CODEC Hamming

Alberto Molina Martínez
José Ángel de Pascual Viciana

27 de febrero de 2002

Índice general

1. El proyecto **3**
1.1. Justificación de proyecto 3
1.2. Un planteamiento general 5
1.3. Módulo codificador 6
1.4. Módulo decodificador 7
1.5. Relojes secundarios 8
1.6. Protocolos de transmisión 11
1.6.1. Protocolo de envío 11
1.6.2. Protocolo de recepción 11
1.7. Código fuente de la implementación 11
1.7.1. Contador de 0 hasta 6 11
1.7.2. Contador de 0 hasta 27 14
1.7.3. Contador de 2 bits 17
1.7.4. Reloj múltiple 19
1.7.5. Retardo 21
1.7.6. Codificador Hamming 23
1.7.7. Decodificador Hamming 28
1.7.8. Protocolo de envío 33
1.7.9. Protocolo de recepción 36
1.7.10. Módulo emisor 40
1.7.11. Módulo de recepción 45
1.7.12. Entrenador general 48

A. Teoría de códigos lineales **53**
A.1. Conceptos algebraicos fundamentales 53
A.1.1. Definiciones 53
A.1.2. Propiedades de GF(2) 54
A.2. Introducción a los códigos bloque lineales 55
A.2.1. Definición de código bloque lineal 56
A.2.2. Forma sistemática 57
A.2.3. Matriz de comprobación de paridad 59

A.3. Síndrome y detección de errores 59
A.4. Distancia mínima de un código 62
A.5. Propiedades de detección y corrección de errores de un código bloque . 63
A.5.1. Propiedades detectoras 63
A.5.2. Propiedades correctoras 64
A.6. Matriz típica y decodificación de síndrome 66
A.6.1. Partición como método de decodificación 66
A.6.2. Construcción de la matriz típica 66
A.7. Códigos Hamming 70

B. Diseño de subcircuitos **72**
B.1. Contador binario de 0 a 6 72
B.2. Contador hasta 27 73

Capítulo 1

El proyecto

1.1. Justificación de proyecto

En cualquier comunicación electrónica (por ejemplo, en una red de computadores) es conveniente saber si se han producido errores en la transmisión. Por desgracia hoy día es muy frecuente que la información enviada por un emisor se altere en el medio de transmisión, por efectos del ruido, con lo cual el receptor recibiría "algo" diferente a lo que se envió.

Esencialmente, hay tres métodos de corrección de errores:

Sustitución de símbolos. La *sustitución de símbolos* se diseñó para usarse en un ambiente humano: en donde hay un ser humano, en la terminal de recepción, para analizar los datos recibidos y tomar decisiones sobre su integridad. Con la sustitución de símbolos, si un carácter se recibe en error, en vez de revertirse a un nivel superior de corrección de errores o mostrar el carácter incorrecto, un carácter único que es indefinido por el código de caracteres, tal que como un signo de interrogación invertido, se sustituye por el carácter malo. Si el carácter erróneo no puede distinguirse por el operador, la retransmisión es para llamada (o sea, la sustitución de símbolos es una forma de retransmisión selectiva). Por ejemplo, si el mensaje "nombre" tenía un error en el primer carácter, se mostraría como "?ombre", un operador puede discernir el mensaje correcto por inspección, y la retransmisión no es necesaria. Sin embargo, si el mensaje "$?,000.00" se recibiera, un operador no podría determinar el carácter correcto y la retransmisión sería requerida.

Retransmisión. La *retransmisión*, como el nombre lo implica, es volver a enviar un mensaje, cuando es recibido un error, y la terminal de recepción automáticamente pide la retransmisión de todo el mensaje. La retransmisión frecuentemente se llama ARQ, el cual es un término antiguo de la comunicación de radio, que significa *petición automática para la retransmisión.* ARQ es probablemente

el método más confiable de corrección de errores, aunque no siempre es el más eficiente. Las dificultades en el medio de transmisión ocurren en ráfagas. Si se usan mensajes cortos, la probabilidad de que una dificultad ocurra, durante la transmisión, es pequeña. Sin embargo, los mensajes cortos requieren de más reconocimientos y regresos de línea que los mensajes largos. Los reconocimientos y regresos de línea para el control de error son formas de *encabezamientos* (caracteres diferentes a los datos que se deben transmitir). Con los mensajes largos, menos tiempo de regreso es necesario, aunque la probabilidad de que un error de transmisión ocurra es mayor que para los mensajes cortos. Se puede mostrar, de manera estadística, que los bloques de mensajes entre 256 y 512 caracteres son de tamaño óptimo, cuando se utiliza ARQ para corrección de errores.

Seguimiento de corrección de error. El *seguimiento de corrección de error* (FEC), es el único esquema de corrección de error que detecta y corrige los errores de transmisión, del lado receptor, sin pedir retransmisión.

Con FEC, se agregan bits al mensaje antes de la transmisión. Un código de corrección de errores popular, es el *código de Hamming*, desarrollado por R. W. Hamming, en los laboratorios Bell. Este será el procedimiento que el proyecto va estudiar, e incluso llegando a hacer un diseño lógico que nos permita simularlo. El desarrollo teórico del código Hamming se encuentra en el **apéndice A**.

El código de Hamming que tratamos, detectará sólo errores de un sólo bit, no se puede usar para identificar errores de bit múltiples.

El código de Hamming, como todos los códigos FEC, requiere de la adición de los bits a los datos, alargando consecuentemente el mensaje transmitido. El propósito de los códigos FEC es reducir o eliminar el tiempo gastado en retransmisiones. Sin embargo, la suma de los bits FEC a cada mensaje gasta el tiempo de transmisión. Obviamente, se negocia entre ARQ y FEC, y los requerimientos del sistema determinan qué método es mejor para un sistema en particular. El FEC frecuentemente se usa para transmisiones sencillas a muchos receptores, cuando los reconocimientos no son prácticos.

Existen muchos tipos de códigos Hamming en función del número de bits redundantes que se quieran añadir. Este número nos determinará la longitud de la palabra código completa, así como el número de bits de información del mensaje. En la sección teórica se habla con más detalle acerca de este concepto. En nuestro caso hemos escogido 3 bits redundantes, por lo que tenemos una palabra código de 7 bits (longitud 7), de los cuales, 4 son de información. A partir de ahora, denominaremos nuestro código como un código Hamming (7,4).

En las siguientes secciones se describirá, paso a paso, cómo ha sido el proceso de diseño de este sistema de corrección de errores.

1.2. Un planteamiento general

Antes de proceder al diseño de componentes es preciso alzar la vista y ver dónde se sitúa el sistema en el contexto de una comunicación electrónica–digital.

Hagamos una descomposición funcional del sistema y de cómo éste interacciona con las entidades externas.

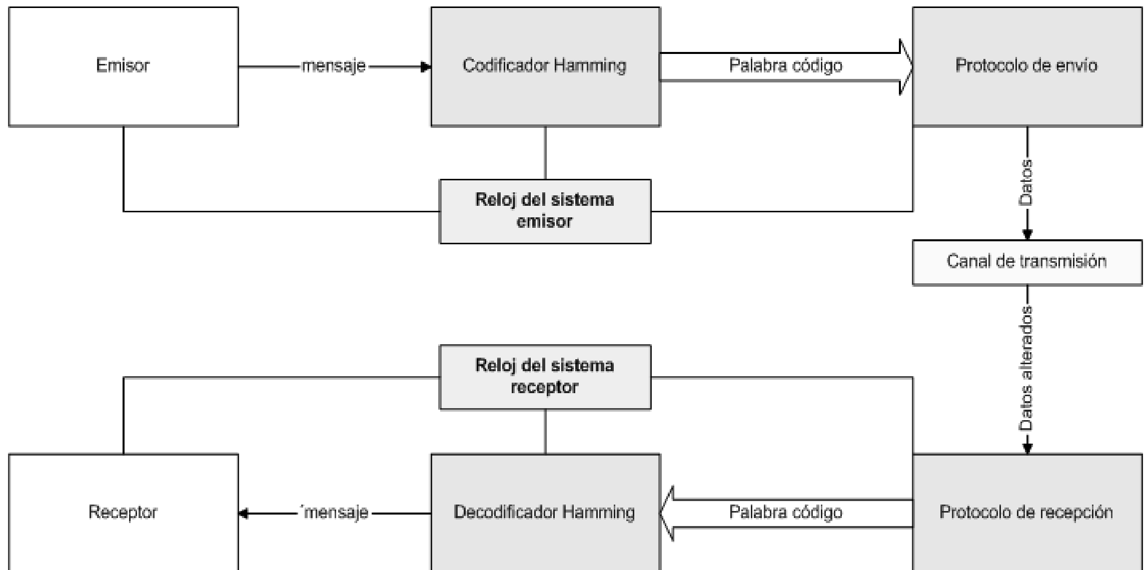


Figura 1.1: Diagrama de contexto de nuestro sistema

Como elementos externos o ajenos al sistema tenemos los siguientes:

- **Emisor:** Es la parte que envía información al receptor.
- **Receptor:** Es quien recibe la información enviada anteriormente.
- **Canal de transmisión**: Es el medio físico por el cual se transmiten los datos enviados por el emisor. Este canal, por lo general, suele alterar ocasionalmente los datos transmitidos.

El sistema que nosotros proponemos estaría compuesto por los siguientes módulos:

- **Codificador:** Es el subsistema encargado de codificar el mensaje enviado por el emisor a una palabra del código Hamming (7,4).
- **Protocolo de envío:** Este elemento recibe una palabra código de 7 bits en paralelo, y la envía al canal de transmisión en serie. En definitiva, lo que tenemos es un conversor paralelo–serie.

- **Protocolo de recepción:** Recibe los datos del canal de transmisión transformándolos de serie a paralelo. Además se encarga de la sincronización de relojes para que se lean los bits en el momento adecuado. Devuelve una palabra de 7 bits en paralelo al decodificador.
- **Decodificador:** Se encarga de decodificar una palabra código en un mensaje, corrigiendo cualquier error que se hubiera cometido.
- **Relojes:** Se trata de dos relojes con la misma frecuencia pero desfasados un tiempo t, igual al tiempo de transmisión por el canal.

1.3. Módulo codificador

El codificador recibe una palabra en serie de cuatro bits procedente del emisor. A continuación esta palabra se multiplica por la matriz generadora (G), resultando así una palabra de siete bits ya codificada. Este proceso se explica con más detalle en la sección de teoría del apéndice. Resumiendo, se realiza la siguiente operación:

$$(x_3 x_2 x_1 x_0) \times \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right) = (y_6 y_5 y_4 y_3 y_2 y_1 y_0)$$

El orden de llegada de los bits al módulo codificador debe ser del más significativo al menos significativo, es decir, primero llega x_3, luego llegaría x_2, X_1 y por último x_0.

Como es de suponer, la velocidad de transmisión de la palabra debe ajustarse a un reloj con una frecuencia preestablecida. Dicho de otro modo, por cada pulso de reloj se leerá la información contenida en el canal y será interpretada como un bit.

Entrando ya en el funcionamiento interno del propio módulo codificador, los bits de entrada van llegando, uno a uno, a un registro de desplazamiento de longitud 4. Por tanto, cada 4 ciclos de reloj el registro contendrá un mensaje válido apto para ser codificado.

En teoría, la codificación se hace mediante una multiplicación de matrices, esto puede simplificarse en un circuito combinacional ya que esta matriz generadora es siempre la misma para nuestro código Hamming. Si observamos esta matriz generadora vemos que las cuatro primeras columnas forman una matriz identidad 4×4, en consecuencia, los 4 primeros bits de la palabra codificada deberán coincidir con el mensaje del emisor. Las tres últimas columnas de G serán las que aporten los bits de redundancia que posibilitarán la futura corrección de errores.

Aprovechamos el hecho de que la multiplicación en base 2 se corresponde con una operación lógica AND, y la suma con la XOR. Entonces la multiplicación de una palabra por una matriz se resume:

$$(x_3x_2x_1x_0)\times\begin{pmatrix} d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,1} & d_{3,2} & d_{3,3} \\ d_{4,1} & d_{4,2} & d_{4,3} \end{pmatrix} = \begin{matrix} ((x_3 \otimes d_{1,1}) \oplus (x_2 \otimes d_{2,1}) \oplus (x_1 \otimes d_{3,1}) \oplus (x_0 \otimes d_{4,1}), \\ (x_3 \otimes d_{1,2}) \oplus (x_2 \otimes d_{2,2}) \oplus (x_1 \otimes d_{3,2}) \oplus (x_0 \otimes d_{4,2}), \\ (x_3 \otimes d_{1,3}) \oplus (x_2 \otimes d_{2,3}) \oplus (x_1 \otimes d_{3,3}) \oplus (x_0 \otimes d_{4,3})) \end{matrix}$$

Esto, aplicado a nuestro caso sería:

$$(x_3x_2x_1x_0)\times\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{matrix} ((x_3 \otimes 0) \oplus (x_2 \otimes 1) \oplus (x_1 \otimes 1) \oplus (x_0 \otimes 1), \\ (x_3 \otimes 1) \oplus (x_2 \otimes 0) \oplus (x_1 \otimes 1) \oplus (x_0 \otimes 1), \\ (x_3 \otimes 1) \oplus (x_2 \otimes 1) \oplus (x_1 \otimes 0) \oplus (x_0 \otimes 1)) \end{matrix}$$

La codificación que hemos visto se realiza constantemente incluso para estados intermedios del registro de desplazamiento. Hay que admitir sólo aquellos resultados correctos, para esto lo que se hace es añadir siete flip–flops D que sólo admiten el resultado al final del cuarto ciclo. Se consigue de este modo que la palabra código que haya en la salida del codificador siempre sea correcta, y además, persista durante todo el tiempo que va llegando la siguiente palabra. Esto último es muy importante porque si no no habría que enviar toda la palabra código de 7 bits en un ciclo y no en 4 como está pensado.

Estos son los puntos claves de este módulo, además hay detalles que no se han mencionado ya que solamente sirven para llevar el control de todo lo anterior.

En la lámina 1 se muestra el circuito correspondiente a este módulo.

1.4. Módulo decodificador

Este módulo es muy similar al codificador. En este caso se reciben 7 bits en serie correspondientes a la palabra código a decodificar. Una vez más, en la sección de teoría se explica cómo decodificar una palabra código usando la matriz de paridad (H) y síndromes[1].

En este módulo también existe un registro de desplazamiento para gestionar la recepción de la serie de bits. Además está la lógica encargada de la multiplicación de dicha palabra por H para el cálculo del síndrome (lógica similar a la empleada en el módulo codificador).

Cuando el síndrome calculado es (001), (010), (011) ó (100) se debe de sumar un 1 a la posición indicada por el síndrome (esta posición es el valor decimal del síndrome). De esta forma queda corregido un posible error (uno y sólo uno). Si el síndrome es cualquier otro valor la palabra decodificada serán los primeros 4 bits sin ningún cambio.

En la lámina 3 está el esquema lógico correspondiente al módulo decodificador.

[1]Este término se usa en el proceso de decodificación y es explicado en la parte de teoría.

1.5. Relojes secundarios

A lo largo del transcurso de codificación y decodificación existen situaciones en las que es preciso, por ejemplo, leer cuatro bits a la vez que se envían 7. Es decir, deben existir dos relojes que, en el mismo tiempo, uno haga cuatro pulsos y el otro siete, todo esto sin ninguna posibilidad de desfase.

En principio no parece fácil ya que 7 no es múltiplo de 4. En una simulación por computador sería posible crear dos relojes independientes de forma que uno sea 1,75 veces más rápido que el otro, pero en la realidad esto es inviable. Sin embargo, lo que sí se puede hacer es tener un reloj principal y los dos relojes mencionados dependiendo de éste. El mínimo común múltiplo de 7 y 4 es 28, por tanto, cada 28 ciclos del reloj principal los relojes secundarios deben coincidir. Además, en estos 28 ciclos uno de los relojes debe hacer 4 pulsos y el otro 7, o lo que es lo mismo, cada 4 pulsos del reloj principal habrá un pulso para el reloj de 7 y cada 7 pulsos habrá un ciclo para el reloj de 4.

A partir de ahora, llamaremos *reloj7* al reloj secundario rápido y *reloj4* al lento.

Necesitamos un contador asociado al reloj principal que cuente desde 0 hasta 27, diseñado en el apéndice B.2.

Esta sería la tabla de verdad para el reloj7 en función de los valores del contador anterior:

Q_4	Q_3	Q_2	Q_1	Q_0	$reloj7$
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

La función lógica del reloj7 sería:

$$reloj7 = \overline{Q_4} \cdot \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_3} \cdot \overline{Q_1} \cdot \overline{Q_0} + \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0}$$

Esta sería la tabla de verdad para el reloj4:

Q_4	Q_3	Q_2	Q_1	Q_0	$reloj4$
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

La función lógica del reloj4 sería:

$$reloj4 = \overline{Q_4}\cdot\overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{Q_0}+\overline{Q_4}\cdot\overline{Q_3}\cdot Q_2\cdot Q_1\cdot Q_0+\overline{Q_4}\cdot Q_3\cdot Q_2\cdot Q_1\cdot\overline{Q_0}+Q_4\cdot\overline{Q_3}\cdot Q_2\cdot\overline{Q_1}\cdot Q_0$$

1.6. Protocolos de transmisión

El módulo codificador devuelve la palabra código como una serie de bits en paralelo, sin embargo, nuestro canal es simplex y la información se transmite en serie. Además, debe haber un sincronismo entre los relojes de los sistemas emisor y receptor.

El emisor envía una palabra de sincronización al inicio de la trama que desea transmitir. Esta palabra de sincronización será 1111 (codificada en Hamming sería 1111111). El receptor se sincroniza con cualquier palabra que comience en 1. Si se produce un error en el primer bit enviado (más significativo de la palabra Hamming 1111111) el receptor no se sincroniza, siendo imposible la interpretación correcta de los datos.

Conseguimos realizar una correcta comunicación, con los módulos: protocolo de envío y protocolo de recepción.

1.6.1. Protocolo de envío

Este módulo sólo realiza la conversión de los datos en paralelo que recibe del módulo codificador a serie, enviando éstos al canal de transmisión.

Puede verse en la implementación física de este módulo en la lámina 7.

1.6.2. Protocolo de recepción

Mientras se reciban 0's (el canal está a un nivel bajo de voltaje), la entrada de habilitación del flip–flop (flip–flop D con entrada de habilitación) estará activa. Al recibir el primer 1, la señal de inicialización se activa, e inmediatamente la entrada de habilitación del flip–flop queda desactivada para siempre. Quedando almacenado en el flip–flop un uno, que deshabilita la señal de inicialización.

En resumen, al recibir el primer uno, la señal de inicialización adquiere un nivel alto durante un tiempo muy pequeño, que reinicia el contador asociado al reloj principal.

1.7. Código fuente de la implementación

1.7.1. Contador de 0 hasta 6

h_contador_06.h

```
/* CONTADOR DESDE 0 HASTA 6 */ /* L'amina 4 */


class CONTADOR_06{
  NEG_JK_FF_PC f[3];
  WIRE Q0n,Q1n,Q2n;
```

```
  WIRE t[3];
  AND a[1];
  OR o[2];
public:
  void run(WIRE &reset, WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2,
  WIRE &voltaje, WIRE &tierra);
};
```

h_contador_06.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_06.h"


void CONTADOR_06::run(WIRE &reset, WIRE &ck, WIRE &Q0, WIRE
&Q1,WIRE &Q2, WIRE &voltaje, WIRE &tierra) {
  f[0].run(t[0],voltaje,ck,voltaje,reset,Q0,Q0n);
  f[1].run(Q0,t[1],ck,voltaje,reset,Q1,Q1n);
  f[2].run(t[2],Q1,ck,voltaje,reset,Q2,Q2n);

  o[0].run(t[0],Q2n,Q1n);
  o[1].run(t[1],Q0,Q2);
  a[0].run(t[2],Q0,Q1);
}
```

test_h_contador_06.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_06.h"
#include "clock.h"
#include <stdio.h>

WIRE voltaje=VCC; WIRE tierra=GND;

void main() {
  int iter=500;
  WIRE c=tierra,Q0=tierra,Q1=tierra,Q2=tierra;
  CONTADOR_06 co;
```

```
  CLOCK clock(20);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,voltaje,tierra);
  while(iter--) {
    clock.run(iter,c);
    co.run(voltaje,c,Q0,Q1,Q2,voltaje,tierra);
    printf("%3d %3d %3d %3d\n",c,Q0,Q1,Q2);
  }
}
```

1.7.2. Contador de 0 hasta 27

h_contador_027.h

```
// CONTADOR DE 0 HASTA 27
// L'amina 6


class CONTADOR_027{
  NEG_JK_FF_PC f[5];
  WIRE Q0n,Q1n,Q2n,Q3n,Q4n;
  AND a[14];
  OR o[7];
  WIRE j[5];
  WIRE k[5];
  WIRE t[12];

public:
  void run(WIRE &clr, WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &Q2,
  WIRE &Q3, WIRE &Q4, WIRE &voltaje, WIRE &tierra);
};
```

h_contador_027.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_027.h"


void CONTADOR_027::run(WIRE &clr, WIRE &ck, WIRE &Q0, WIRE &Q1,
WIRE &Q2, WIRE &Q3, WIRE &Q4, WIRE &voltaje, WIRE &tierra){
  // L'ogica del flip-flop 0
  o[0].run(j[0],3,Q4n,Q3n,Q2n);
  f[0].run(j[0],voltaje,ck,voltaje,clr,Q0,Q0n);

  // L'ogica del flip-flop 1
  a[0].run(t[0],Q2n,Q0);
  a[1].run(t[1],Q3n,Q0);
  a[2].run(t[2],Q4n,Q0);
  o[1].run(j[1],3,t[0],t[1],t[2]);
  a[3].run(t[3],3,Q4,Q3,Q2);
```

```
  o[2].run(k[1],t[3],Q0);
  f[1].run(j[1],k[1],ck,voltaje,clr,Q1,Q1n);

  // L'ogica del flip-flop 2
  a[4].run(t[4],3,Q3n,Q1,Q0);
  a[5].run(t[5],3,Q4n,Q1,Q0);
  o[3].run(j[2],t[4],t[5]);
  a[6].run(t[6],Q1,Q0);
  a[7].run(t[7],Q4,Q3);
  o[4].run(k[2],t[6],t[7]);
  f[2].run(j[2],k[2],ck,voltaje,clr,Q2,Q2n);

  // L'ogica del flip-flop 3
  a[10].run(j[3],3,Q2,Q1,Q0);
  a[8].run(t[8],Q4,Q2);
  a[9].run(t[9],3,Q4,Q1,Q0);
  o[5].run(k[3],3,j[3],t[8],t[9]);
  f[3].run(j[3],k[3],ck,voltaje,clr,Q3,Q3n);

  // L'ogica del flip-flop 4
  a[12].run(t[10],3,Q3,Q1,Q0);
  a[13].run(j[4],t[10],Q2);
  a[11].run(t[11],Q3,Q2);
  o[6].run(k[4],t[10],t[11]);
  f[4].run(j[4],k[4],ck,voltaje,clr,Q4,Q4n);
}
```

test_h_contador_027.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_027.h"
#include "clock.h"
#include <stdio.h>

WIRE voltaje=VCC; WIRE tierra=GND;

void main() {
  int iter=1000;
  WIRE c=0,Q0=0,Q1=0,Q2=0,Q3=0,Q4=0;
```

```
  CONTADOR_027 co;
  CLOCK clock(20);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
  while(iter--) {
    clock.run(iter,c);
    co.run(voltaje,c,Q0,Q1,Q2,Q3,Q4,voltaje,tierra);
    printf("%3d %3d %3d %3d %3d %3d\n",c,Q0,Q1,Q2,Q3,Q4);
  }
}
```

1.7.3. Contador de 2 bits

h_contador_2bits.h

```
// CONTADOR DE 2 BITS
// L'amina 9


class CONTADOR_2BITS{
  NEG_JK_FF_PC f[2];
  WIRE Q0n,Q1n;
public:
  void run(WIRE &reset, WIRE &ck, WIRE &Q0, WIRE &Q1,
  WIRE &voltaje, WIRE &tierra);
};
```

h_contador_2bits.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_2bits.h"


/* Contador de 2 bits */
void CONTADOR_2BITS::run(WIRE &reset,
WIRE &ck, WIRE &Q0, WIRE &Q1, WIRE &voltaje, WIRE &tierra) {
  f[0].run(voltaje,voltaje,ck,voltaje,reset,Q0,Q0n);
  f[1].run(Q0,Q0,ck,voltaje,reset,Q1,Q1n);
}
```

test_h_contador_2bits.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_2bits.h"
#include "clock.h"
#include<stdio.h>

WIRE voltaje = VCC; WIRE tierra = GND;

void main() {
```

```
  int iter=5000;
  WIRE c=0,Q0=0,Q1=0;
  CONTADOR_2BITS co;
  CLOCK clock(50);

  co.run(tierra,c,Q0,Q1,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,voltaje,tierra);
  co.run(tierra,c,Q0,Q1,voltaje,tierra);

  while(iter--) {
    clock.run(iter,c);
    co.run(voltaje,c,Q0,Q1,voltaje,tierra);
    printf("%3d %3d %3d\n",c,Q0,Q1);
  }
}
```

1.7.4. Reloj múltiple

h_reloj.h

```
// RELOJ MULTIPLE
// Lamina 5

class RELOJ{
  CONTADOR_027 contador;
  WIRE s[5];
  NOT n[5];
  AND a[7];
  OR o[2];
  WIRE t[11];
  WIRE ns[5];
  D_LATCH latch[2];

public:
  void run(WIRE &ck, WIRE &reset, WIRE &reloj4, WIRE &reloj7,
  WIRE &voltaje, WIRE &tierra);
};
```

h_reloj.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_p_recibe.h"

void P_RECIBE::inicializa(WIRE &voltaje, WIRE &tierra){
  t[0] = tierra;
  t[1] = tierra;
  t[2] = tierra;
  t[3] = tierra;
  t[4] = tierra;
  t[5] = voltaje;
  t[6] = tierra;
  t[7] = voltaje;
}

void P_RECIBE::run(WIRE &entrada, WIRE &salida, WIRE
&inicializacion, WIRE &voltaje, WIRE &tierra) {
```

```
  latch[0].run(entrada,t[4],t[5]);
  latch[1].run(t[4],t[6],t[7]);

  n.run(t[2],t[1]);
  a.run(inicializacion,t[2],t[6]);
  f.run(entrada,t[0],t[1],Qn);
  na.run(t[0],t[6],t[2]);
  salida = t[6];
}
```

test_h_reloj.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_027.h"
#include "h_reloj.h"
#include"clock.h"
#include <stdio.h>


WIRE voltaje=VCC; WIRE tierra=GND;

void main() {
  int iter=5000;
  WIRE c=tierra,reloj7,reloj4;
  RELOJ reloj;
  CLOCK clock(20);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,reloj4,reloj7,voltaje,tierra);
  while(iter--) {
    clock.run(iter,c);
    reloj.run(c,voltaje,reloj4,reloj7,voltaje,tierra);
    printf("%3d %3d %3d\n",c,reloj4,reloj7);
  }
}
```

1.7.5. Retardo

h_retardo.h

```
// RETARDO DE PROPAGACI'ON DE 6 NIVELES
class RETARDO{
  NOT n[6];
  WIRE t[5];

public:
  void run(WIRE &salida, WIRE &entrada, WIRE &voltaje, WIRE &tierra);
};
```

h_retardo.c

```
#include "gates.h"
#include "h_retardo.h"

void RETARDO::run(WIRE &salida, WIRE &entrada, WIRE &voltaje, WIRE
&tierra) {
  n[0].run(t[0],entrada);
  n[1].run(t[1],t[0]);
  n[2].run(t[2],t[1]);
  n[3].run(t[3],t[2]);
  n[4].run(t[4],t[3]);
  n[5].run(salida,t[4]);
}
```

test_h_retardo.c

```
#include "gates.h"
#include "clock.h"
#include "h_retardo.h"
#include <stdio.h>

WIRE voltaje=VCC; WIRE tierra=GND;

void main() {
  int iter=1000;
  WIRE  entrada=tierra,salida=tierra;
  CLOCK c(20);
  RETARDO ret;
```

```
  while(iter--) {
    c.run(iter,entrada);
    ret.run(salida,entrada,voltaje,tierra);
    printf("%3d %3d\n",entrada, salida);
  }
}
```

1.7.6. Codificador Hamming

h_codificador.h

```
// CODIFICADOR HAMMING
// L'amina 1


class CODIFICADOR{
  CONTADOR_2BITS contador;
  NEG_D_FF f[11];
  AND a[12];
  XOR xo[3];
  AND na[1];
  WIRE Qn[11];
  WIRE t[25];
  RETARDO ret;
  D_LATCH latch;

public:
  void inicializa(WIRE &voltaje, WIRE &tierra);
  void run(WIRE &reset, WIRE &ck, WIRE &dato,
  WIRE &s0, WIRE &s1, WIRE &s2, WIRE &s3,
  WIRE &s4, WIRE &s5, WIRE &s6,
  WIRE &voltaje, WIRE &tierra);
};
```

h_codificador.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_2bits.h"
#include "h_retardo.h"
#include"h_codificador.h"
#include <stdio.h>


void CODIFICADOR::inicializa(WIRE &voltaje, WIRE &tierra){
  for (int i=0;i<11;i++)
    Qn[i]=voltaje;
```

```
  for (int i=0;i<25;i++)
    t[i]=tierra;

}

void CODIFICADOR::run(WIRE &reset, WIRE &ck, WIRE &dato,
                      WIRE &s0, WIRE &s1, WIRE &s2, WIRE &s3, WIRE &s4, WIRE &s5, WI
                      WIRE &voltaje, WIRE &tierra){

  contador.run(reset,ck,t[20],t[21],voltaje,tierra);

  // registros de desplazamiento
  f[0].run(dato,ck,t[0],Qn[0]);
  f[1].run(t[0],ck,t[1],Qn[1]);
  f[2].run(t[1],ck,t[2],Qn[2]);
  f[3].run(t[2],ck,t[3],Qn[3]);

  // logica de codificaci'on
  xo[0].run(t[16],4,t[4],t[5],t[6],t[7]);
  a[0].run(t[4],t[3],tierra);
  a[1].run(t[5],t[2],voltaje);
  a[2].run(t[6],t[1],voltaje);
  a[3].run(t[7],t[0],voltaje);

  xo[1].run(t[17],4,t[8],t[9],t[10],t[11]);
  a[4].run(t[8],t[3],voltaje);
  a[5].run(t[9],t[2],tierra);
  a[6].run(t[10],t[1],voltaje);
  a[7].run(t[11],t[0],voltaje);

  xo[2].run(t[18],4,t[12],t[13],t[14],t[15]);
  a[8].run(t[12],t[3],voltaje);
  a[9].run(t[13],t[2],voltaje);
  a[10].run(t[14],t[1],tierra);
  a[11].run(t[15],t[0],voltaje);

  // control de la salida
  na[0].run(t[19],t[20],t[21]);
  latch.run(t[19],t[22],t[23]);

  ret.run(t[24],t[22],voltaje,tierra);
```

```
  f[4].run(t[3],t[24],s6,Qn[4]);
  f[5].run(t[2],t[24],s5,Qn[5]);
  f[6].run(t[1],t[24],s4,Qn[6]);
  f[7].run(t[0],t[24],s3,Qn[7]);
  f[8].run(t[16],t[24],s2,Qn[8]);
  f[9].run(t[17],t[24],s1,Qn[9]);
  f[10].run(t[18],t[24],s0,Qn[10]);

}
```

test_h_codificador.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_2bits.h"
#include "clock.h"
#include"h_retardo.h"
#include "h_codificador.h"
#include<stdio.h>

WIRE voltaje = VCC; WIRE tierra = GND;

void main (void){
  WIRE c = tierra;
  WIRE s0=tierra,s1=tierra,s2=tierra,s3=tierra,s4=tierra,
  s5=tierra,s6=tierra;
  int i,iter;

  CODIFICADOR codificador;
  CLOCK clock(30);

  WIRE entrada[600];

    for (i=0;i<60;i++)
    {
      entrada[i]=tierra;
    }

    for (i=0;i<60;i++)
```

```
{
  entrada[i+60]=tierra;
}

for (i=0;i<60;i++)
{
  entrada[i+120]=tierra;
}

for (i=0;i<60;i++)
{
  entrada[i+180]=voltaje;
}

for (i=0;i<60;i++)
{
  entrada[i+240]=tierra;
}

for (i=0;i<60;i++)
{
  entrada[i+300]=tierra;
}

for (i=0;i<60;i++)
{
  entrada[i+360]=voltaje;
}

for (i=0;i<60;i++)
{
  entrada[i+420]=voltaje;
}

for (i=0;i<60;i++)
{
  entrada[i+480]=voltaje;
}

for (i=0;i<60;i++)
{
```

```
    entrada[i+540]=tierra;
  }

  codificador.inicializa(voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);
  codificador.run(tierra,c,entrada[0],s0,s1,s2,s3,s4,s5,s6,voltaje,tierra);

  for (iter = 0; iter < 600; iter++) {
    clock.run(iter,c);
    codificador.run(voltaje,c,entrada[iter],s0,s1,s2,s3,s4,s5,s6,
      voltaje,tierra);
    printf("%3d %3d   %3d %3d %3d %3d %3d %3d %3d\n",c,
      entrada[iter],s0,s1,s2,s3,s4,s5,s6);
  }
}
```

1.7.7. Decodificador Hamming

h_decodificador.h

```
// DECODIFICADOR HAMMING
// L'amina 3


class DECODIFICADOR{
  CONTADOR_06 contador;
  NEG_D_FF f[11];
  AND a[25];
  XOR xo[7];
  AND na[1];
  WIRE Qn[11];
  NOT n[4];
  WIRE t[50];
  RETARDO ret;
  D_LATCH latch;

public:
  void inicializa(WIRE &voltaje, WIRE &tierra);
  void run(WIRE &reset, WIRE &ck, WIRE &dato,WIRE &s0, WIRE &s1,
  WIRE &s2, WIRE &s3,WIRE &voltaje, WIRE &tierra);
};
```

h_decodificador.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_06.h"
#include "h_retardo.h"
#include"h_decodificador.h"


void DECODIFICADOR::inicializa(WIRE &voltaje, WIRE &tierra){
  for (int i=0;i<11;i++)
    Qn[i]=voltaje;

  for (int i=0;i<50;i++)
    t[i]=tierra;
```

```
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
  contador.run(tierra,tierra,t[42],t[43],t[44],voltaje,tierra);
}

void DECODIFICADOR::run(WIRE &reset, WIRE &ck, WIRE &dato,WIRE
&s0, WIRE &s1, WIRE &s2, WIRE &s3,WIRE &voltaje, WIRE &tierra){


  contador.run(reset,ck,t[42],t[43],t[44],voltaje,tierra);
  // registros de desplazamiento
  f[0].run(dato,ck,t[0],Qn[0]);
  f[1].run(t[0],ck,t[1],Qn[1]);
  f[2].run(t[1],ck,t[2],Qn[2]);
  f[3].run(t[2],ck,t[3],Qn[3]);
  f[4].run(t[3],ck,t[4],Qn[4]);
  f[5].run(t[4],ck,t[5],Qn[5]);
  f[6].run(t[5],ck,t[6],Qn[6]);

  // C'alculo del s'indrome
  xo[0].run(t[28],7,t[7],t[8],t[9],t[10],t[11],t[12],t[13]);
  a[0].run(t[7],t[6],tierra);
  a[1].run(t[8],t[5],tierra);
  a[2].run(t[9],t[4],tierra);
  a[3].run(t[10],t[3],voltaje);
  a[4].run(t[11],t[2],voltaje);
  a[5].run(t[12],t[1],voltaje);
  a[6].run(t[13],t[0],voltaje);

  xo[1].run(t[29],7,t[14],t[15],t[16],t[17],t[18],t[19],t[20]);
  a[7].run(t[14],t[6],tierra);
  a[8].run(t[15],t[5],voltaje);
  a[9].run(t[16],t[4],voltaje);
  a[10].run(t[17],t[3],tierra);
  a[11].run(t[18],t[2],tierra);
  a[12].run(t[19],t[1],voltaje);
  a[13].run(t[20],t[0],voltaje);
```

```
  xo[2].run(t[30],7,t[21],t[22],t[23],t[24],t[25],t[26],t[27]);
  a[14].run(t[21],t[6],voltaje);
  a[15].run(t[22],t[5],tierra);
  a[16].run(t[23],t[4],voltaje);
  a[17].run(t[24],t[3],tierra);
  a[18].run(t[25],t[2],voltaje);
  a[19].run(t[26],t[1],tierra);
  a[20].run(t[27],t[0],voltaje);

  // L'ogica de decodificaci'on
  n[0].run(t[31],t[28]);
  n[1].run(t[32],t[29]);
  n[2].run(t[33],t[30]);

  a[21].run(t[34],3,t[31],t[32],t[30]);
  a[22].run(t[35],3,t[31],t[29],t[33]);
  a[23].run(t[36],3,t[31],t[29],t[30]);
  a[24].run(t[37],3,t[28],t[32],t[33]);

  xo[3].run(t[38],t[34],t[6]);
  xo[4].run(t[39],t[35],t[5]);
  xo[5].run(t[40],t[36],t[4]);
  xo[6].run(t[41],t[37],t[3]);

  // control de la salida
  n[3].run(t[45],t[42]);
  na[0].run(t[46],3,t[43],t[44],t[45]);
  latch.run(t[46],t[47],t[48]);

  ret.run(t[49],t[47],voltaje,tierra);

  f[7].run(t[38],t[49],s3,Qn[7]);
  f[8].run(t[39],t[49],s2,Qn[8]);
  f[9].run(t[40],t[49],s1,Qn[9]);
  f[10].run(t[41],t[49],s0,Qn[10]);

}
```

test_h_decodificador.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_contador_06.h"
#include "clock.h"
#include"h_retardo.h"
#include "h_decodificador.h"
#include<stdio.h>

WIRE voltaje = VCC; WIRE tierra = GND;

void main (void){

WIRE c = tierra; WIRE s0=tierra,s1=tierra,s2=tierra,s3=tierra; int
i,iter;

DECODIFICADOR decodificador; CLOCK clock(10);

WIRE entrada[200];

  for (i=0;i<20;i++)
  {
        entrada[i]=voltaje;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+20]=tierra;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+40]=tierra;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+60]=voltaje;
```

```
  }
  for (i=0;i<20;i++)
  {
        entrada[i+80]=voltaje;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+100]=tierra;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+120]=voltaje;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+140]=voltaje;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+160]=voltaje;

  }
  for (i=0;i<20;i++)
  {
        entrada[i+180]=voltaje;

  }

  decodificador.inicializa(voltaje,tierra);
  for (iter = 0; iter < 200; iter++) {
    clock.run(iter,c);
    decodificador.run(voltaje,c,entrada[iter],s0,s1,s2,s3,voltaje,tierra);
    printf("%3d %3d   %3d %3d %3d %3d\n",c,entrada[iter],s0,s1,s2,s3);
  }
}
```

1.7.8. Protocolo de envío

h_p_envio.h

```
// PROTOCOLO DE ENV'IO
// L'amina 7

class P_ENVIO{
  CONTADOR_06 contador;
  WIRE t[3];
  MUX multiplexor;

public:
  void inicializa(WIRE &voltaje,WIRE &tierra);
  void run(WIRE &reset,WIRE datos[], WIRE &ck, WIRE &salida,
  WIRE &voltaje,WIRE &tierra);

};
```

h_p_envio.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "decods.h"
#include "muxs.h"
#include "h_contador_06.h"
#include "h_p_envio.h"


void P_ENVIO::inicializa(WIRE &voltaje, WIRE &tierra){
  multiplexor.create(3);
  t[0]=tierra;
  t[1]=tierra;
  t[2]=tierra;
}

void P_ENVIO::run(WIRE &reset, WIRE datos[], WIRE &ck, WIRE
&salida,WIRE &voltaje,WIRE &tierra){
  contador.run(reset,ck,t[0],t[1],t[2],voltaje,tierra);
  multiplexor.run(datos,t,salida);
}
```

test_h_p_envio.c

```
#include "gates.h"
#include "clock.h"
#include "latches.h"
#include "decods.h"
#include "flip-flops.h"
#include "muxs.h"
#include "h_contador_06.h"
#include "h_p_envio.h"
#include<stdio.h>

WIRE voltaje=VCC; WIRE tierra=GND;

void main() {
  int iter=420;
  WIRE s=tierra;
  WIRE c=tierra;
  WIRE v[8];
   v[0]=voltaje;
   v[1]=tierra;
   v[2]=voltaje;
   v[3]=tierra;
   v[4]=voltaje;
   v[5]=tierra;
   v[6]=voltaje;
   v[7]=tierra;

  P_ENVIO p;
  CLOCK clock(30);

  p.inicializa(voltaje,tierra);

  p.run(tierra,v,c,s,voltaje,tierra);
  p.run(tierra,v,c,s,voltaje,tierra);
  p.run(tierra,v,c,s,voltaje,tierra);
  p.run(tierra,v,c,s,voltaje,tierra);
  p.run(tierra,v,c,s,voltaje,tierra);
  p.run(tierra,v,c,s,voltaje,tierra);

  while(iter--) {
```

```
    clock.run(iter,c);
    p.run(voltaje,v,c,s,voltaje,tierra);
    printf("%3d %3d\n",c,s);
  }
}
```

1.7.9. Protocolo de recepción

h_p_recibe.h

```
// PROTOCOLO DE RECEPCI'ON
// L'amina 8

class P_RECIBE{
  NEG_D_FF f;
  NAND a;
  NAND na;
  NOT n;
  WIRE t[8];
  WIRE Qn;
  D_LATCH latch[2];

public:
  void inicializa(WIRE &voltaje, WIRE &tierra);
  void run(WIRE &entrada, WIRE &salida, WIRE &inicializacion,
  WIRE &voltaje, WIRE &tierra);
};
```

h_p_recibe.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_p_recibe.h"

void P_RECIBE::inicializa(WIRE &voltaje, WIRE &tierra){
  t[0] = tierra;
  t[1] = tierra;
  t[2] = tierra;
  t[3] = tierra;
  t[4] = tierra;
  t[5] = voltaje;
  t[6] = tierra;
  t[7] = voltaje;
}

void P_RECIBE::run(WIRE &entrada, WIRE &salida, WIRE
&inicializacion, WIRE &voltaje, WIRE &tierra) {
```

```
  latch[0].run(entrada,t[4],t[5]);
  latch[1].run(t[4],t[6],t[7]);

  n.run(t[2],t[1]);
  a.run(inicializacion,t[2],t[6]);
  f.run(entrada,t[0],t[1],Qn);
  na.run(t[0],t[6],t[2]);
  salida = t[6];
}
```

test_h_p_recibe.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_retardo.h"
#include "h_p_recibe.h"
#include <stdio.h>


WIRE tierra = GND; WIRE voltaje = VCC;

void main() {
  int i;
  int iter;
  WIRE salida = tierra;
  WIRE ini = tierra;
  P_RECIBE pr;
  WIRE entrada[200];

  for (i=0;i<=19;i++)
  {
        entrada[i]=tierra;

  }
  for (i=0;i<=19;i++)
  {
        entrada[i+20]=tierra;

  }
  for (i=0;i<=19;i++)
```

```
{
      entrada[i+40]=tierra;

}
for (i=0;i<=19;i++)
{
      entrada[i+60]=voltaje;

}
for (i=0;i<=19;i++)
{
      entrada[i+80]=tierra;

}
for (i=0;i<=19;i++)
{
      entrada[i+100]=tierra;

}
for (i=0;i<=19;i++)
{
      entrada[i+120]=voltaje;

}
for (i=0;i<=19;i++)
{
      entrada[i+140]=voltaje;

}
for (i=0;i<=19;i++)
{
      entrada[i+160]=voltaje;

}
for (i=0;i<=19;i++)
{
      entrada[i+180]=tierra;

}

pr.inicializa(voltaje,tierra);
```

```
  for (iter = 0; iter < 200;iter++) {

    pr.run(entrada[iter],salida,ini,voltaje,tierra);

    printf("%3d %3d %3d\n",entrada[iter],salida,ini);
  }
}
```

1.7.10. Módulo emisor

h_emisor.h

```
// M'ODULO EMISOR
//

class EMISOR{
  WIRE t[7];
  WIRE reloj7;
  CODIFICADOR codificador;
  P_ENVIO p_envio;
  RELOJ reloj;

public:
  EMISOR(WIRE &voltaje, WIRE &tierra);
  void run(WIRE &reset, WIRE &ck, WIRE &entrada, WIRE &salida,
  WIRE &reloj4, WIRE &voltaje, WIRE &tierra);

};
```

h_emisor.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "decods.h"
#include "muxs.h"
#include "h_retardo.h"
#include "h_contador_06.h"
#include "h_contador_2bits.h"
#include"h_contador_027.h"
#include "h_reloj.h"
#include"h_codificador.h"
#include "h_p_envio.h"
#include"h_emisor.h"




EMISOR::EMISOR(WIRE &voltaje, WIRE &tierra){
  WIRE s=tierra;
```

```
  for (int i = 0; i < 7; i++){
        t[i] = tierra;
  }

  codificador.inicializa(voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);
  codificador.run(tierra,tierra,tierra,t[0],t[1],t[2],t[3],t[4],t[5],
    t[6],voltaje,tierra);

  p_envio.inicializa(voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);
  p_envio.run(tierra,t,tierra,s,voltaje,tierra);

  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);
  reloj.run(tierra,tierra,s,reloj7,voltaje,tierra);

  reloj7 = tierra;
}


void EMISOR::run(WIRE &reset, WIRE &ck, WIRE &entrada, WIRE
&salida, WIRE &reloj4, WIRE &voltaje, WIRE &tierra){
  reloj.run(ck,reset,reloj4,reloj7,voltaje,tierra);
```

```
  codificador.run(reset,reloj4,entrada,t[0],t[1],t[2],t[3],t[4],t[5],t[6],voltaje,ti
  p_envio.run(reset,t,reloj7,salida,voltaje,tierra);
}
```

test_h_emisor.c

```
#include "gates.h"
#include "clock.h"
#include"latches.h"
#include"flip-flops.h"
#include "decods.h"
#include"muxs.h"
#include "h_retardo.h"
#include "h_contador_06.h"
#include"h_contador_2bits.h"
#include "h_contador_027.h"
#include"h_reloj.h"
#include "h_codificador.h"
#include"h_p_envio.h"
#include "h_emisor.h"
#include <stdio.h>


WIRE voltaje = VCC; WIRE tierra = GND;


void main (void){

  CLOCK clock(20);
  WIRE salida = tierra;
  WIRE reloj4 = tierra;
  WIRE c = tierra;

  EMISOR emisor(voltaje,tierra);

  int iter = 0;
  int k,n;

  WIRE dato[40];
    dato[0]=voltaje;
    dato[1]=tierra;
```

```
  dato[2]=voltaje;
  dato[3]=tierra;
  dato[4]=voltaje;
  dato[5]=tierra;
  dato[6]=voltaje;
  dato[7]=tierra;
  dato[8]=voltaje;
  dato[9]=tierra;
  dato[10]=voltaje;
  dato[11]=tierra;
  dato[12]=voltaje;
  dato[13]=tierra;
  dato[14]=voltaje;
  dato[15]=tierra;
  dato[16]=voltaje;
  dato[17]=tierra;
  dato[18]=voltaje;
  dato[19]=tierra;
  dato[20]=voltaje;
  dato[21]=tierra;
  dato[22]=voltaje;
  dato[23]=tierra;
  dato[24]=voltaje;
  dato[25]=tierra;
  dato[26]=voltaje;
  dato[27]=tierra;
  dato[28]=voltaje;
  dato[29]=tierra;
  dato[30]=voltaje;
  dato[31]=tierra;
  dato[32]=voltaje;
  dato[33]=tierra;
  dato[34]=voltaje;
  dato[35]=tierra;
  dato[36]=voltaje;
  dato[37]=tierra;
  dato[38]=voltaje;
  dato[39]=tierra;

WIRE entrada[1600];
for (k=0;k<40;k++)
```

```
    for (n=0;n<40;n++)
      entrada[n+(k*40)]=dato[k];

  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,tierra,entrada[0],salida,reloj4,voltaje,tierra);


  for (iter = 0; iter < 1600; iter++)
  {
     clock.run(iter,c);
     emisor.run(voltaje,c,entrada[iter],salida,reloj4,voltaje,tierra);
     //printf("%3d  %3d  %3d         %3d\n",c,entrada[iter],salida,reloj4);
  }
}
```

1.7.11. Módulo de recepción

h_receptor.h

```
// M'ODULO RECEPTOR
//

class RECEPTOR{
  WIRE t[2];
  WIRE reloj7;
  DECODIFICADOR decodificador;
  P_RECIBE p_recibe;
  RELOJ reloj;

public:
  RECEPTOR(WIRE &voltaje, WIRE &tierra);
  void run(WIRE &ck, WIRE &entrada, WIRE &s0, WIRE &s1, WIRE &s2,
  WIRE &s3, WIRE &voltaje, WIRE &tierra);

};
```

h_receptor.c

```
#include "gates.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_p_recibe.h"

void P_RECIBE::inicializa(WIRE &voltaje, WIRE &tierra){
  t[0] = tierra;
  t[1] = tierra;
  t[2] = tierra;
  t[3] = tierra;
  t[4] = tierra;
  t[5] = voltaje;
  t[6] = tierra;
  t[7] = voltaje;
}

void P_RECIBE::run(WIRE &entrada, WIRE &salida, WIRE
&inicializacion, WIRE &voltaje, WIRE &tierra) {
  latch[0].run(entrada,t[4],t[5]);
```

```
  latch[1].run(t[4],t[6],t[7]);

  n.run(t[2],t[1]);
  a.run(inicializacion,t[2],t[6]);
  f.run(entrada,t[0],t[1],Qn);
  na.run(t[0],t[6],t[2]);
  salida = t[6];
}
```

test_h_receptor.c

```
#include "gates.h"
#include "clock.h"
#include "latches.h"
#include "flip-flops.h"
#include "h_retardo.h"
#include"h_contador_06.h"
#include "h_contador_027.h"
#include"h_reloj.h"
#include "h_decodificador.h"
#include"h_p_recibe.h"
#include "h_receptor.h"
#include <stdio.h>


WIRE voltaje = VCC; WIRE tierra = GND;


void main (void){

  CLOCK clock(30);
  WIRE c = tierra;
  WIRE s0=tierra, s1=tierra, s2=tierra, s3=tierra;

  RECEPTOR receptor(voltaje,tierra);

  int iter = 0;
  int k,n;

  WIRE dato[60];
    dato[0]=tierra;
```

```
dato[1]=tierra;
dato[2]=tierra;
dato[3]=tierra;
dato[4]=voltaje;
dato[5]=voltaje;
dato[6]=voltaje;
dato[7]=voltaje;
dato[8]=voltaje;
dato[9]=voltaje;
dato[10]=voltaje;
dato[11]=voltaje;
dato[12]=voltaje;
dato[13]=voltaje;
dato[14]=voltaje;
dato[15]=voltaje;
dato[16]=voltaje;
dato[17]=voltaje;
dato[18]=voltaje;
dato[19]=voltaje;
dato[20]=voltaje;
dato[21]=voltaje;
dato[22]=voltaje;
dato[23]=voltaje;
dato[24]=voltaje;
dato[25]=voltaje;
dato[26]=voltaje;
dato[27]=voltaje;
dato[28]=voltaje;
dato[29]=voltaje;
dato[30]=voltaje;
dato[31]=voltaje;
dato[32]=voltaje;
dato[33]=voltaje;
dato[34]=voltaje;
dato[35]=voltaje;
dato[36]=voltaje;
dato[37]=voltaje;
dato[38]=voltaje;
dato[39]=voltaje;
dato[40]=voltaje;
dato[41]=voltaje;
```

```
    dato[42]=voltaje;
    dato[43]=voltaje;
    dato[44]=voltaje;
    dato[45]=voltaje;
    dato[46]=voltaje;
    dato[47]=voltaje;
    dato[48]=voltaje;
    dato[49]=voltaje;
    dato[50]=voltaje;
    dato[51]=voltaje;
    dato[52]=voltaje;
    dato[53]=voltaje;
    dato[54]=voltaje;
    dato[55]=voltaje;
    dato[56]=voltaje;
    dato[57]=voltaje;
    dato[58]=voltaje;
    dato[59]=voltaje;

  WIRE entrada[2400];
  for (k=0;k<60;k++)
    for (n=0;n<40;n++)
      entrada[n+(k*40)]=dato[k];

  for (iter = 0; iter < 2400; iter++)
  {
     clock.run(iter,c);
     receptor.run(c,entrada[iter],s0,s1,s2,s3,voltaje,tierra);
     printf("%3d  %3d       %3d  %3d  %3d %3d\n",c,entrada[iter],s0,s1,s2,s3);
  }
}
```

1.7.12. Entrenador general

```
#include "gates.h"
#include "clock.h"
#include "latches.h"
#include "flip-flops.h"
#include "decods.h"
#include "muxs.h"
#include "h_retardo.h"
```

```
#include "h_contador_06.h"
#include"h_contador_2bits.h"
#include "h_contador_027.h"
#include"h_reloj.h"
#include "h_p_recibe.h"
#include"h_codificador.h"
#include "h_p_envio.h"
#include"h_decodificador.h"
#include "h_emisor.h"
#include"h_receptor.h"
#include <stdio.h>

WIRE voltaje = VCC; WIRE tierra = GND;


void main (void) {
  int iter = 0;
  int n=0;
  int k=0;

  WIRE s0 = tierra;
  WIRE s1 = tierra;
  WIRE s2 = tierra;
  WIRE s3 = tierra;

  EMISOR emisor(voltaje,tierra);
  RECEPTOR receptor(voltaje,tierra);


  WIRE c=tierra;
  WIRE salida=tierra;
  WIRE reloj4=tierra;

  CLOCK clock(30);

  WIRE dato[80];


  dato[0]=voltaje;
  dato[1]=voltaje;
  dato[2]=voltaje;
```

```
dato[3]=voltaje;
dato[4]=voltaje;
dato[5]=voltaje;
dato[6]=voltaje;
dato[7]=voltaje;
dato[8]=voltaje;
dato[9]=voltaje;
dato[10]=tierra;
dato[11]=tierra;
dato[12]=tierra;
dato[13]=tierra;
dato[14]=tierra;
dato[15]=tierra;
dato[16]=tierra;
dato[17]=tierra;
dato[18]=voltaje;
dato[19]=voltaje;
dato[20]=voltaje;
dato[21]=voltaje;
dato[22]=voltaje;
dato[23]=voltaje;
dato[24]=voltaje;
dato[25]=voltaje;
dato[26]=voltaje;
dato[27]=voltaje;
dato[28]=voltaje;
dato[29]=voltaje;
dato[30]=voltaje;
dato[31]=voltaje;
dato[32]=voltaje;
dato[33]=voltaje;
dato[34]=voltaje;
dato[35]=voltaje;
dato[36]=voltaje;
dato[37]=voltaje;
dato[38]=voltaje;
dato[39]=voltaje;
dato[40]=voltaje;
dato[41]=voltaje;
dato[42]=voltaje;
dato[43]=voltaje;
```

```
dato[44]=voltaje;
dato[45]=voltaje;
dato[46]=voltaje;
dato[47]=voltaje;
dato[48]=voltaje;
dato[49]=voltaje;
dato[50]=voltaje;
dato[51]=voltaje;
dato[52]=voltaje;
dato[53]=voltaje;
dato[54]=voltaje;
dato[55]=voltaje;
dato[56]=voltaje;
dato[57]=voltaje;
dato[58]=voltaje;
dato[59]=voltaje;
dato[60]=voltaje;
dato[61]=voltaje;
dato[62]=voltaje;
dato[63]=voltaje;
dato[64]=voltaje;
dato[65]=voltaje;
dato[66]=voltaje;
dato[67]=voltaje;
dato[68]=voltaje;
dato[69]=voltaje;
dato[70]=voltaje;
dato[71]=voltaje;
dato[72]=voltaje;
dato[73]=voltaje;
dato[74]=voltaje;
dato[75]=voltaje;
dato[76]=voltaje;
dato[77]=voltaje;
dato[78]=voltaje;
dato[79]=voltaje;

WIRE entrada[4800];
for (k=0;k<80;k++)
  for (n=0;n<60;n++)
    entrada[n+(k*60)]=dato[k];
```

```
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);
  emisor.run(tierra,c,entrada[0],salida,reloj4,voltaje,tierra);


  for (iter = 0; iter < 4800; iter++)
  {
    clock.run(iter,c);
    emisor.run(voltaje,c,entrada[iter],salida,reloj4,voltaje,tierra);
    receptor.run(c,salida,s0,s1,s2,s3,voltaje,tierra);
    printf("%3d %3d %3d          %3d %3d %3d %3d\n",c, entrada[iter],
      salida,s0,s1,s2,s3);
  }

}
```

Apéndice A

Teoría de códigos lineales

A.1. Conceptos algebraicos fundamentales

A.1.1. Definiciones

En esta sección se hará una introducción al álgebra binaria, que es la que define las estructuras y operaciones sobre las que más tarde basaremos la construcción de los códigos lineales.

Primeramente, definiremos los conceptos de *grupo* y *cuerpo* para describir los **cuerpos de Galois**, y más tarde enumeraremos sus principales propiedades.

Sea G un conjunto genérico de elementos:

Diremos que...

- G es un **grupo** $\Leftrightarrow$
  1. Tiene definida la operación binaria: $*$.
  2. La operación $*$ es asociativa: $\forall a, b, c \in G, a * (b + c) = a * b + a * c$.
  3. Existe elemento neutro: $\exists e \in G$ tal que $\forall a \in G, a * e = e * a$.
  4. Existe elemento inverso: $\forall a \in G, \exists a' \in G$ tal que $a * a' = e$.
- G es un **grupo conmutativo** $\Leftrightarrow$
  1. $\forall a, b \in G, a * b = b * a$.
  2. $(G, *)$ es un grupo.

Sea ahora F otro conjunto de elementos, con dos operaciones: $+$ y $\cdot$.

Diremos que...

- F es un **cuerpo** $\Leftrightarrow$
  1. F es un **grupo conmutativo** bajo la operación $+$.
  2. $\forall a \in F, a \neq 0 \Rightarrow aF$ es un grupo conmutativo para la operación $\cdot$.

3. La operación $\cdot$ es distributiva con respecto a $+$: $\forall a, b \in G, a \cdot (b + c) = a \cdot b + a \cdot c$.

- F es un **cuerpo de Galois** $\Leftrightarrow$
  1. F es un cuerpo.
  2. El alfabeto de F es finito.

Concretemos ahora las definiciones generales anteriores para el caso que nos interesa, el álgebra binaria:

- El alfabeto fuente de nuestros conjuntos es ahora $A = (0, 1)$.
- Las operaciones $+$ (XOR) y $\cdot$ (AND) quedan definidas de la siguiente forma:

$+$	**0**	**1**
0	0	1
1	1	0

$\cdot$	**0**	**1**
0	0	0
1	0	1

Se puede demostrar, por lo tanto, que el conjunto $A = (0, 1)$ junto con las operaciones anteriores cumplen las condiciones necesarios para construir un *cuerpo de Galois* $GF(2)$.

A.1.2. Propiedades de GF(2)

1. Sea $\mathbf{V_n}$ un conjunto de n–tuplas en la forma $\vec{v} = (v_1, \ldots, v_n)$ y donde $v_i \in GF(2)$. Las operaciones $+$ y $\cdot$ quedan así definidas:
   - $\vec{v} + \vec{u} = \vec{w}$, donde $w_i = v_i + u_i$.
   - $a \cdot \vec{v} = \vec{w}$, donde $w_i = a \cdot v_i$.

   El conjunto $\mathbf{V_n}$ es un *espacio vectorial* sobre $GF(2)$ con 2^n elementos y de dimensión n.

2. Sea $S \subseteq \mathbf{V_n}$. Diremos que es un subespacio vectorial suyo $\Leftrightarrow$
   - $\emptyset \in S$.
   - $\forall \vec{u}, \vec{v} \in S, (\vec{u} + \vec{v}) \in S$. Esta propiedad va a resultar especialmente importante para nuestros códigos.
   - $(a \cdot \vec{u}) \in S, \forall a \in GF(2)$ y $\forall \vec{u} \in S$.

3. Diremos que los vectores $\vec{v_1}, \ldots, \vec{v_n}$ son linealmente independientes $\Leftrightarrow$ no $\exists c_1, \ldots, c_n \in GF(2)$ tal que $c_1 \cdot \vec{v_1} + \ldots + c_n \cdot \vec{v_n} = \emptyset$ y $(c_1, \ldots, c_n) \neq \emptyset$.

4. El conjunto de vectores $\{\vec{g_1}, \ldots, \vec{g_n}\}$ genera el espacio vectorial $\mathbf{V_n} \Leftrightarrow \forall \vec{v} \in \mathbf{V_n}, \vec{v} = \sum_{i=1}^{n}(c_i \cdot \vec{g_i})$.
5. $\forall \mathbf{V_n}, \exists \{\vec{g_1}, \ldots, \vec{g_n}\} \in \mathbf{V_n}$ y linealmente independientes que generan es espacio $\mathbf{V_n}$.
6. Si $k > m$ y $\{\vec{v_1}, \ldots, \vec{v_n}\} \in \mathbf{V_n}$ son linealmente independientes, entonces el conjunto definido como $\mathbf{S} = \{\vec{u}$ tal que$\vec{v} = \sum_{i=1}^{n}(c_i \cdot \vec{v_i})\} \in \mathbf{V_n}$ es un subespacio vectorial de $\mathbf{V_n}$.
7. Se define la operación *producto interior* de dos vectores de la siguiente forma: $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n}(a_i \cdot b_i)$. Según esta operación diremos que:
   - $\vec{u}$ y $\vec{v}$ son ortogonales $\Leftrightarrow \vec{u} \cdot \vec{v} = 0$.
   - El **espacio vectorial dual** de $\mathbf{S}$ (subespacio vectorial de dimensión k de V_n) es el conjunto $\mathbf{S_d}$ cuyos elementos son todos los vectores ortogonales a los vectores $\vec{v} \in S$. Se dice que es un espacio dual de orden $(n - k)$.

A.2. Introducción a los códigos bloque lineales

Asumimos que la salida de una fuente de información es una secuencia de dígitos binarios "0" ó "1". En la *codificación de bloque*, esta secuencia de información binaria es segmentada en bloques de mensajes de una longitud fija; cada bloque de mensaje, llamado u, está formado por k dígitos de información. Hay un total de 2^k mensajes distintos.

El codificador, de acuerdo con ciertas reglas, transforma cada mensaje entrante u en una n–tupla binaria v con $n > k$. Esta n–tupla binaria v es lo que llamamos *palabra código* (o vector código) del mensaje u. Por lo tanto, para los 2^k posibles mensajes, hay 2^k palabras código. A este conjunto de 2^k palabras código se le llama **código bloque**. Para que un código bloque sea útil, las 2^k palabras código deben ser distintas. En consecuencia, tiene que haber una correspondencia uno a uno entre un mensaje u y su palabra código v.

Para un código con 2^k palabras código de longitud n, a menos que tenga una estructura especial, el aparato de codificación será prohibitivamente complejo si k y n son grandes, ya que tiene que almacenar las 2^k palabras código de longitud n en un diccionario. Por lo tanto, debemos restringir nuestra atención a códigos de bloque que pueden ser mecanizados de una manera práctica. Una estructura que se desea que tenga un código bloque es la **linealidad**. Con esta estructura, la complejidad de la codificación se reduce enormemente, como veremos más adelante.

A.2.1. Definición de código bloque lineal

A un código bloque de longitud n y 2^k palabras código se le llama *código lineal* (n, k) si y sólo si sus 2^k palabras código forman un subespacio k–dimensional del espacio vectorial de las n–tuplas sobre el campo GF(2).

De hecho, un código binario es lineal si y sólo si la suma módulo 2 de dos palabras código es también una palabra código. El bloque código dado en la siguiente tabla es un código lineal (7,4). Se puede comprobar fácilmente que la suma de dos palabras código es también otra palabra código.

Mensajes	**Palabras código**
(0 0 0 0)	(0 0 0 0 0 0 0)
(1 0 0 0)	(1 1 0 1 0 0 0)
(0 1 0 0)	(0 1 1 0 1 0 0)
(1 1 0 0)	(1 0 1 1 1 0 0)
(0 0 1 0)	(1 1 1 0 0 1 0)
(0 1 1 0)	(0 0 1 1 0 1 0)
(1 1 1 0)	(1 0 0 0 1 1 0)
(0 0 0 1)	(0 1 0 1 1 1 0)
(1 0 0 1)	(1 0 1 0 0 0 1)
(0 1 0 1)	(0 1 1 1 0 0 1)
(1 1 0 1)	(1 1 0 0 1 0 1)
(0 0 1 1)	(0 1 0 0 0 1 1)
(1 0 1 1)	(1 0 0 1 0 1 1)
(0 1 1 1)	(0 0 1 0 1 1 1)
(1 1 1 1)	(1 1 1 1 1 1 1)

Cuadro A.1: Ejemplo de código lineal (7,4)

Dado que un código lineal (n, k) C es un subespacio k–dimensional del espacio de vectores V_n de las n–tuplas binarias, es posible encontrar k palabras código linealmente independientes $(g_0, g_1, \ldots, g_{k-1})$ en C, de tal forma que cada palabra código v en C es una combinación lineal de esas k palabras código. Por tanto, $v = u_0 g_0 + u_1 g_1 + \ldots + u_{k-1} g_{k-1}$, donde $u_i = 0 \vee 1$ para $0 \leq i \leq k$.

Ahora vamos a colocar estas k palabras código linealmente independientes como las filas de una matriz $k \times n$:

$$G = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{pmatrix} = \begin{pmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{pmatrix}$$

Si $u = (u_0, u_1, \ldots, u_{k-1})$ es el mensaje que va a ser codificado, la palabra código correspondiente puede ser dada de la siguiente manera:

$$v = u \cdot G = u \cdot \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{pmatrix} = u_0 g_0 + u_1 g_1 + \ldots + u_{k-1} g_{k-1}$$

Evidentemente las filas de G generan el código lineal (n,k) C. Por esta razón a la matriz **G** se le llama **matriz generadora** de C. Nótese que cualquier conjunto de k palabras código linealmente independientes pueden ser usadas para formar una matriz generadora del código. Por lo tanto, un código lineal (n,k) está completamente definido por las k filas de la matriz generadora G. Como consecuencia de esto, el codificador sólo tiene que almacenar las k filas de G y formar una combinación lineal de estas k filas basada en el mensaje de entrada u.

Ejemplo: El código lineal (7,4) dado en la tabla A.2.1 tiene la siguiente matriz generadora:

$$G = \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Sea $u = (1101)$ el mensaje que hay que codificar, su palabra código correspondiente será: $v = 1 \cdot g_0 + 1 \cdot g_1 + 0 \cdot g_2 + 1 \cdot g_3 = (1101000) + (0110100) + (1010001) = (0001101)$

A.2.2. Forma sistemática

Una propiedad deseable en un código lineal es una estructura sistemática donde una palabra código se divide en dos partes: la parte del *mensaje* y la parte de *redundancia*. La parte del mensaje consiste en k bits de información inalterada (o mensaje); y la parte de redundancia consiste de $(n-k)$ bits de comprobación de paridad, los cuales son una suma lineal de los bits de información. A un código lineal de bloque con esta estructura se le llama **código lineal sistemático de bloque**. El código (7,4) dado en la tabla A.2.1 es un código sistemático, los cuatro bits que están más a la derecha de cada palabracódigo son idénticos a los bits correspondientes de información.

REDUNDANCIA	**MENSAJE**
$n-k$ dígitos	k dígitos

Un código lineal (n, k) sistemático queda completamente definido por una matriz G $k \times n$ de la siguiente forma:

$$G = \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{pmatrix} = \left(\begin{array}{cccc|ccccc} p_{0,0} & p_{0,1} & \dots & p_{0,n-k-1} & 1 & 0 & 0 & \dots & 0 \\ p_{1,0} & p_{1,1} & \dots & p_{1,n-k-1} & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \dots & p_{k-1,n-k-1} & 0 & 0 & 0 & \dots & 1 \end{array}\right)$$

Si llamamos I_k a la matriz identidad de dimensión k observamos que $G = [PI_k]$. Como $u = (u_0, u_1, \dots, u_{k-1})$ es el mensaje que hay que codificar, entonces la palabra código correspondiente es $v = (v_0, v_1, \dots, v_{n-1}) = (u_0, u_1, \dots, u_{k-1})G$.

Por tanto, podemos decir que las componentes de v son:

- $v_{n-k+i} = u_i$ con $0 \leq i < k$.
- $v_j = u_0 p_{0,j} + u_1 p_{1j} + \dots + u_{k-1} p_{k-1,j}$ con $0 \leq j < (n-k)$.

Esto nos demuestra que los k primeros dígitos por la derecha de una palabra código v son idénticos a los dígitos información $u_0, u_1, \dots, u_{k-1}$ que hay que codificar, y que los $n-k$ dígitos de redundancia que están a la izquierda son sumas lineales de los de información.

Ejemplo: La matriz G dada en el ejemplo anterior de la página 57 está en forma sistemática. Si $u = (u_0, u_1, u_2, u_3)$ es el mensaje que hay que codificar, y $v = (v_0, v_1, v_2, v_3, v_4, v_5, v_6)$ es la correspondiente palabra código, entonces:

$$v = u \cdot \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiplicando las matrices obtenemos los dígitos de la palabra código v:

$$\begin{aligned} v_6 &= u_3 \\ v_5 &= u_2 \\ v_4 &= u_1 \\ v_3 &= u_0 \\ v_2 &= u_1 + u_2 + u_3 \\ v_1 &= u_0 + u_1 + u_2 \\ v_0 &= u_0 + u_2 + u_3 \end{aligned}$$

Por lo tanto, la palabra código correspondiente a (1011) es (1001011).

A.2.3. Matriz de comprobación de paridad

Existe otra matriz útil asociada con cada código lineal de bloque. Para cada matriz **G** de dimensiones $k \times n$ con k filas linealmente independientes, existe una matriz **H** de dimensiones $(n-k) \times n$ con $n-k$ filas linealmente independientes de tal manera que cada vector en el espacio de las filas de **G** es ortogonal a las filas de **H**. Además, cada vector que es ortogonal a las filas de **H** está en el espacio de las filas de **G**. Como consecuencia de esto podemos describir el código lineal (n, k) generado por **G** de otra forma alternativa que a continuación se detalla.

Una n–tupla v es una palabra código en el código generado por G si y sólo si $v\mathbf{H}^T = 0$. Esta matriz **H** es la **matriz de comprobación de paridad** del código. Las 2^{n-k} combinaciones lineales de las filas de la matriz **H** forman un código lineal C_d $(n, n-k)$. Este código es el espacio nulo del código lineal C (n, k) generado por la matriz **G**, esto significa que para cualquier $\forall v \in C \wedge \forall w \in C_d, vw = 0$. C_d es el **código dual** de C. Por lo tanto, una matriz de comprobación de paridad de un código lineal C es la matriz generadora de su código dual.

Si la matriz generadora de un código lineal (n, k) está en la forma sistemática, la matriz de comprobación de paridad tiene la siguiente forma:

$$H = [I_{n-k}P^T] = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ 0 & 1 & 0 & \dots & 0 & p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{pmatrix}$$

Donde P^T es la traspuesta de la matriz P. Sea h_j la j–ésima fila de H. Podemos comprobar que el producto de la i–ésima fila de G definida en la página 58 y la j–ésima fila de H anterior es $g_i h_j = p_{ij} + p_{ij} = 0$ con $0 \le j < (n-k)$.

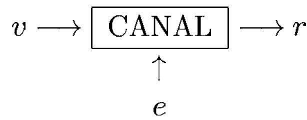
Esto implica que $\mathbf{GH^T} = \mathbf{0}$. Además, las $n-k$ filas de **H** son linealmente independientes. Por lo tanto, la matriz **H** es una matriz de comprobación de paridad del código lineal generado por **G**.

A.3. Síndrome y detección de errores

Consideramos un código lineal (n, k) con su matriz generadora G y su matriz de comprobación de paridad H. Sea v una palabra código que se transmite en un canal ruidoso[1], y **r** es el vector recibido a la salida del canal. Debido a que el canal es ruidoso, **r** puede ser distinto de **v**.

El vector suma de **r** y **v** es **e**; **e** es una n–tupla tal que $e_i = 1$ si $r_i \neq v_i$, y $e_i = 0$ si $r_i = v_i$. A esta n–tupla se le llama **vector de error**. Los 1's que aparecen en **e** son errores de transmisión producidos porque el canal es ruidoso.

[1] Un canal es ruidoso cuando la información de entrada puede sufrir alguna modificación resultando una salida distinta a la esperada



El receptor recibe **r**, que es la suma de la palabra código transmitida y el vector error. Cuando recibe **r**, el decodificador debe determinar si contiene errores de transmisión. Si se detectan errores, el decodificador intentará corregirlos (FEC) o pedirá una retransmisión (ARQ).

Cuando se recibe **r**, el decodificador calcula la siguiente (n–k)–tupla: $\mathbf{s} = rH^T = (s_0, s_1, \ldots, s_{n-k-1})$, esta (n–k)–tupla es el **síndrome** de **r**.

Podemos afirmar que $\mathbf{s} = 0$ si y sólo si r es una palabra código (el receptor acepta r como la palabra código transmitida), y $\mathbf{s} \neq 0$ si y sólo si r no es una palabra código (el receptor detecta la presencia de al menos un error). Pero es posible que los errores no sean detectables, es decir, que r contenga errores pero $\mathbf{s} = 0$. Esto sucede cuando el vector de error es idéntico a una palabra código no nula. En este caso, r es la suma de dos palabras código y por lo tanto el síndrome es igual a cero. Estos errores son **errores indetectables**. Como hay $2^k - 1$ palabras código no nulas, hay $2^k - 1$ errores indetectables.

Ejemplo: Vamos a calcular el síndrome del código cuya matriz de comprobación de paridad es la siguiente:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Si $\mathbf{r} = (r_0, r_1, r_2, r_3, r_4, r_5, r_6)$ es el vector recibido, el síndrome se calcula de la siguiente manera:

$$\mathbf{s} = r \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Con lo cual los dígitos que componen el síndrome son:

$$s_0 = r_0 + r_3 + r_5 + r_6$$
$$s_1 = r_1 + r_3 + r_5 + r_6$$
$$s_2 = r_2 + r_4 + r_5 + r_6$$

Como se ha visto, el síndrome depende sólo del vector de error, y no de la palabra código transmitida: $s = rH^T = (v+e)H^T = vH^T + eH^T$, como $vH^T = 0$ obtenemos la siguiente expresión:

$$\mathbf{s} = \mathbf{eH^T}$$

Llegado a este punto, uno puede pensar que cualquier esquema de corrección de errores consiste en resolver las $n-k$ ecuaciones lineales dadas por la expresión anterior. Una vez que se encuentra el error, se considera que el vector $\mathbf{r}+\mathbf{e}$ es la palabra código transmitida. Desgraciadamente, determinar cuál es el verdadero vector de error no es una cuestión sencilla. Esto es debido a que las $n-k$ ecuaciones lineales no tienen una solución única, sino que tienen 2^k soluciones. En otras palabras, hay 2^k patrones de error que dan el mismo síndrome, y el verdadero error que se ha producido es únicamente uno de ellos.

Para minimizar la probabilidad de error, se elige como vector de error al patrón de error más probable. Si el canal es BSC[2], el vector más probable es el que tiene el mayor número de ceros.

Ejemplo: Consideramos el código $(7,4)$ dado en el ejemplo anterior de la página 60. Si $v = (1001011)$ es la palabra código transmitida y $r = (1001001)$ es el vector recibido. El receptor calcula el síndrome: $s = rH^T = (111)$.

Una vez hecho esto, el receptor intenta determinar el verdadero vector de error $\mathbf{e}$. Los dígitos de error están relacionados con los del síndrome por las siguientes ecuaciones lineales:

$$\begin{aligned} 1 &= e_0 + e_3 + e_5 + e_6 \\ 1 &= e_1 + e_3 + e_4 + e_5 \\ 1 &= e_2 + e_4 + e_5 + e_6 \end{aligned}$$

Hay $2^4 = 16$ patrones de error que satisfacen las ecuaciones de arriba. Son:

(0000010)	(1010011)
(1101010)	(0111011)
(0110110)	(1100111)
(1011110)	(0001111)
(1110000)	(0100001)
(0011000)	(1001001)
(1000100)	(0010101)
(0101100)	(1111101)

El vector de error $e = (0000010)$ es el que tiene el menor número de componentes distintas de cero. Si el canal es BSC, $e = (0000010)$ es el vector de error más probable

[2]BSC (binary simmetric channel): Es un canal ruidoso en el que la probabilidad de que un 1 se transforme en 0 es igual a la probabilidad de que un 0 se transforme en 1. Además, si la probabilidad de que ocurra un error es p, la posibilidad de que se produzcan dos errores es p^2.

de los que satisfacen las ecuaciones anteriores. Si tomamos $e = (0000010)$ como el verdadero vector de error, el receptor decodifica el vector $\mathbf{r}$ en la siguiente palabra código: $v^* = r + e = (1001001) + (0000010) = (1001011)$. Como se puede apreciar, v^* es la palabra código transmitida. Por lo tanto, el receptor ha hecho una decodificación correcta.

A.4. Distancia mínima de un código

Tomamos una n–tupla $v = (v_0, v_1, \ldots, v_{n-1})$. El **peso Hamming** (o simplemente peso) de v, que se denota como $w(v)$, se define como el número de componentes distintas de cero de v. Por ejemplo, el peso de $v = (1001011)$ es 4.

Sean v y w dos n–tuplas, las **distancia Hamming** (o simplemente distancia) entre v y w, que se denota como $d(v, w)$, se define como el número de dígitos en el mismo sitio que tienen diferentes. Por ejemplo, la distancia Hamming entre $v = (1001011)$ y $w = (0100011)$ es 3, tienen diferentes las posiciones cero, uno y tres.

La distancia Hamming es una función métrica que satisface la *desigualdad triangular*. Es decir, sean v, w y x tres n–tuplas, entonces $d(u, w) + d(w, x) \geq d(v, x)$.

De todo esto se deduce que la distancia Hamming entre dos n–tuplas, v y w, es igual al peso Hamming de la suma de v y w, esto es, $d(v, w) = w(v + w)$. Por ejemplo, la distancia Hamming entre $v = (10001011)$ y $w = (1110010)$ es 4, y el peso de $v + w = (0111001)$ es también 4.

Dado un código bloque C, se puede calcular la distancia Hamming entre cualquiera dos palabras código distintas. La **distancia mínima** de $C(d_{min})$ se define como $d_{min} = min\{d(v, w) : v, w \in C, v \neq w\}$.

Sea C un código lineal, la suma de dos vectores es también un vector código. Por lo tanto, la distancia Hamming entre dos vectores código en C es igual al peso Hamming de un tercer vector código en C. De esto obtenemos que:

$$d_{min} = min\{d(v, w) : v, w \in C, v \neq w\} = min\{w(x) : x \in C, x \neq 0\} = w_{min}$$

w_{min} es el **peso mínimo** del código lineal C. De lo cual obtenemos el siguiente teorema:

Teorema 1 *La distancia mínima de un código lineal bloque es igual al mínimo peso de sus palabras distintas de cero.*

Teorema 2 *Sea C un código lineal (n, k) con su matriz de comprobación de paridad $\mathbf{H}$. Para cada vector código de peso Hamming 1, existen 1 columnas de $\mathbf{H}$ tales que el vector suma de esas columnas es igual al vector cero. Recíprocamente, si existen 1 columnas de $\mathbf{H}$ cuyo vector suma es el vector cero, existe un vector código con peso Hamming 1 en C.*

Corolario 1 *Sea C un código bloque lineal cuya matriz de comprobación de paridad es* $\mathbf{H}$*. Si $d-1$ o menos columnas de H suman 0, el código tiene un peso mínimo de por lo menos d.*

Corolario 1 *Sea C un código bloque lineal cuya matriz de comprobación de paridad es* $\mathbf{H}$*. El peso mínimo (o la mínima distancia) de C es igual al menor número de columnas de* $\mathbf{H}$ *que suman 0.*

Ejemplo: Vamos a considerar el código lineal dado en la tabla de la página 56. Su matriz de comprobación de paridad es la siguiente:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Se ve que todas las columnas de $\mathbf{H}$ son distintas de cero y que ninguna de ellas es igual a otra. Por consiguiente, no hay dos columnas que sumen 0. Entonces, el peso mínimo de este código es por lo menos 3. Por otro lado, las columnas cero, dos y seis suman 0. Por lo tanto, el peso mínimo del código es 3. Y, aplicando el teorema 1, concluimos que la distancia mínima es 3.

Los corolarios 1 y 1 se utilizan generalmente para determinar la distancia mínima o establecer un límite inferior en la distancia mínima de un código lineal de bloque.

A.5. Propiedades de detección y corrección de errores de un código bloque

A.5.1. Propiedades detectoras

Cuando se transmite un vector código $\mathbf{v}$ por un canal ruidiso, un vector de error con 1 errores provoca que el vector recibido $\mathbf{r}$ sea diferente del vector $\mathbf{v}$ en 1 posiciones ($d(v,r) = 1$). Si la distancia mínima del código C es d_{min}, cada pareja de dos vectores de C tienen al menos d_{min} posiciones distintas. Por lo tanto, cualquier vector de error con $d_{min}-1$ o menos errores da un vector $\mathbf{r}$ que no es una palabra código de C. Cuando el receptor detecta que el vector recibido no es una palabra código de C, el error ha sido detectado.

Por lo tanto, un código bloque con distancia mínima d_{min} es capaz de detectar todos los patrones de error de $d_{min} - 1$ o menos errores.

Un código bloque detecta todos los patrones de error de $d_{min} - 1$ o menos errores, pero también detecta una gran fracción de los patrones de error con d_{min} o más errores. En realidad, un código lineal (n, k) es capaz de detectar $2^n - 2^k$ patrones de error de longitud n.

De entre los $2^n - 1$ patrones de error distintos de cero, hay $2^k - 1$ patrones de error que son idénticos a las $2^k - 1$ palabras código. Si sucede cualquiera de esos $2^k - 1$ patrones de error, la palabra código transmitida $\mathbf{v}$ se transforma en otra palabra código $\mathbf{w}$. Por lo tanto, se recibe $\mathbf{w}$ y su síndrome es cero. En este caso, el decodificador acepta $\mathbf{w}$ como la palabra transmitida y realiza una decodificación incorrecta. De todo esto concluimos que hay $2^k - 1$ **patrones de error que son indetectables.**

Hay $2^n - 2^k$ patrones de error detectables. Para n suficientemente grande, $2^k - 1$ es en general mucho más pequeño que 2^n. Por lo tanto, **sólo una pequeña fracción de errores pasa por el decodificador sin ser detectados**.

Probabilidad de no detectar un error Sea C un código lineal (n, k). Sea A_i el número de vectores del código de peso i. A los números $A_0, A_1, \ldots, A_n$ se les llama distribución del peso de C. Si C se usa sólo para detección de errores en un BSC, la probabilidad de que el decodificador falle al detectar errores se puede calcular como la distribución del peso de C.

Sea p la probabilidad de error de bit del canal, si denotamos la probabilidad de no detectar un error como $P_U(E)$, entonces:

$$P_U(E) = \sum_{i=0}^{n} A_i p^i (1-p)^{n-i}$$

Además, si la distancia mínima de C es d_{min} entonces desde A_1 hasta $A_{d_{min}-1}$ son cero.

Ejemplo: Vamos a considerar el código dado en la tabla de la página 56. La distribución del peso de este código es $A_0 = 1, A_1 = 2, A_2 = 0, A_3 = A_4 = 7, A_5 = A_6 = 0, A_7 = 1$. La probabilidad de que no se detecte un error es:

$$P_U(E) = 7p^3(1-p)^4 + 7p^4(1-p)^3 + p^7$$

Si $p = 10^{-2}$, esta probabilidad es aproximadamente 7×10^{-6}. En otras palabras, si un millón de palabras se transmiten por un canal BSC con $p = 10^{-2}$, por término medio siete palabras código erróneas pasarán por el decodificador sin ser detectados.

A.5.2. Propiedades correctoras

Si un código bloque C con distancia mínima d_{min} se usa para corregir errores aleatorios, es conveniente saber cuántos errores es capaz de corregir dicho código.

La distancia mínima es par o impar. Por lo tanto, podemos elegir un entero t tal que: $2t + 1 \leq d_{min} \leq 2t + 2$.

Ahora vamos a probar que el código C es capaz de corregir todos los patrones de error de t o menos errores. Sean $\mathbf{v}$ y $\mathbf{r}$ el vector transmitido y recibido respectivamente.

Sea **w** cualquier otro vector código de C. La distancia Hamming entre **v**, **r** y **w** satisface la desigualdad triangular: $d(v, r) + d(w, r) \geq d(v, w)$.

Supongamos que un patrón de error de t' sucede durante la transmisión de **v**. Entonces, el vector recibido tiene t' posiciones distintas de **v** y $d(v, r) = t'$. Puesto que **v** y **w** son vectores código de C, tenemos que $d(v, w) \geq d_{min} \geq 2t + 1$.

Obtenemos la siguiente desigualdad: $d(w, r) \geq 2t + 1 - t'$. Si $t' \leq t$, entonces $d(w, r) > 1$.

La desigualdad anterior dice que si un patrón de t o menos errores tiene lugar, el vector recibido **r** está más cerca del vector transmitido **v** que a cualquier otro vector **w** en C. Basándose en el *esquema de decodificación de máxima similitud*, **r** es decodificado en **v**, que es el vector que se transmitió. Se ha realizado una decodificación correcta y por lo tanto una corrección de errores.

Por otra parte, el código no es capaz de corregir todos los patrones de error de 1 errores con $1 > t$. Esto sucede porque por lo menor hay un caso en el cual un patrón de 1 errores da lugar en recepción un vector **r** que está más cerca de un vector código incorrecto que al vector transmitido. Veamos esto con dos vectores código **v** y **w**: $d(v, w) = d_{min}$.

Sean e_1 y e_2 dos patrones de error que satisfacen las siguientes condiciones:

1. $e_1 + e_2 = v + w$.

2. e_1 y e_2 no tienen componentes distintas de cero en las mismas posiciones.

Obviamente, tenemos que $w(e_1) + w(e_2) = w(v + w) = d(v, w) = d_{min}$.

Ahora suponemos que **v** se transmite y es corrompida por e_1. El vector recibido es $r = v + e_1$.

La distancia Hamming entre **v** y **w** es $d(v, r) = w(v + r) = w(e_1)$.

La distancia Hamming entre **w** y **r** es $d(w, r) = w(v+r) = w(w+v+e_1) = w(e_2)$.

Ahora suponemos que el patrón de error e_1 contiene más de t errores ($w(e_1) > t$). Puesto que $2t + 1 \leq d_{min} \leq 2t + 2$, obtenemos que $w(e_2) \leq t + 1$.

Combinando las tres últimas expresiones y usando el hecho de que $w(e_1) < t$, obtenemos la siguiente desigualdad: $d(v, r) \geq d(w, r)$.

Esta desigualdad dice que existe un patrón de error de 1 ($1 > t$) errores que da lugar a un vector recibido que está más cerca de un vector incorrecto que del vector transmitido. Basándose en el esquema de decodificación de máxima similitud se puede producir una decodificación incorrecta.

Para resumir, un código bloque con distancia mínima d_{min} garantiza que **corrige todos los patrones de error de** $t = [(d_{min} - 1)/2]$ **o menos errores**.

La distancia mínima del código dado en la tabla de la página 56 es 3, con lo que $t = 1$. Por lo tanto, es capaz de corregir cualquier patrón de error con un sólo error.

Un código bloque con capacidad de corregir t errores aleatorios generalmente es capaz de corregir bastantes patrones de error $t + 1$ errores o más. Un código lineal

(n,k) con capacidad de corregir t errores, es capaz de corregir un total de $2^n - 2^k$ patrones de error.

Probabilidad de decodificar de forma incorrecta Si usamos un código bloque, que es capaz de corregir t errores, únicamente para corregir en un canal BSC con probabilidad de error p, la probabilidad de que el decodificador realice una decodificación incorrecta es:

$$P(E) \leq \sum_{i=t+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}$$

A.6. Matriz típica y decodificación de síndrome

A.6.1. Partición como método de decodificación

Sea $C(n,k)$ un código lineal y v_1, v_2, $\dots$, v_k los vectores código que componen C. Si enviamos un vector código cualquiera a través de un canal ruidoso, el vector símbolo recibido estará dentro de las n–tuplas $(U_1, \dots, U_n)$.

Todo sistema de decodificación divide el total de vectores en 2^k subconjuntos disjuntos $D_1, D_2, \dots, D_k$. En el método que veremos, cada palabra código v_j se encuentra en un y sólo un subconjunto D_j. Si el vector recibido r_j se encuentra en ese subconjunto D_j, se decodificará como v_j.

A.6.2. Construcción de la matriz típica

Empezamos colocando en la primera fila los 2^k vectores del código con el vector nulo $v_1 = (0, 0, \dots, 0)$ como primer elemento. De las $2^n - 2^k$ n–tuplas restantes se toma e_2 como vector de error y se sitúa debajo del v_1, obteniendo el resto de elementos de la fila como suma de e_2 con los vectores código de la fila primera poniendo $e_2 + v_j$ bajo el vector v_j. Para la tercera fila se coge otro que no se encuentre en las dos primeras y se le llama e_3. El resto de la fila lo forman las palabras $e_3 + v_j$ que se sitúa bajo v_j. Así continuaremos el proceso hasta que todas las n–tuplas sean usadas quedando la matriz:

$$\begin{array}{cccccc}
v_1 = 0 & v_2 & \dots & v_i & \dots & v_2k \\
e_2 & e_2 + v_2 & \dots & e_2 + v_i & \dots & e_2 + v_2k \\
e_3 & e_3 + v_2 & \dots & e_3 + v_i & \dots & e_3 + v_2k \\
\vdots & & & & & \vdots \\
e_j & e_j + v_2 & \dots & e_j + v_i & \dots & e_j + v_2k \\
\vdots & & & & & \vdots \\
e_{2n-k} & e_{2n-k} + v_2 & \dots & e_{2n-k} + v_i & \dots & e_{2n-k} + v_2k
\end{array}$$

Teorema 3 *En una fila de una matriz típica no existen dos n–tuplas iguales. Cada n–tupla aparece en una y sólo una fila.*

Demostración

La primera parte del teorema se deduce del hecho de que todos los vectores código de C son distintos. Si suponemos que dos n–tuplas de la fila k–ésima son iguales, es decir, $e_k + v_i = e_k + v_j$ con i distinto de j, entonces tendríamos $v_i = v_j$, lo cual es imposible. Por tanto dos n–tuplas de una misma fila no pueden ser iguales.

De la regla de construcción de la matriz típica se sigue que toda n–tupla aparece al menos una vez. Supongamos ahora que una n–tupla apareciera en las filas t y m con $t < m$. Entonces esta n–tupla sería igual a $e_1 + v_s$ para algun s y también $e_m + v_j$ para algún j. Pero entonces $e_t + v_s = e_m + v_j$ y, despejando, $e_m = e_t + (v_s + j) = e_t + v_k$ (pues la suma de dos palabras código es otra palabra código). Esta ecuación significa que el vector e_m estaba también en la fila t, lo que contradice la regla de construcción de la matriz típica, pues el líder de un coconjunto ha de ser un vector que no aparezca todavía en la matriz. Esto demuestra la segunda parte del teorema.

Se deduce pues que hay e^{n-k} filas distintas, cada una con 2^k elementos distintos. Estas filas son denominadas coconjuntos del código C y la primera n–tupla de cada coconjunto se denomina líder del coconjunto. Cualquier elemento del coconjunto ha de ser líder del mismo, permutándose los elementos, pero sin cambiar el coconjunto.

Una matriz típica de un código lineal C(n,k) consta de 2^k columnas disjuntas, cada una con 2^{n-k} n–tuplas siendo la más alta un vector código de C. Si denominamos D_j a la columna j–ésima de la matriz típica, entonces:

$D_j = \{v_j, e_2 + v_j, e_3 + v_j, \ldots, e_{2^{n-k}} + v_j\}$, donde v_j es un vector código de C y $e_2, \ldots, e_{2^{n-k}}$ son los líderes de los coconjuntos. Las 2^k columnas disjuntas $D_1, D_2, \ldots, D_{2^k}$ pueden ser usadas para decodificar C.

Teorema 4 *Todo código lineal de bloque es capaz de corregir 2^{n-k} patrones de error.*

Demostración

Enviamos un vector código v_j a través de un canal ruidoso y recibimos un vector r. Este vector será correctamente decodificado si se encuentra en el coconjunto D_j, y para ello el vector ruido del canal ha se ser líder del coconjunto. Si no lo es, aparecerá en la matriz bajo la columna t cuya palabra código asociada es el vector v_t de tal forma que si llamamos x al ruido del canal será $x = e_1 + v_t$. Por tanto, al enviar v_j tendremos que $r = v_j + x = v_j + v_t + e_1 = v_n + e_1$, siendo v_n vector código por ser suma de palabras código. Al decodificar obtendremos erróneamente v_n en vez de v_j.

Por tanto, la decodificación es correcta si el vector del canal es líder del coconjunto. Por ello, a estos 2^{n-k} vectores líderes de los coconjuntos se les denomina ***patrones de error*** *y son los únicos que pueden ser corregidos correctamente en el código C(n,k).*

Para minimizar la probabilidad de error hemos de coger como líderes de los coconjuntos los patrones de error más probables, que en un BSC son los de menos peso. Por consiguiente, al formar la matriz típica, deben elegirse los líderes de lo coconuntos como un vector del menor peso entre los que aún no han sido incluidos. Así, el líder tiene mínimo peso en su coconjunto, siendo la decodificación basada en la matriz típica una **decodificación de mínima distancia**.

Demostración

Esta es la demostración del mínimo peso del líder del coconjunto:

Sea **r** *el vector recibido. Supongamos que r se encuentra en la columna iésima D_1 y el coconjunto l–ésimo de la matriz típica. Entonces r se deocodifica como el vector v_1. Como $r = e_1 + v_i$, la distancia entre r y v_i es $d(r, v_i) = w(r + v_i) = w(e_1 + v_i + v_i) = e(e_1)$.*

Consideremos ahora la distancia entre r y otro vector código v_j: $d(r, v_j) = w(r + v_j) = w(e_1 + v_i + v_j)$. Como v_i y v_j son dos vectores diferentes, $v_i + v_j$ es un vector código no nulo v_s. Así $d(r, v_s) = w(e_1 + v_s)$ y como e_1 y $e_1 + v_s$ están en el mismo coconjunto cumpliéndose $w(e_1) \leq w(e_1 + v_s)$, se sigue: $\mathbf{d(r, v_i)} \leq \mathbf{d(r, v_j)}$.

Un código lineal (n,k) es capaz de detectar $2^n - 2^k$ patrones de error y de corregir $2^(n - k)$. Para n muy grande el error de decodificación es mucho mayor que la probabilidad de error sin detectar.

Teorema 5 *Para un código lineal C(n,k) con distancia mínima d_{min}, todas las n–tuplas de peso $t = [(d_{min} - 1)/2]$ o menor pueden ser usadas como líderes de coconjuntos de una matriz típica de C. Si todas las n–tuplas de peso t o menor son usadas, hay al menos una n–tupla de peso $t + 1$ que no puede ser usada como líder de coconjunto.*

Este teorema nos confirma que un código lineal C(n,k) con distancia mínima d_{min}, es capaz de corregir todos los patrones de error de $[(d_{min} - 1)/2]$ o menos errores, pero no es capaz de corregir todos los patrones de error de peso $t + 1$.

Teorema 6 *Todas las 2^k n–tuplas de un coconjunto tienen el mismo síndrome. Los síndromes para diferentes coconjuntos son distintos.*

Sabemos que el síndrome de una n–tupla es una (n-k)–tupla y hay $2^(n-k)$ (n-k)–tuplas. Usando la correspondencia uno a uno que existe entre un líder de coconjunto y un síndrome podemos formar una tabla de decodificación, que es un método mucho más simple para usar la matriz típica. La tabla contiene los $2^(n - k)$ líderes de coconjuntos y sus correspondientes síndromes. La decodificación consta de tres pasos:

1. Calcular el síndrome del vector recibido r, multiplicándolo por la matriz H traspuesta.

2. Localizar el líder del coconjunto cuyo síndrome es igual al hallado. Entonces dicho líder (e_i) se considera el patrón de error causado por el canal.

3. Decodificar el vector recibido r en el vector código $v = r + e_i$.

A este esquema de decodificación se le denomina **decodificación de síndrome**. El resultado de aplicar este método a un código lineal C(n,k) es un retraso en decodificación mínimo y una probabilidad de error mínima. No obstante para $n-k$ grandes, el esquema se vuelve poco práctico.

Ejemplo: Consideremos el código lineal C(7,4). La matriz de comprobación de paridad es:

$$H = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

El código tiene $2^3 = 8$ coconjuntos y, así hay ocho patrones de corrección de errores (incluyendo el vector nulo). Como la distancia mínima del código es 3, es capaz de corregir todos los patrones de error de peso 1 ó 0. Aquí pueden ser usadas como líderes de coconjuntos todas las 7–tuplas de peso 1 ó 0. Hay 8 vectores que cumplen esto. En este ejemplo, para el código lineal (7,4), el número de patrones de error que se pueden corregir garantizado por la distancia mínima es igual al total de patrones de error corregibles. Los patrones de error corregibles y sus correspondientes síndromes vienen dados en la siguiente tabla:

Síndrome	Líderes de coconjuntos
(100)	(1000000)
(010)	(0100000)
(001)	(0010000)
(110)	(0001000)
(011)	(0000100)
(111)	(0000010)
(101)	(0000001)

Supongamos que se transmite el vector código $v = (1001011)$ y se recibe $r = (1001111)$. Para decodificar r calculamos su síndrome:

$$s = (1001111) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} = (011)$$

Si observamos la tabla averiguamos que (011) es el síndrome del líder de coconjunto $e = (0000100)$. Así, se asume que el patrón de error causado por el canal es (0000100), y r se decodifica en $v^* = r + e = (1001111) + (0000100) = (1001011)$ que coincide con el vector transmitido. La decodificación será correcta si el patrón de error causado por el canal es un líder de cconjunto.

Ahora supongamos que se transmite $v = (0000000)$ y se recibe $r = (1000100)$. Vemos que se han producido dos errores durante la transmisión de v. El patrón de rror no se puede corregir y causará un error de decodificación. Cuando se recibe r, el receptor calcula el síndrome $s = rH^T = (111)$.

De la tabla de decodificación encontramos que el líder del coconjunto $e = (0000010)$ corresponde al síndrome $s = (111)$. Como resultado, r se decodifica en el vector código en $v^* = r + e = (1000100) + (0000010) = (1000110)$.

Como v^* no es el vector código transmitido, se comoete un error de decodificación. Usando pues la tabla anterior se pueden corregir errores simples sobre un bloque de 7 dígitos. Cuando ocurren dos o más errores se comete un error de decodificación.

A.7. Códigos Hamming

Los códigos Hamming fueron la primera clase de códigos ideados para corrección de errores. Estos códigos y sus variaciones han sido ámpliamente usados para control de errores en comunicación digital y en sistemas de almacenaje de información.

Para cualquier entero positivo $m \geq 3$ existe un código Hamming con los siguientes parámetros:

Longitud del código	$n = 2^m - 1$
Número de símbolos de información	$k = 2^m - m - 1$
Número de símbolos de comprobación de paridad	$n - k = m$
Capacidad de corrección de errores	$t = 1(d_{min} = 3)$

La matriz de comprobación de paridad H de este código consta de todas las m–tuplas no nulas como columnas. En forma sistemática, las columnas de H están ordenadas de la siguiente forma: $H = [I_m Q]$, donde I_m es una matriz identidad $m \times m$ y la submatriz Q consta de $2^m - m - 1$ columnas, las cuales son las m–tuplas de peso 2 o más. Estas columnas pueden ser colocadas en cualquier orden sin afectar a la propiedad de distancia y distribución de peso del código. En forma sistemática, la matriz generadora del código es $G = [Q^T I_{2^m-m-1}]$, donde Q^T es la traspuesta de Q y I_{2^m-m-1} es una matriz identidad de orden $2^m - m - 1$.

Como las columnas de H son no nulas y distintas, dos columnas no pueden sumar cero. Se sigue que la mínima distancia de un código Hamming es, al menos, 3. Como H consta de todas las m–tuplas no nulas como columnas, el vector suma de dos columnas cualesquiera, h_i y h_j, debe ser también una columna de H, h_1. Así $h_i + h_j + h_1 = 0$.

De aquí se sigue que la mínima distancia de un código Hamming es exactamente 3. Así, el código es capaz de corregir todos los patrones de error con un error simple o de detectar todos los patrones de error de dos errores o menos.

Si formamos una matriz típica para el código Hamming de longitud $2^m - 1$, se pueden usar todas las $(2^m - 1)$–tuplas de peso 1 como líderes de coconjunto. El número de $(2^m - 1)$–tuplas de peso 1 es $2^m - 1$. Como $n - k = m$, el código tiene 2^m coconjuntos. Así, el vector 0 y las $(2^m - 1)$–tuplas de peso 1 forman todos los líderes de los coconjuntos de la matriz típica. Esto nos dice que un código Hamming corrige solamente los patrones de error simple y ningún otro. Esta es una construcción muy interesante. Un código corrector de errores de peso t se llama **código perfecto** si su matriz típica tiene todos los patrones de error de peso t o menor y ninguno otro como líderes de coconjunto. Así, los códigos Hamming forman una clase de códigos perfectos de corrección de errores simples.

Podemos borrar cualesquiera l columnas de la matriz de comprobación de paridad H de un código Hamming. Esto da como resultado una matriz H' de orden $m \times (2^m - l - 1)$. Usando H' como matriz de comprobación de paridad, obtenemos un código Hamming recortado por los siguientes parámetros:

Longitud del código	$n = 2^m - l - 1$
Número de símbolos de información	$k = 2^m - m - l - 1$
Número de símbolos de comprobación de paridad	$n - k = m$
Capacidad de corrección de errores	$d_{min} = 3$

Si borramos las columnas apropiadas de H, obtenemos un código Hamming recortado de distancia mínima 4. El código Hamming recortado de distancia 4 puede ser usado para corregir los patrones de error simple y simultáneamente detectar los patrones de error doble. Cuando un error simple ocurre durante la transmision de un vector codigo, el síndrome resultante es distinto de cero y contiene un número par de unos. No obstante, cuando ocurre un error doble el síndrome es también distinto de cero, pero consta de un número impar de unos. Basándonos en estos hechos, la decodificación puede ser realizada de la siguiente manera:

1. Si el síndrome s es cero, asumimos que el error no existió.

2. Si s es distinto de cero y consta de un número par de unos, asumimos que el error ocurrido fue un error simple. El patrón del error simple que corresponde a s se suma al vector recibido para la corrección de errores.

3. Si s es distinto de cero y contiene un número impar de unos se ha detectado un patrón de error no corregible.

Apéndice B

Diseño de subcircuitos

B.1. Contador binario de 0 a 6

Nos va a hacer falta un contador que cuente 7 veces, o lo que es lo mismo, que cuente de 0 a 6. Para ello vamos a seguir una metodología de diseño de circuitos secuenciales. A continuación se muestra la "tabla del estado futuro" en la que se basará el contador.

Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	J_2	K_2	J_1	K_1	J_0	k_0
0	0	0	0	0	1	0	x	0	x	1	x
0	0	1	0	1	0	0	x	1	x	x	1
0	1	0	0	1	1	0	x	x	0	1	x
0	1	1	1	0	0	1	x	x	1	x	1
1	0	0	1	0	1	x	0	0	x	1	x
1	0	1	1	1	0	x	0	1	x	x	1
1	1	0	0	0	0	x	1	x	1	0	x
1	1	1	0	0	0	x	1	x	1	x	1

Se debe simplificar cada J y cada K en función de las entradas Q_2, Q_1 y Q_0. El resultado de esta simplificación es el siguiente:

$J_2 = Q_1 Q_0$
$K_2 = Q_1$
$J_1 = Q_0$
$K_1 = Q_2 + Q_0$
$J_0 = \overline{Q_2} + \overline{Q_0}$
$K_0 = 1$

B.2. Contador hasta 27

Ahora diseñaremos un contador que cuente de 0 hasta 27 (28 veces). Esta es su tabla de estado futuro:

Q_4	Q_3	Q_2	Q_1	Q_0	Q'_4	Q'_3	Q'_2	Q'_1	Q'_0	J_4	K_4	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	0	0	1	0	x	0	x	0	x	0	x	1	x
0	0	0	0	1	0	0	0	1	0	0	x	0	x	0	x	1	x	x	1
0	0	0	1	0	0	0	0	1	1	0	x	0	x	0	x	x	0	1	x
0	0	0	1	1	0	0	1	0	0	0	x	0	x	1	x	x	1	x	1
0	0	1	0	0	0	0	1	0	1	0	x	0	x	x	0	0	x	1	x
0	0	1	0	1	0	0	1	1	0	0	x	0	x	x	0	1	x	x	1
0	0	1	1	0	0	0	1	1	1	0	x	0	x	x	0	x	0	1	x
0	0	1	1	1	0	1	0	0	0	0	x	1	x	x	1	x	1	x	1
0	1	0	0	0	0	1	0	0	1	0	x	x	0	0	x	0	x	1	x
0	1	0	0	1	0	1	0	1	0	0	x	x	0	0	x	1	x	x	1
0	1	0	1	0	0	1	0	1	1	0	x	x	0	0	x	x	0	1	x
0	1	0	1	1	0	1	1	0	0	0	x	x	0	1	x	x	1	x	1
0	1	1	0	0	0	1	1	0	1	0	x	x	0	x	0	0	x	1	x
0	1	1	0	1	0	1	1	1	0	0	x	x	0	x	0	1	x	x	1
0	1	1	1	0	0	1	1	1	1	0	x	x	0	x	0	x	0	1	x
0	1	1	1	1	1	0	0	0	0	1	x	x	1	x	1	x	1	x	1
1	0	0	0	0	1	0	0	0	1	x	0	0	x	0	x	0	x	1	x
1	0	0	0	1	1	0	0	1	0	x	0	0	x	0	x	1	x	x	1
1	0	0	1	0	1	0	0	1	1	x	0	0	x	0	x	x	0	1	x
1	0	0	1	1	1	0	1	0	0	x	0	0	x	1	x	x	1	x	1
1	0	1	0	0	1	0	1	0	1	x	0	0	x	x	0	0	x	1	x
1	0	1	0	1	1	0	1	1	0	x	0	0	x	x	0	1	x	x	1
1	0	1	1	0	1	0	1	1	1	x	0	0	x	x	0	x	0	1	x
1	0	1	1	1	1	1	0	0	0	x	0	1	x	x	1	x	1	x	1
1	1	0	0	0	1	1	0	0	1	x	0	x	0	0	x	0	x	1	x
1	1	0	0	1	1	1	0	1	0	x	0	x	0	0	x	1	x	x	1
1	1	0	1	0	1	1	0	1	1	x	0	x	0	0	x	x	0	1	x
1	1	0	1	1	0	0	0	0	0	x	1	x	1	0	x	x	1	x	1
1	1	1	0	0	0	0	0	0	0	x	1	x	1	x	1	0	x	0	x
1	1	1	0	1	0	0	0	0	0	x	1	x	1	x	1	0	x	x	1
1	1	1	1	0	0	0	0	0	0	x	1	x	1	x	1	x	1	0	x
1	1	1	1	1	0	0	0	0	0	x	1	x	1	x	1	x	1	x	1

Estas serían las funciones lógicas:

$J_4 = Q_3Q_2Q_1Q_0$

$K_4 = Q_3Q_1Q_0 + Q_3Q_2$

$J_3 = Q_2Q_1Q_0$

$K_3 = Q_2Q_1Q_0 + Q_4Q_1Q_0 + Q_4Q_2$

$J_2 = \overline{Q_4}Q_1Q_0 + \overline{Q_3}Q_1Q_0$

$K_2 = Q_4Q_3 + Q_1Q_0$

$J_1 = \overline{Q_4}Q_0 + \overline{Q_3}Q_0 + \overline{Q_2}Q_0$

$K_1 = Q_4Q_3 + Q_0$

$J_0 = \overline{Q_4} + \overline{Q_3} + \overline{Q_2}$

$K_0 = 1$