**浮点数双零问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
#file: float_two_zero.c
#include "stdio.h"
#include "stdint.h"
union
{
  char c[4];                //联合体，多变量共享存储空间
  float f;
  int i;
```

```
} t1,t2;

int main()
{
   t1.i=0X80000000;     //直接机器码赋值，如解释为浮点数为负零
   t2.i=0X00000000;     //直接机器码赋值，如解释为浮点数为正零
   if (t1.f==t2.f)
     printf("float data is equal\n");
   if (t1.i!=t2.i)
     printf("int data is not equal\n");
}
```

（6）**浮点数的精度问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
// file : double_comparisons .c
# include < stdio .h>
# include < stdlib .h>
int main ( int argc , char ** argv )
{
  double a = 0.1;
  double b = 0.2;
  double c = 0.3;
  double d = a + b;
  printf ("a (0.1) : %.30 g\n",a);
  printf ("b (0.2) : %.30 g\n",b);
  printf ("c (0.3) : %.30 g\n",c);
  printf ("d (0.3) : %.30 g\n",d);
  printf ("c == d? %s\n", c == d ? " true " : " false ");
  printf ("c < d? %s\n", c < d ? " true " : " false ");
  return 0;
}
```

（7）**浮点数运算误差问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
#include "stdio.h"
test_float_cal()
{
   float a,b,c;     int d;
   b=3.3;     c=1.1;
   a=b/c;
```

```
    d=b/c;
    printf("a=%f,d=%d",a,d);
    if (3.0==a)
        printf("\nFloat a=3.3/1.1==3.0");
}

test_double_cal()
{
    double a,b,c;      int d;
    b=3.3;      c=1.1;
    a=b/c;
    d=b/c;
    printf("\n\na=%f,d=%d",a,d);
    if (3.0!=a)
      printf("\nDouble a=3.3/1.1 != 3.0");
}

main()
{
    test_float_cal();
    test_double_cal();
}
```

（8）**浮点数结合律问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
  // file : float_associativity .c      example from standford cs107
  # include < stdio .h>
  # include < stdlib .h>
  int main ( int argc , char ** argv )
  {
    float a = 3.14;
    float b = 1e20;
    printf (" (3.14 + 1e20 ) - 1e20 = %f\n", (a + b) - b);
    printf (" 3.14 + (1 e20 - 1e20 ) = %f\n", a + (b - b));
    return 0;
  }
  (3.14 + 1e20) - 1e20 = 0.000000
  3.14 + (1e20 - 1e20) = 3.140000
```

（9）**浮点数溢出问题：**运行下面的 C 语言程序，观察运行输出结果，并解释运行现象。

```
// file : float_add_overflow.c
# include < stdio .h>
union
   {
      char c[4];
      float f;
      int i;
  }   t1,t2,t3,t4,;

void float_add_overflow()
{
   t1.i=0x7F000000;       t2.i=0x7F000000;
   t3.f=t1.f+t2.f;
   char_hex_out (&t1.f);                  //该函数见1.1.3节程序中的定义
   char_hex_out (&t2.f);
   char_hex_out (&t3.f);
   printf("t1 = %.60f \n",t1.f);
   printf("t2 = %.60f \n",t2.f);
   printf("t1+t2 = %.60f \n",t3.f);
}

void float_sub_overflow()
{
   t1.i=0x00C00000;     t2.i=0x00800000;
   t3.f=t1.f-t2.f;
   char_hex_out (&t1.f);
   char_hex_out (&t2.f);
   char_hex_out (&t3.f);
   printf("t1 = %.61f \n",t1.f);
   printf("t2 = %.61f \n",t2.f);
   printf("t1-t2 = %.61f \n",t3.f);
}

main()
{
   float_add_overflow();
   float_sub_overflow();
```

```
}
```

1.2.4 实验思考

（1） 请问银行如何表示一毛钱，一分钱，如果用浮点数表示会存在什么可能的问题？
（2） 在编写 C 语言程序时，浮点数适不适合进行比较，为什么？

1.3 汉字编码实验

1.3.1 实验目的

(1) 学生理解汉字机内码、区位码，最终能通过相关工具批量获取一段文字的 GB2312 机内码，并利用简单电路实现 GB2312 编码与区位码的转换；
(2) 学生了解字形码显示的基本原理，能在实验环境中实现汉字 GB2312 编码的点阵显示。

1.3.2 背景知识

ASCII 编码（American Standard Code for Information Interchange）以及 GB2312 编码都是机内码，是计算机内存储和处理字符时使用的编码。ASCII 码是国际通用的字符编码，包括 128 个常用符号，其中 33 个控制符、10 个数字、52 个英文字母、33 个专用符号，这 128 个字符使用 7 个比特表示，由于计算机中数据存储以字节为单位，故字节最高位（MSB）为零。

随着计算机的发展一些非英语国家也开始使用计算机，此时 128 个字符就不够使用，比如法国就将 ASCII 编码扩展为 8 位用于表示法语，称为扩展 ASCII 编码。汉字数量众多，为此汉字编码采用了双字节编码，为与 ASCII 编码兼容并区分，汉字编码双字节的最高位都为 1，也就是实际使用了 14 位来表示汉字，这就是 1980 年颁布的 GB2312 标准，也称为国标码。

GB2312 编码用两个字节来表示汉字，理论上可以表示 2^{16}=65536 个汉字，但由于两字节最高位必须为 1，所以实际只能表示 2^{14}=16384 个编码，GB2312 包含 7445 个字符，其中 6763 个常用汉字、682 个全角非汉字字符。为了检索方便，该标准采用 94*94=8836 的二维行列矩阵对字符集所有字符进行了编码，矩阵的每一行称为“区”，每一列称为“位”，区号和行号都从 1 开始编码，采用十进制表示，所有字符都在矩阵中有唯一的位置，这个位置可以用区号和位号合成表示，称为汉字的区位码，如图 1.5 所示，区位码“1818”是“膊”字。区位码和 GB2312 编码之间可以互相转换：区位码+A0A0H=GB2312 编码。但使用区位码比 GB2312 编码更为直观简单，而且在存储汉字字形码字库空间浪费最小，检索更方便。

位号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
16区	啊	阿	埃	挨	哎	唉	哀	皑	癌	蔼	矮	艾	碍	爱	隘	鞍	氨	安	俺	按
17区	薄	雹	保	堡	饱	宝	抱	报	暴	豹	鲍	爆	杯	碑	悲	卑	北	辈	背	贝
18区	病	并	玻	菠	播	拨	钵	波	博	勃	搏	铂	箔	伯	帛	舶	脖	膊	渤	泊
19区	场	尝	常	长	偿	肠	厂	敞	畅	唱	倡	超	抄	钞	朝	嘲	潮	巢	吵	炒
20区	础	储	矗	搐	触	处	揣	川	穿	椽	传	船	喘	串	疮	窗	幢	床	闯	创

图 1.5 汉字区位码表

GB2312 引入的常用汉字较少，很多生僻的人名无法表示，很快 GB2312 中没有使用的一些码位也开始用于表示汉字，但后来还是不够用，为此直接不再要求低字节最高位必须是 1，扩展之后的标准称为 GBK 标准，该标准兼容了 GB2312，同时新增了近 20000 个新的汉字和符号，包括繁体字。后来少数民族文字也被引入到标准中，新增了 4 字节的汉字编码，也就是 GB18030 标准，该标准兼容 GB2312 标准，基本兼容 GBK 标准，共包括 70244 个汉字，支持少数民族文字。

字形码是汉字的输出码，也称字型码，输出汉字时都采用图形点阵的方式，点阵越大汉字显示质量越高，为了能准确地表达汉字的字形，对于每一个汉字都有相应的字形码，甚至不同的字体字形码也不同，汉字字形码按区位码排列，以二进制文件形式存放在存储器中，构成汉字字模字库，简称汉字库。

1.3.3 实验内容

1) 设计国标码转区位码电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成国标码转区位码的子电路设计。其中输入引脚为 16 位的 GB2312 双字节国标码；输出为区号和位号（区号位号均从 1 开始计数），图 1.6 为转换电路引脚定义，请在电路中复制对应隧道信号使用，注意不要增改引脚，不要修改子电路封装，以免影响子电路在其它电路模块中的正常使用。

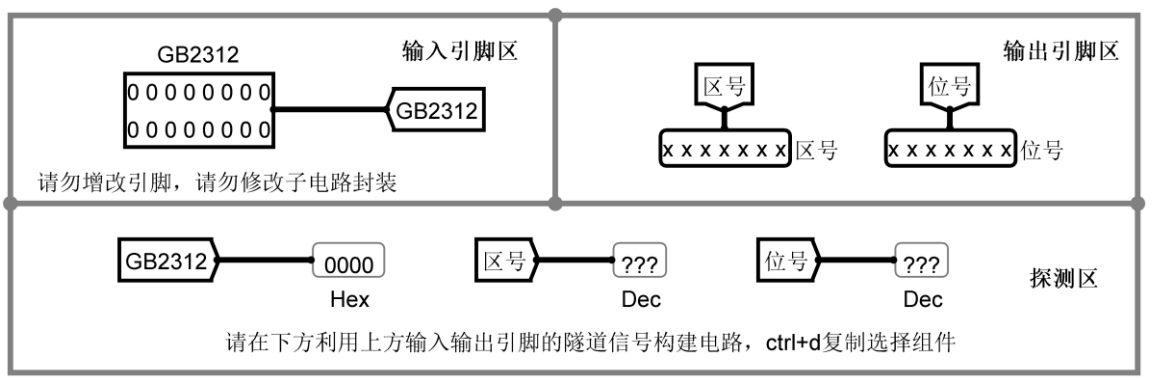


图 1.6 GB2312 转区位码电路引脚定义

2) 汉字 GB2312 编码实验

完成国标码到区位码的转换电路后，可以在汉字显示电路中进行测试，尝试在图 1.7 所示的电路中的 ROM 存储器中存入 100 个成句的汉字（要求与原始数据不同，另外将自己的姓名放在最前面），注意，不允许使用逐字查码表的方式获得编码，ROM 存储器使用方法**错误!未找到引用源。**的 Logisim 参考手册。

该电路中计数器用于顺序生成存放汉字 GB2312 编码的 ROM 组件地址，启用时钟自动仿真后可以右侧 LED 矩阵区域会依次显示 ROM 中的事先预存的汉字内容（时钟自动仿真快捷键：Ctrl+k）。

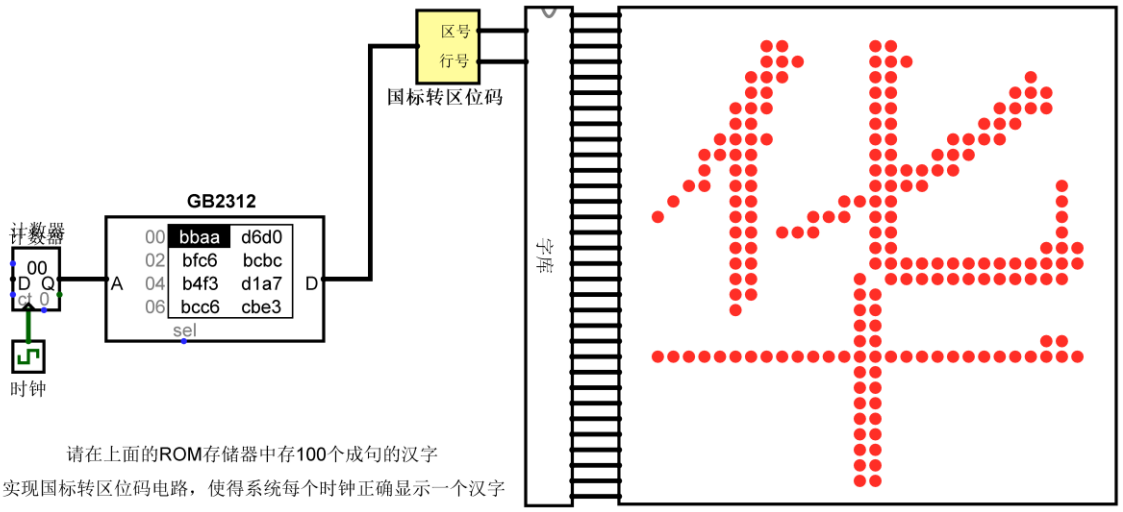


图 1.7 汉字字型码显示电路

1.3.4 实验思考

(1) 除了查找编码表，如何快速批量获取汉字的 GB2312 编码，GB2312 是汉字在机器内的存储编码吗？

(2) Windows 记事本存储汉字的编码有几种，分别对应什么编码标准？

(3) 『鼉』这个汉字能否用 GB2312 表示，为什么？如不能，如何表示这个汉字？

(4) 如何识别一个编码是汉字还是 ASCII 编码，GB2312 编码中是否包含 ASCII 字符？

(5) 图 1.7 中的一个汉字显示信息需要多少位，实验中的汉字显示电路是如何完成的，字库里面包括什么内容，字库为什么要用区位码进行索引，而不是 GB2312 编码进行索引？

(6) 超市针式打印机打印小票上的汉字打印的原理和这里的字形码是否相同，生活中还有那些这样的应用场景？

1.4 奇偶校验实验

1.4.1 实验目的

学生掌握奇偶校验基本原理和特性，能在 Logisim 中实现偶校验编码电路，检错电路。

1.4.2 背景知识

校验码是用于提升数据在时间（存储）和空间两个维度上的传输可靠性的机制，其主要原理是在被校验数据（原始数据）中引入部分冗余信息（校验数据），使得最终的校验码（原始数据+校验数据）符合某种规则，当校验码中某些位发生错误时（原始数据，校验数据都有可能发生错误），会破坏预定规则，从而使得错误可以被检测，甚至可以被纠正，如图 1.8 所示。校验码在生活中有很多的应用，如身份证号，银行卡号，商品条形码，ISBN 号等。

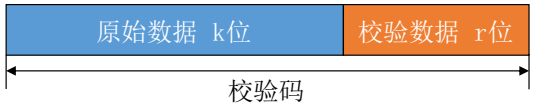


图 1.8 校验码构成

在实际使用过程中，数据校验流程的主要是由发送方对原始数据按照预定规则进行编码，生成包含冗余信息的校验码，校验码经过可能不可靠的传输或存储后，由接收方利用解码模块解析校验码是否符合预定编码规则，如不符合编码规则表明编码出错，需要纠错或者重传。

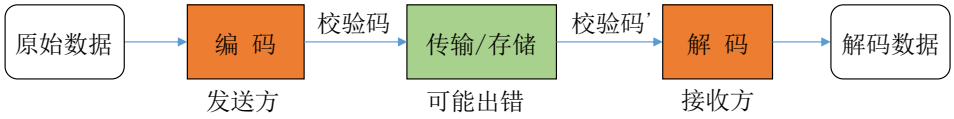


图 1.9 数据校验编解码流程

奇偶校验是一种常见的简单校验方法，其编码规则是引入一位校验位使得最终的校验码中数字 1 的个数保持奇/偶性。奇校验约定的编码规则是让整个校验码（包含原始数据和校验位）中 1 的个数为奇数，而偶校验约定的编码规则是让整个校验码中 1 的个数为偶数，设被校验信息 $D=D_1D_2\ldots D_n$，校验位为 P，偶校验时 P 的逻辑表达式为：

$$P = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_n$$

最终生成的校验码为（$D_1D_2\ldots D_nP$），接收方收到发送方传输的编码后，利用如下公式生成检错码 G。

$$G = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_n \oplus P$$

若 G =1，则表示接收的信息一定有错，数据应丢弃。若 G =0，则表示传送没有出错，严格地说，是没有出现奇数位错；奇偶校验能够检测出任意奇数位的错误，但无法检测偶数位的错误，奇偶校验无法纠错，但其结构简单，编码效率高，当能通过辅助手段确认出错位置时，还可以实现数据纠错所以在内存数据校验，磁盘阵列条带数据校验中还普遍使用。

简单奇偶校验，只有一个校验组，一个校验位，故只能提供一位检错信息进行错误检查，

无法纠错。如果将原始数据信息按某种规律分成若干个校验组，每个数据位至少参加两个以上的校验组，当校验码中的某一位发生错误时，就能在多个检错码中指出，使得偶数位错误也可以被检出，甚至还可以指出最大可能是哪位出错，从而将其纠正，这就是多重奇偶校验的原理。

多重奇偶校验最典型的例子是交叉奇/偶校验，其基本原理是将待编码的原始数据信息构造成行列矩阵式结构，同时进行行列两个方向的奇/偶校验。表 1.5 是一个 4 行 7 列的传输数据组，R_3～R_0 每行产生一个偶校验位 P_r，C_6～C_0 每列产生一个偶校验位 P_c，所有行校验数据 P_r 和列校验数据 P_c 还有一个公共的校验位，这里将生成 $G_{r4}G_{r4}G_{r2}G_{r1}G_{pc}$ 共四个行检错码，$G_{C6}G_{C5}G_{C4}G_{C3}G_{C2}G_{C1}G_{C0}G_{pr}$ 共 8 个列检错码。

表 1.5　交叉奇校验

	C_6	C_5	C_4	C_3	C_2	C_1	C_0	P_r
R_3	1	0	1	0	1	1	0	0
R_2	1	1	1	0	1	1	0	1
R_1	0	0	1	0	0	0	1	0
R_0	1	1	0	0	1	0	0	1
P_c	1	0	1	0	1	0	1	0

当 R_1 的 C_3 出错时，行列两个检错码都会报错，如果能假定是一位错，可以直接通过行列检错码的值定位出错位；当 R_1 的 C_3 和 C_4 同时出错时，R_1 的横向校验组的检错码不会发生变化，因此也检测不到这种错误；但此时 C_3、C_4 的列向检错码会发生变化，可以检测双位错，交叉校验编码可以检测出所有奇数错，可以检测出所有双位错，所有三位错，可以检测出大多数 4 位错（四个出错位正好位于矩形四个顶点除外）。

1.4.3 实验内容

1)　设计 16 位数据编码的偶校验编码电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成偶校验编码电路。实验电路输入输出引脚定义如图 1.10 所示。输入：16 位原始数据；输出：17 位校验码（16 位数据位+1 位校验位），其中校验位存放在最高位，注意输入 16 位原始数据的每一位都已经通过分线器利用隧道标签引出，可以直接复制到绘图区使用。

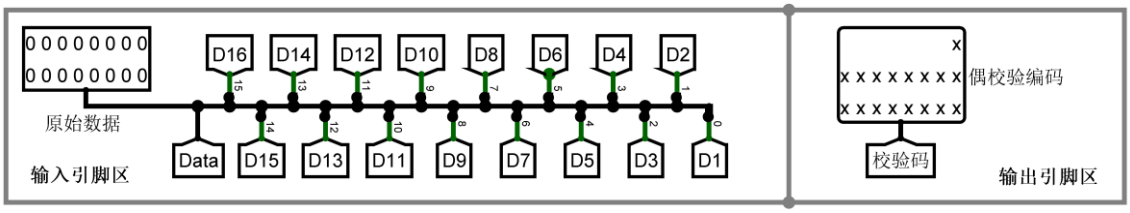


图 1.10 偶校验编码电路引脚定义

2)　设计 17 位偶校验编码的检错电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成偶校验检错电路。输入：17 位校验码，校验位存放在最高位；输出：16 位原始数据，1 位检错位；实验电路

输入输出引脚定义如图 1.11 所示，注意 17 位校验码的每一位都通过分线器利用隧道标签引出，方便实验时使用。

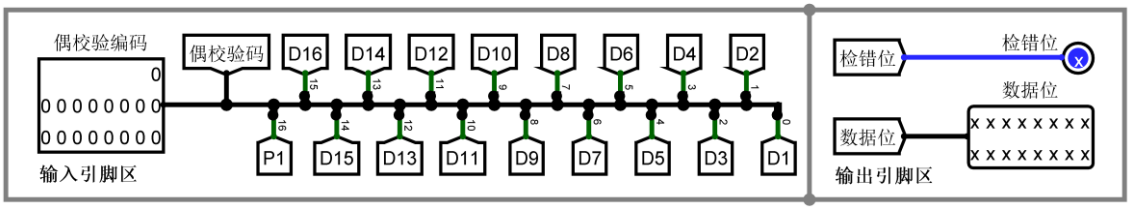


图 1.11 偶校验检错电路引脚定义

快速解题思路：新建子电路，利用 Logisim 中组合逻辑分析的功能输入电路逻辑表达式，自动生成电路，然后复制子电路到本电路中进行简单连接。

3) 偶校验传输测试

在偶校验传输测试电路中测试偶校验编解码电路功能实现是否正确，并观察数据传输过程中何时会出现误报情况，分析奇偶校验传输的性能。

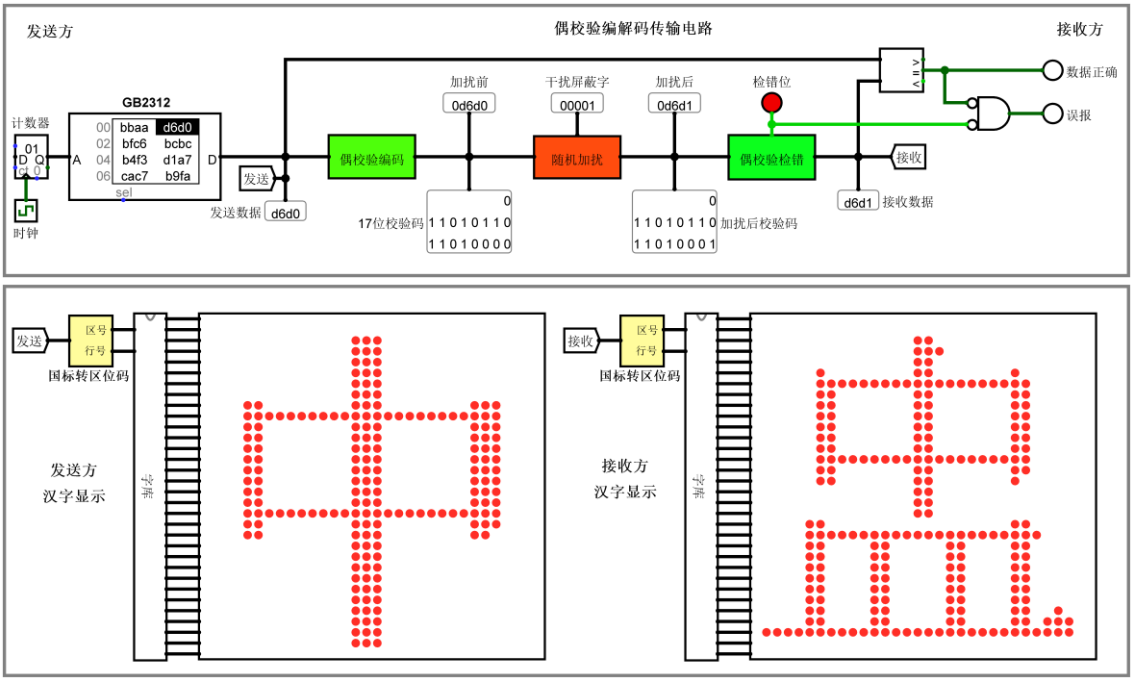


图 1.12 偶校验传输测试电路

偶校验传输测试电路如图 1.12 所示，左侧发送方通过地址计数器不断从 ROM 存储器中取出 16 位的汉字 GB2312 编码，每来一个时钟，地址计数器自动加一，通过 ROM 组件取出新的汉字编码，送入偶校验编码电路生成 17 位的校验码，再经过随机加扰模块进行随机加扰（有可能出现 1~3 位错），随机加扰后的校验码再通过偶校验检错电路进行检错判断，并将 16 位原始数据位提取出来，图中右上角部分电路用于检测偶校验检错位是否存在误报情况。该电

路下方引入了汉字显示模块，可以直接显示发送端和接收端的汉字，通过汉字显示是否一致可以很直观的观察传输是否发生错误，从而观察采用偶校验进行数据传输时传输的可靠性，用户可以使用 ctrl+t 快捷键开启时钟单步仿真测试。

1.4.4 实验思考

(1) 完成电路后，仔细观察传输过程，是否有发生错误但两边显示的汉字内容一样的情况，为什么会出现这种情况？

(2) 如果发生偶数个错误，检错位的值是多少？奇偶校验是否具有纠错的功能？

(3) Windows 系统中可以利用多个磁盘构建磁盘阵列，以提升存储容量和性能，但多个磁盘构建的存储系统会带来可靠性的降低，通常采用引入一个奇偶校验盘的形式解决，当损坏一块硬盘时，系统仍然可以工作，此时为什么奇偶校验可以工作？

(4) 内存条也可以采用奇偶校验，发现错误如何处理？

1.5 海明校验码设计实验

1.5.1 实验目的

学生掌握海明码设计原理与检错纠错性能，能独立设计实现汉字 GB2312 编码的海明校验编码体系，并最终在实验环境中利用硬件电路实现对应的编解码电路。

1.5.2 背景知识

1950 年理查德·海明(Richard Hanmming)提出了海明校验码，它本质上也是一种多重奇偶校验编码。它不仅具有检测错误的能力，还具有指出错误准确位置的能力。本节下面所讨论的海明编码均以指出并纠正一位错误的海明编码为例。

校验位位数：设海明校验码 $H_1H_2H_3$ …H_N 共 N 位，其中原始数据信息为 $D_1D_2D_3$ …D_k 共 k 位，校验位 $P_1P_2P_3$ …P_r 共 r 位，分成 r 组作奇偶校验，这样能产生 r 位检错信息。这 r 位信息就构成一个指错字 $G_r...G_2G_1$，可指出 2^r 种状态，其中全零状态表示无错误发生，其它所有状态均表示海明编码出错位。要满足以上海明编码规则，则应满足如下关系式：

$$\mathbf{N = k + r \le 2^r - 1}$$

表 1.6　有效信息位与校验位位数的关系

k	1	2～4	5～11	12～26	27～57	58～120	…
r	2	3	4	5	6	7	…

根据上述关系式，可以算出不同长度有效信息编成海明码所需要的最少校验位数。信息位

与检验位的对应关系如表 1.6 所示。

分组规则：原始数据 $D_1D_2D_3$ …D_k，校验位 $P_1P_2P_3$ …P_r 如何映射到海明码 $H_1H_2H_3$ …H_N 中，才能满足海明码利用检错码的值给出一位错位置的要求？下面我们看一下表 1.7，海明码 $H_1H_2H_3$ …H_N 各位均有对应的位置码，见表中第二行，假定传送过程中只有一位错发生，如果是 H_1 出错时，则检错码 G_r...G_2G_1=0001，检错码中只有 G_1=1，表明 G1 组有一位出错，如果数据位出错应该引起多个校验组的检错位出错，所以这里应该是校验位出错，H_1 位置应该放置 P_1。同理 H_2、H_4、H_8、H_{16} 应该分别存放 P_2、P_3、P_4、P_5，也就是所有校验位都应该存放在幂次方位上。

假设 H_3 出错，检错码 G_r...G_2G_1 应等于位置码 0011，则表明 G_1、G_2 校验组出错，H_3 编码应参与了 G_1、G_2 两个校验组，由于校验位只能参加一个校验组，所以这里 H_3 应该存放一个数据位，将 D_1 存放在 H_3 的位置。依次类推，可以根据海明码各位位置码的值将其数据位参与校验组的信息在表中逐一明晰，并逐一将数据位安排在对应的海明编码中，由此可以生成任意长度海明码的分组规则。表中 H_{11}～H_{16} 分组信息未完成，请补齐相关信息。

表 1.7 海明编码分组规则

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}	H_{14}	H_{15}	…
位置码	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	…
对应位	P_1	P_2	D_1	P_3	D_2	D_3	D_4	P_4	D_5	D_6	D_7					…
G_1 校验组	√		√		√		√		√		√					…
G_2 校验组		√	√			√	√			√	√					…
G_3 校验组				√	√	√	√									…
G_4 校验组								√	√	√	√					…
G_5 校验组																…

实际设计海明编码时，可以根据数据位长度直接截短或扩展该表格，根据表格得到海明码分组信息后，校验位以及检错码的值可以利用偶校验公式直接得到。检错码 G_r...G_2G_1 为全零时表示编码正确，非零时如果假设是一位错，检错码的值可以表示出错数据位的位置，将对应位置的数据取反即可纠正编码错误。

以上编码只有在假定一位错时才能进行纠错，当出现两位错时，假设 H_3、H_5 同时发生错误，由于 H_3 参与了 G_1、G_2 组，H_5 参与了 G_1、G_3 组，G_1 组发生了两位错，无法检错，而 G_2、G_3 组均发生了 1 位错，故对应的检错码应该是 0110，和 H_6 出错的检错码重叠，此时无法区分一位错和两位错，更无法纠错。为解决这个问题，通常可以为整个海明码引入一个总偶校验位，在海明码检错码报错时，如果总检错码为零，表示出现两位错，如果总检错码为 1，表示一位错，当然这种方法也仅仅是假定信道可靠，无三位以上错误的情况。

1.5.3 实验内容

1) 设计海明校验编码电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成海明校验编码电路。

输入输出引脚定义如图 1.13 所示。输入：16 位原始数据；输出：22 位校验码（16 位数据位+5 位校验位+1 位总校验位），注意输入 16 位原始数据的每一位都已经通过分线器利用隧道标签引出，可以直接复制到绘图区使用。

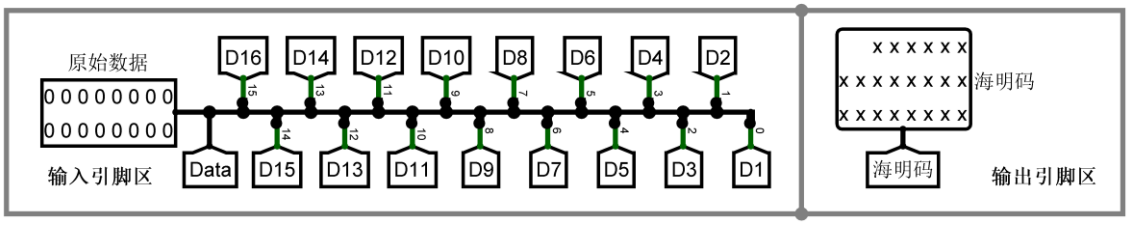


图 1.13 海明编码电路引脚定义

实验步骤：

1. 设计（16,5）的海明编码，根据海明编码的定义推导出该编码奇偶校验分组规则，设计编码中校验位数据位的放置顺序，给出校验码的逻辑表达式。
2. 在 logisim 中实现该电路，（尽量用子电路生成的方式绘制电路，可大量使用隧道标签简化线路连接）
3. 为上述电路增加所有 21 位的总校验位，构成最终的海明码。请思考为什么要引入这个总校验位？

2) 设计海明校验解码电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成海明校验解码电路。输入输出引脚定义如图 1.14 所示，输入：22 位校验码；输出：16 位原始数据，1 位检错位；2 位检错位；无错误状态位。注意输入 16 位原始数据的每一位都已经通过分线器利用隧道标签引出，可以直接复制到绘图区使用。

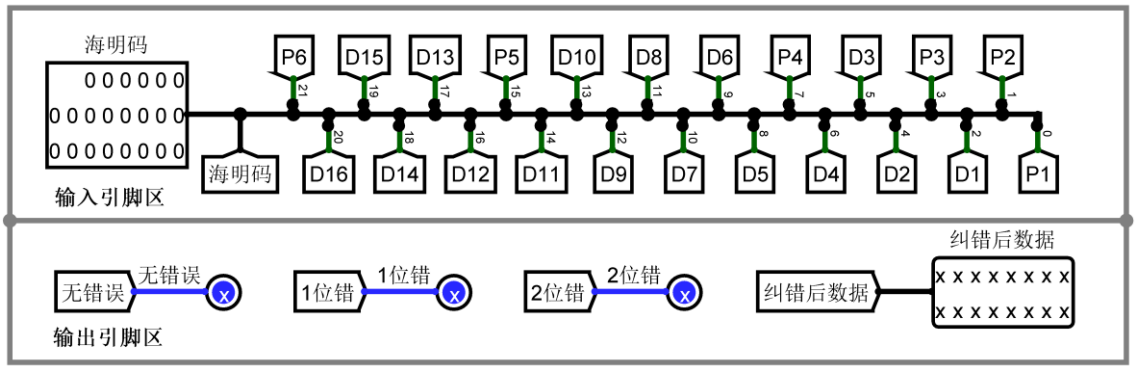


图 1.14 海明解码电路引脚定义

3) 海明校验传输测试 1

在海明校验传输测试 1 电路中测试海明校验编解码电路的正确性，注意图中随机干扰电路只能产生最多 2 位错误，具体测试电路如图 1.15 所示：

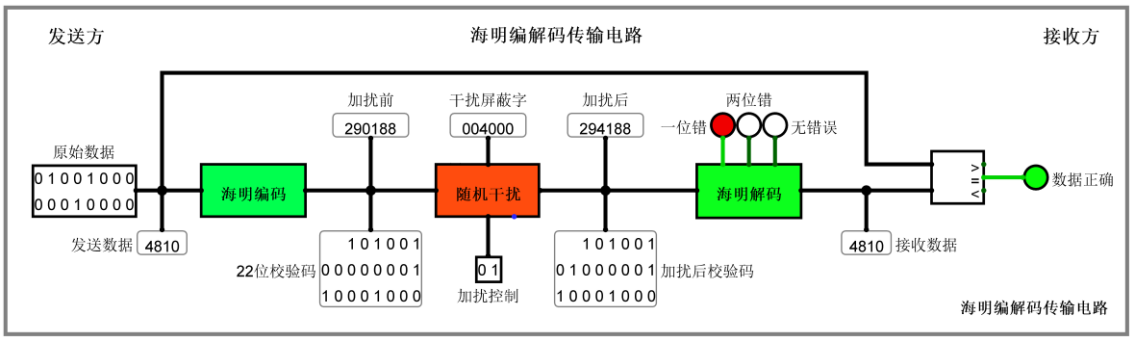
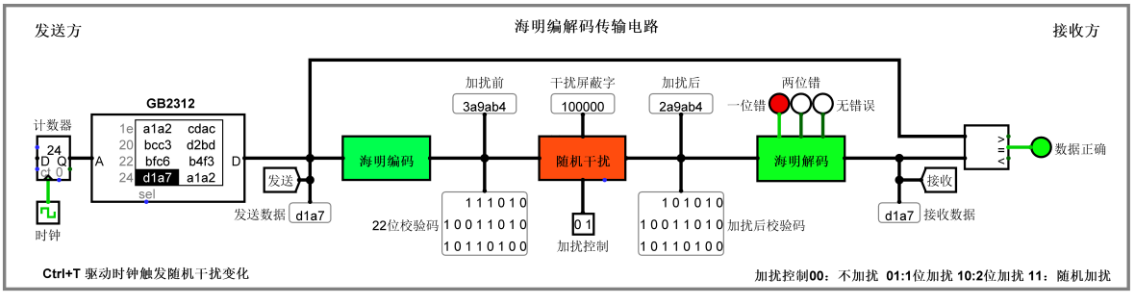


图 1.15 海明编码传输电路 1

可以通过修改加扰控制引脚的值来控制随机干扰模式，其中 00 表示无干扰，01 表示 1 位错随机干扰，10 表示两位错随机干扰，11 表示 0~2 位错随机干扰。

4) 海明校验传输测试 2

在海明校验传输测试 2 电路中测试海明校验编解码电路的正确性，该电路引入了汉字显示模块，可以直接显示发送端和接收端的汉字字形码，通过汉字显示可以很直观的看出海明纠错的效果，用户可以使用 CTRL+K 快捷键开启时钟自动仿真测试（请将 logisim 仿真频率调整到 8Hz），具体测试电路如图 1.16 所示：



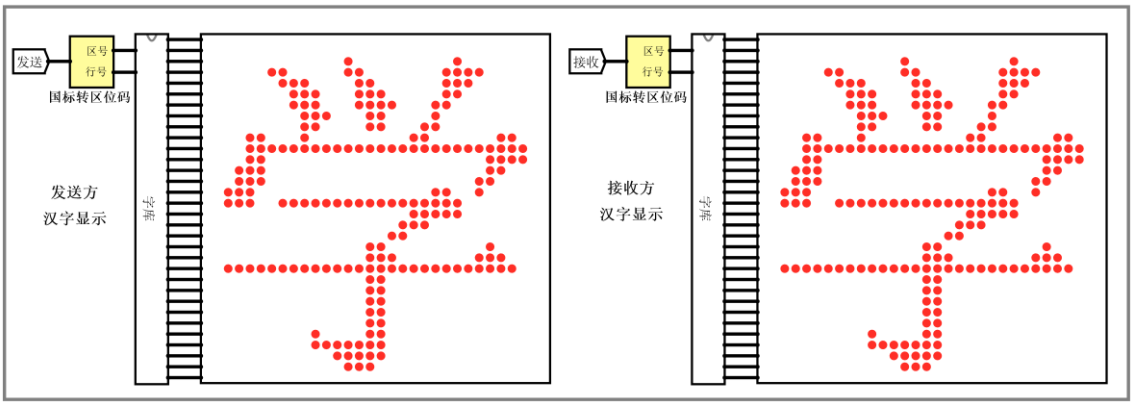


图 1.16 海明编码传输电路 2

1.5.4 实验思考

(1) 如果总校验位出错，其它 21 位编码出现一位错时，你是如何处理的？此时能否直接纠错？

(2) 如果出现三位错误，海明编码传输电路会出现什么情况？

(3) 海明编码的编码效率如何？

1.6 CRC 校验码设计实验

1.6.1 实验目的

学生掌握 CRC 循环冗余校验码的基本原理，能看懂串行 CRC 编解码电路，并利用所学数字逻辑知识设计实现 GB2312 编码 16 位数据的并行 CRC 编解码电路。

1.6.2 背景知识

循环冗余校验 CRC（Cyclic Redundancy Check）是一种基于模 2 运算建立编码规则的校验码，在磁存储和计算机通信方面应用广泛。

1) 编码规则

设 CRC 编码长度共 n 位，其中原始数据信息为 $C_{k-1}C_{k-2}\ldots C_1C_0$ 共 k 位，校验位共 r 位，称为（n,k）码，同样应满足关系式：

$$\mathbf{n = k+r \le 2^r-1}$$

(1) 对于一个给定的（n，k）码，假设待发送的 k 位二进制数据用信息多项式 M(x)表示。

$$M(x)=C_{k-1}x^{k-1}+C_{k-2}x^{k-2}+\ldots+C_ix^i+\ldots+C_1x+C_0$$

(2) 将 M(x)左移 r 位，可表示成 $M(x)\cdot 2^r$，右侧空出的 r 位用来放置校验位。

(3) 选择一个生成多项式 G(x)，其最高次幂等于校验位的位数 r，最低次幂等于 0，即 G(x)必须是 r+1 位。

(4) 用 $M(x)\cdot 2^r$ 按模 2 的运算规则除以生成多项式 G(x)所得的余数 R(x)作为校验码。设商为 Q(x)，余数为 R(x)，将余数 R(x)放置到 $M(x)\cdot 2^r$ 中右侧空出的 r 位上，就形成了 CRC 校验码。其多项式为：

$$M(x)\cdot 2^r+R(x)=[Q(x)\cdot G(x)+R(x)]+R(x)$$
$$=[Q(x)\cdot G(x)]+[R(x)+R(x)]$$

按模 2 的运算规则，$R(x)+R(x)=0$, 所以

$$M(x)\cdot x^r+R(x)=Q(x)\cdot G(x)$$

上面的公式表明，CRC 码一定能被生成多项式 G(x)整除。

2) CRC 编解码流程

图 1.17 中发送部件将原始数据信息 $a_k\ldots a_2a_1$ 左移 r 位后送入 CRC 编码电路，按模 2 的运

算除以 r+1 位的生成多项式 $g_r \ldots g_1 g_0$，将得到 r 位余数 $b_r \ldots b_1$ 与原始数据 $a_k \ldots a_2 a_1$ 拼接成 CRC 校验码，再经过不可靠链路传输给接收方，接收方将接受到的可能出错的 CRC 编码 $c_k \ldots c_2 c_1 d_r \ldots d_1$ 传送至 CRC 解码电路，同样按模 2 的运算除以 r+1 位的生成多项式 $g_r \ldots g_1 g_0$,将得到 r 位余数 $s_r \ldots s_1$ 送决策逻辑，决策逻辑根据余数值判断是否有错，若余数为零，表明传输未发生错误（注意：也有一定概率出错），接收该数据，若余数不为 0，表示出错，再由决策逻辑根据余数的值决定是否纠错或者直接丢弃该数据要求发送方重传。

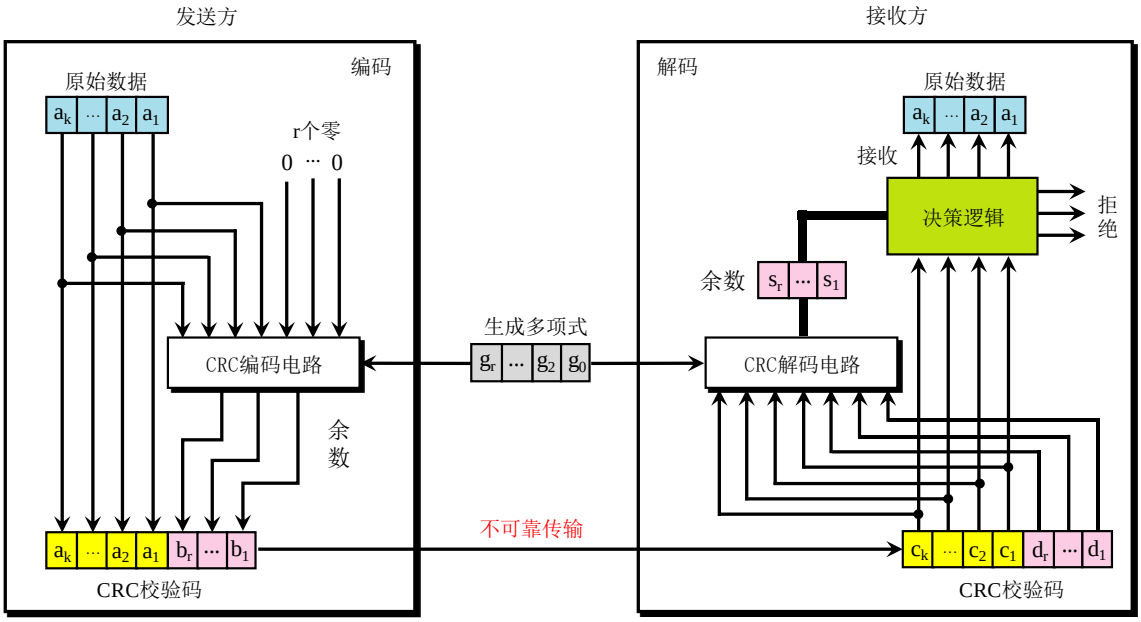


图 1.17 CRC 编码传输

3) CRC 编码特性

CRC 编码不为零的余数的具有循环特性。即将余数左移一位除以生成多项式，将得到下一个余数，继续在新余数基础上左移一位除以生成多项式，余数最后能循环到最开始的余数。以（7,3）码为例，当生成多项式为 11101 时，不同出错位的余数将按表 1.8 列出的规律变化，表中第四行余数为 0100，表明第五位出错，左移一位继续除 11101，余数将为 1000，1000 左移一位继续除 11101，余数为 0111，持续左移做除法，余数将回滚到 0100，这就是循环冗余校验码名称的来由。

表 1.8 （7,3）码的出错模式（G(x)＝11101）

#	X_1	X_2	X_3	X_4	X_5	X_6	X_7	余数	余数值	出错位
1	1	1	0	1	0	0	1	0 0 0 0	0	无
2	1	1	0	1	0	0	**0**	0 0 0 1	1	7
3	1	1	0	1	0	**1**	1	0 0 1 0	2	6
4	1	1	0	1	**1**	0	1	**0 1 0 0**	**4**	5
5	1	1	0	**0**	0	0	1	1 0 0 0	8	4

#	X_1	X_2	X_3	X_4	X_5	X_6	X_7	余数	余数值	出错位
6	1	1	1	1	0	0	1	1 1 0 1	13	3
7	1	0	0	1	0	0	1	0 1 1 1	7	2
8	0	1	0	1	0	0	1	1 1 1 0	14	1
9	1	1	0	1	0	1	0	**0 0 1 1**	3	6+7
10	1	1	0	1	1	1	1	0 1 1 0	6	5+6
11	1	1	0	0	1	0	1	1 1 0 0	12	4+5
12	1	1	1	0	0	0	1	0 1 0 1	5	3+4
13	1	0	1	1	0	0	1	1 0 1 0	10	2+3
14	0	0	0	1	0	0	1	1 0 0 1	**9**	1+2
15	1	1	1	1	1	0	1	1 0 0 1	**9**	3+5
16	0	0	1	1	0	0	1	0 1 0 0	**4**	1+2+3

上表中所有一位错的余数均不同，且具有可循环的特性，另外所有双位错的余数与一位错的余数均不同，注意由于模 2 运算的性质，所有双位错运算的余数等于对应出错位余数按模 2 相加的结果，比如第 9 行第 6、7 位错的余数=0001+0010=0011，据此规律，将所有一位错的余数值 1、2、4、8、13、7、14 两两组合得到所有双位错的情况，见表 1.9。从表中可以发现所有双位错余数均不为零，但存在重复情况，且与一位错余数均不同，由此说明（7,3）编码可以区分一位错和两位错。另外表 1.8 中第 16 行给出了 1～3 位全错的余数，和第 4 行中第 5 位错的余数相同，所以对于（7,3）编码，无法区分一位错和三位错，可以区分一位错还是两位错，假定没有 3 位错的前提下，是可以通过余数纠正一位错的，当然（7,3）编码的编码效率是较低的，实际使用的 CRC 编码大多是无法区分一位错和两位错的。

表 1.9 （7,3）CRC 码的双位错余数（生成多项式 G(x)＝11101）

出错位（余数）	7（1）	6（2）	5（4）	4（8）	3（13）	2（7）	1（14）
7（1）		3	5	9	12	6	15
6（2）			6	10	15	5	12
5（4）				12	9	3	10
4（8）					5	15	6
3（13）						10	3
2（7）							9
1（14）							

4) 生成多项式

生成多项式由发送方和接收方共同约定。在发送方利用生成多项式对信息多项式做模 2 除法生成校验码；接收方利用生成多项式对收到的编码做模 2 除法以检测和确定错误位置。对于 CRC 校验中的生成多项式有如下特殊要求：

(1)生成多项式的最高位和最低位必须为 1；

(2)当 CRC 校验码任何一位发生错误时，被生成多项式进行模 2 除后余数应不为 0；

(3)不同位发生的错误，余数不同；

(4)对余数继续做模 2 除，应使余数循环。常用生成多项式如表 1.10 所示。

表 1.10 常见生成多项式

CRC 编码	用途	多项表达式
CRC-1	奇偶校验	$x+1$
CRC-3	GSM 移动网络	x^3+x+1
CRC-4	ITU-T G.704	x^4+x+1
CRC-5-ITU	ITU-T G.704	$x^5+x^4+x^2+1$
CRC-5-EPC	二代 RFID	x^5+x^3+1
CRC-5-USB	USB 令牌包	x^5+x^2+1
CRC-6-GSM	GSM 移动网络	$x^6+x^5+x^3+x^2+x+1$
CRC-7	MMC/SD 卡	x^7+x^3+1
CRC-16-CCITT	USB、bluetooth	$x^{16}+x^{15}+x^2+1$
CRC-32	Ehernet，SATA MPEG-2,PKZIP,Gzip…	$x^{32}+x^{26}++x^{23}+x^{22}+x^{16}+x^{12}$ $+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

5) CRC 检错性能

在数据通信与网络中，通常 k 相当大，由一千甚至数千数据位构成一帧，采用 CRC 编码产生 r 位的校验位，具有如下检错能力：

1. 所有突发长度小于等于 r 的突发错；
2. $(1-2^{-(r-1)})$比例的突发长度为 r+1 的突发错；
3. $(1-2^{-(r)})$比例的突发长度大于 r+1 的突发错；
4. 小于最小码距的任意位数的错误；
5. 如果生成多项式中 1 的个数为偶数，可以检测出所有奇数错误。

这里突发错误是指几乎是连续发生的一串错，突发长度就是指从出错的第一位到出错的最后一位的长度(但是，中间并不一定每一位都错)。如 r=16，就能检测出所有突发长度小于等于 16 的突发错，以及 99.997%的突发长度为 17 的突发错和 99.998%的突发长度大于 17 的突发错。所以 CRC 码的检错能力还是很强的。

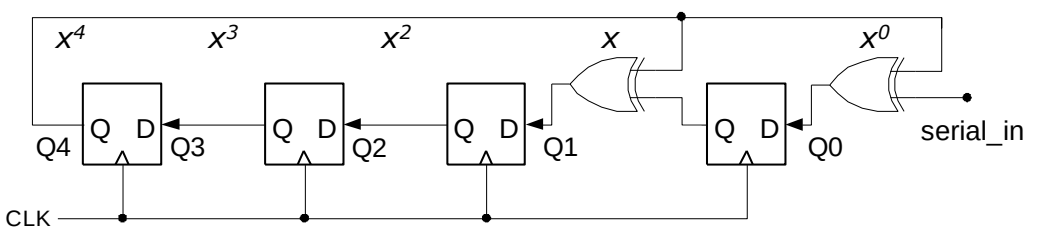


图 1.18 CRC 串行编码电路

6) CRC 编解码电路

图 1.18 是 CRC-4 采用的串行编码电路，待编码的 n 位数据从右侧 serial_in 端输入，经过 n-1 个时钟周期后可以计算出最终的余数 $Q_3Q_2Q_1Q_0$，D 触发器初始状态均为 0，图中所有异或门与 Q4 进行异或，最开始 Q4=0，所有异或门异或零相当于数据直通，整体电路变成一个同

步左移电路，与模 2 运算中的首位为零，不够减，直接左移相同；当串行输入中第一个为 1 的数字传输到 Q4 时，此时所有异或门异或上 1，这个操作就是模 2 除法中首位为 1，商上 1，够除，被除数与除数进行模 2 的减法---异或操作，这里有异或门的位置相当于生成多项式对应位为 1 的位置，无异或门的位置，相当于生成多项式对应位置为零的位置，注意图中 x 幂次方的标记，首位运算结果一定是 0，不存在异或门，所以图中的生成多项式为 10011。

串行 CRC 编码电路结构简单，但时间复杂度较高，需要 n-1 个时钟周期才能完成 n 位数据的 CRC 编码运算，在高速通讯领域应用中，串行编码结构无法胜任，现在普遍采用快速的并行 CRC 编码电路。

1.6.3 实验内容

1) CRC 串行编解码 电路设计

图 1.19 是在 Logisim 中实现的 CRC(7,3)编码的串行编码电路，可以直接在左上角移位寄存器中利用戳工具输入 3 位数据，然后驱动时钟进行自动仿真，最终的校验码会在 LED 中显示，尝试验证表 1.8 中各项的正确性。

将该电路适当修改，使其功能变成 CRC(7,3)编码的串行解码电路，尝试给出一种该电路无法检错的例子，分析该编码的检错性能。

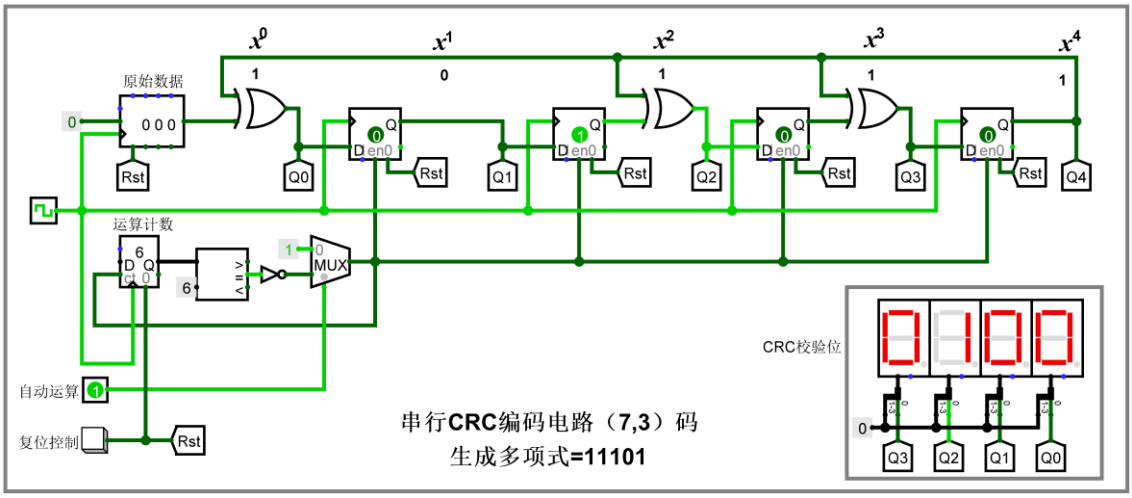


图 1.19 串行 CRC（7，3）编码电路

2) CRC 并行编解码电路设计

在 logisim 中打开实验资料包中的 data.circ 文件，尝试利用纯组合逻辑电路实现 CRC 并行编解码电路，输入输出引脚定义如图 1.13 所示。输入：16 位原始数据；输出：22 位校验码（16 位数据位+5 位 CRC 校验位+1 位偶校验位），注意输入 16 位原始数据的每一位都已经通过分线器利用隧道标签引出，可以直接复制到绘图区使用。

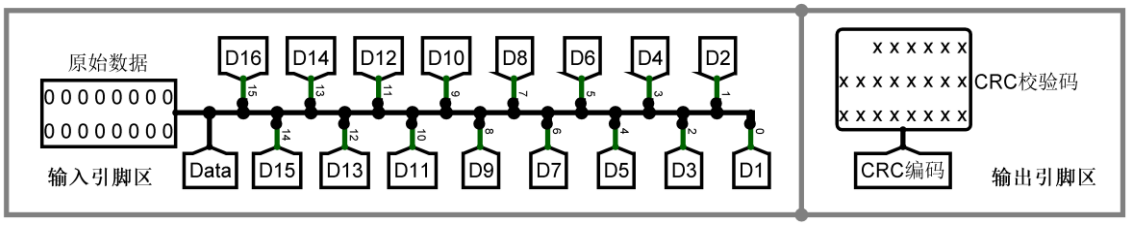


图 1.20 CRC 编码电路引脚定义

1. 选择合适的生成多项式，构建 CRC-5（21,16）编码。
2. 设计并行 CRC 编码电路。（提示：两数据按模 2 加的和的校验码等于两数据校验码按模 2 加的和）
3. 假设数据只能发生两位错和一位错，该编码能否区分一位错和两位错，在最终实现的电路中检验这个问题，并思考为什么？
4. 为上述电路增加所有 21 位的总校验位，构成最终的校验码？请思考为什么要引入这个总校验位？

3) 设计 CRC 校验解码电路

在 logisim 中打开实验资料包中的 data.circ 文件，在对应电路中完成 CRC 校验解码电路。输入输出引脚定义如图 1.14 所示：输入：22 位校验码（16 位数据位+5 位 CRC 校验位+1 位偶校验位）；输出：16 位原始数据，1 位检错位；2 位检错位；无错误状态位，发生 1 位错时应该能纠正错误。注意输入 16 位原始数据的每一位都已经通过分线器利用隧道标签引出，可以直接复制到绘图区使用。

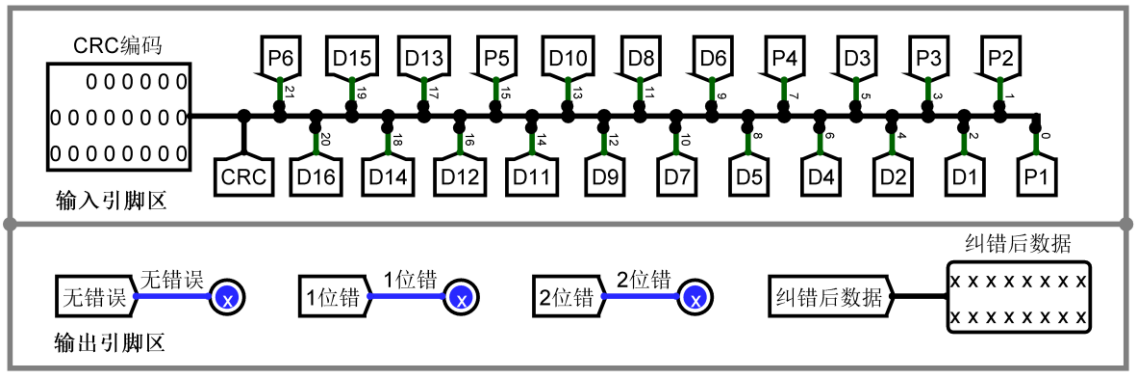


图 1.21 CRC 解码电路引脚定义

4) CRC 校验码传输测试 1

在 CRC 校验传输测试 1 电路中测试 CRC 校验编解码电路的正确性，注意图中随机干扰电路只能产生最多 2 位错误，具体测试电路如图 1.15 所示：

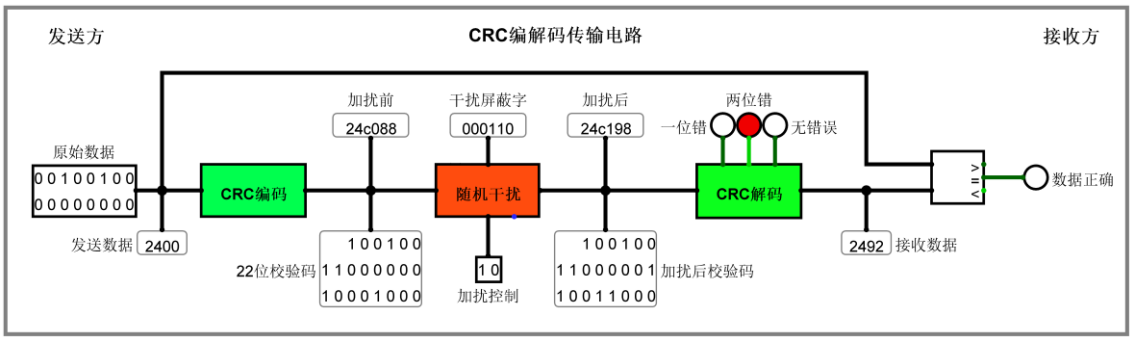
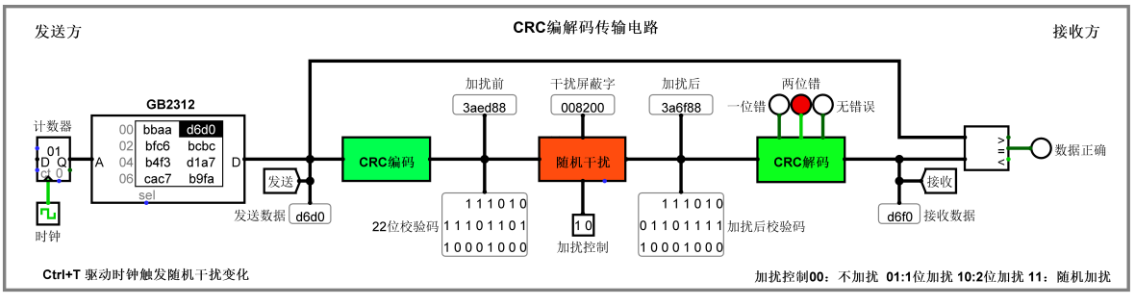


图 1.22 CRC 编码传输电路 1

可以通过修改加扰控制引脚的值来控制随机干扰模式，其中 00 表示无干扰，01 表示 1 位错随机干扰，10 表示两位错随机干扰，11 表示 0-2 位错随机干扰。

5) CRC 校验码传输测试 2

在 CRC 校验传输测试 2 电路中测试海明校验编解码电路的正确性，该电路引入了汉字显示模块，可以直接显示发送端和接收端的汉字，通过汉字显示可以很直观的看出海明纠错的效果，用户可以使用 CTRL+K 快捷键开启时钟自动仿真测试（请将 logisim 仿真频率调整到 8Hz），具体测试电路如图 1.23 所示：



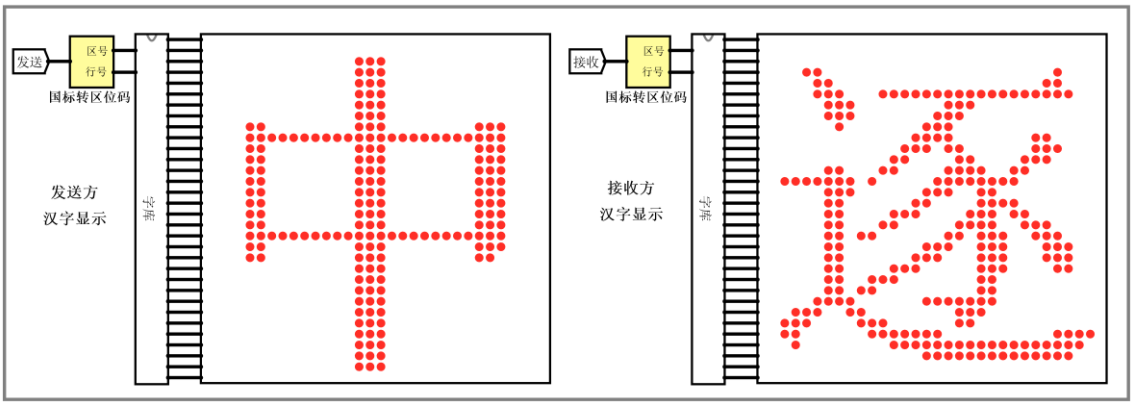


图 1.23 CRC 编码传输电路 2

1.6.4 实验思考

(1) CRC 编码的编码效率如何？

(2) 如果总校验位出错，其它 21 位编码出现一位错时，你是如何处理的？此时能否直接纠错？

(3) 如果出现三位错误，编码传输电路会出现什么情况？

(4) CRC-6 编码为什么不能像（7,3）码那样能区分一位错和两位错？

1.7 编码流水传输实验

1.7.1 实验目的

学生熟悉流水数据传输机制，流水暂停原理，为最终的流水 CPU 设计做好技术储备，最终学生能对实验环境提供的五段流水编码传输电路进行简单修改，实现数据编码在不可靠网络中的可靠传输。

1.7.2 背景知识

计算机中的流水线技术将一个复杂的任务分解为若干个阶段，每个阶段与其它阶段并行运行，其运行方式和工业流水线十分相似，因此被称为流水线技术。计算机中常见的流水机制有指令流水线，流水乘法器，流水浮点运算器等。具体结构如图 1.24 所示，复杂任务被分成若干个阶段，每个阶段利用组合逻辑对输入数据进行加工，组合逻辑加工的结果用锁存器锁存。

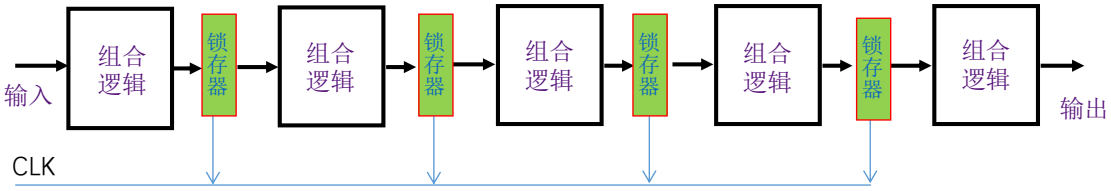


图 1.24 流水线结构

所有锁存器均采用公共时钟同步，每来一个时钟，各段组合逻辑功能部件处理完成的数据会锁存到锁存器中，作为下段的输入。由于流水各阶段逻辑延迟时间可能并不一致，为保证流水线正确运行，最大时钟频率取决于流水各阶段中最慢的阶段，所以分段时应该尽量让各阶段时间延迟相等，采用流水技术可以大大提升系统效率，假设各段时间延迟均为 T，当流水充满后每隔一个时钟周期 T 就会完成一个数据加工，如果不采用流水技术，上图需要 5T 才能完成一个数据的加工。锁存器在公共时钟的驱动下可以锁存流水线前段加工完成的数据以及控制信号，锁存的数据和信号将用于后段的继续加工或处理。

1.7.3 实验内容

图 1.25 中将海明编码传输过程分成了 5 个阶段（取数，编码，传输，解码，显示）类似 CPU 指令流水线的处理过程，注意此实验必须在完成海明校验实验或 CRC 校验实验后方能进行，CRC 编码传输也有相同的测试电路。中间蓝色长条为流水接口部件（内部实际是若干锁存器/寄存器，用于锁存数据和控制信号），流水接口部件提供同步清零控制信号，试启用时钟自动仿真运行该电路（CTRL+T），观察接收方接受到的信息，当发生两位错时，将会发生错误。

尝试使用最少的器件简单修改该电路，使得解码阶段出现两位错时，系统能自动重传出错的编码（类似指令流水线中的分支跳转），从而使得该电路能正确传输所有数据。

解题思路：当出现两位错时，首先要将编码阶段和传输阶段的数据清空，这部分数据会导致接收端数据顺序不一致，同时要将已经进入显示阶段的数据锁定，并将取数阶段的地址回滚到正确的位置。

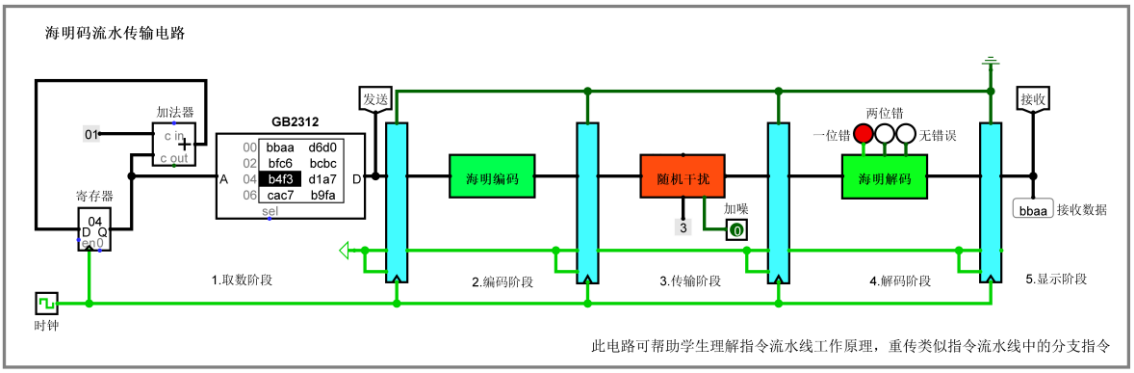


图 1.25 海明编码流水传输电路

1.7.4 实验思考

(1) 将随机加扰电路加噪控制端置 1，可能会引起 3 位以上的错误，此时流水传输是否还能正常完成？如不能如何处理？

(2) 思考一下，流水接口部件的清空信号是时钟同步清空还是电平异步清空好，为什么？