

ASGN5 Design

notes:

m = message 4 bits	XOR for addition
c = hamming code 8 bits	OR for multiplication
G = vector-matrix multiplication	
e = error syndrome	

X	Y	AND(X,Y)	OR(X,Y)	NAND(X,Y)	NOR(X,Y)	XOR(X,Y)
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Hamming(8,4)
8 = Hamming Code
4 = message

PRELAB IS LATER IN DOCUMENT

Quicknotes*

Encode and Decode will both be in hamming.c

File is just a sequence of bytes,

Writeup: show error rate and stuff and entropy

Lookup tables, You don't need to search through them, should be indexed

If the value of error syndrome is 15, then just look at index 15 of the table. The error syn value corresponds to exact index

BitVector

Struct:

- `uint32_t length` // Length in bits
- `uint8_t *vector` // Array of bytes. It's `uint8_t` because we extended the last bit to be p3

16 bit vector

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Byte 0								Byte 1								

Setbit(n)

- `Byte = n/8`
 - Now we have to locate the specific bit inside of that byte.
- `Offset = n % 8`
 - `byte = Byte (OR) 1 << (offset)`
- One line: `v->vector[n/8] |= (1 << (n%8))`

Ex.

Set bit (14)

Byte = $14/8 = 1$ // Byte 1

Offset = $14 \% 8 = 6$ // Bit 14 is in byte 1 and bit 6 offset

To set the 6th bit

MSB	1	0	1	1		0	1	1	0	LSB
	0	0	0	0		0	0	0	1	Shift it 6 times
OR	0	0	0	1		0	0	0	0	
=	1	1	1	1		0	1	1	0	

To set a bit in a byte, shift a "1" to the index then bitwise OR operation

= byte w/ bit set

Bv_clr_bit(n):

- Get 1's in every position BUT the bit
- MSB 0111 0110 LSB, Shift to the position, Invert it = 1101 1111 AND it, to become 0101 0110.
- `v->vector[n/8] &= ~(1 << (n%8))`

Bv_get_bit(n):

- `v->vector[n/8] >> (n % 8) & 0x1`

MSB 0011 0010 LSB LS AND 0001 0000 0001 0000 Now shift right 4 back to position	MSB 0011 0010 LSB Right shift 4, 0000 0011 AND 0000 0001 = 0000 0001
--	---

XOR_BIT:

- MSB 0011 1100 LSB

MSB 0011 0010 LSB LS to position XOR 0010 0000 0001 1100	
--	--

4x8 Matrix = 32 bits.

4x4 Matrix = 16 bits.

Bc we know num of rows and columns, we always know the amount of bits.

Bit Matrix:

Bit Matrix serves as a wrapper over Bit Vector so most of these could be done with one line.

Struct:

Stores u32 rows

Store u32 Cols

Bitvector *vector

Bitmatrix = m rows, n columns

Rth row, Cth column

$R \cdot N + C$

Bit matrix Create:

Bm rows

Bm cols

Bm set bit (rows, cols)

Bm from data - bm to data

Needed bc we need to multiply bit vectors/matrices

We need a way to convert bytes to specific matrix

Msg = MSB 0000 1101 LSB // We need to convert this into a bit matrix

We can loop through bits, clear bit, set bit.

Bitmatrix *M = bm_from_data(msg, 4/(hamming 4/8scheme))

To convert into bit matrix:

Code = MSB 1110 0000 LSB

Bit Matrix 8C = bm_from_data(code, 8)

Hamming Encode

As arguments we receive the Bitmatrix G and a uint8_t msg.

The Bitmatrix G is a matrix that looks like this

The left side shows

$$C = m.g$$

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

What Hamming Encode is doing is receiving a msg (Which is one byte, also one character)

To encode a message we need to multiply $M \cdot g$

We need to follow the formula: $C = m.g$ to get the encoded message.

First: Convert the msg into a matrix

Second: Multiply the new matrix and G.

Third: Profit, convert that matrix back into binary and return it.

Hamming Decode:

Hamm decode is similar to encode but doing the exact opposite. To decode a message we need to multiply the code by the transposed matrix of G, which looks like :

This is the transposed version of the matrix used in encode, and we use the transposed matrix because the code is 8 bits.

When multiplying matrices together, $(A \cdot B)$. We need the A's columns to equal the B's rows. As the code is 8 bits, 8 columns, we can multiply that to the 8 rows of the transposed matrix.

Once we multiply it, we convert it into binary. (We get a 1x4 matrix) or 4 bit long binary. This is called the error rate

If the error rate is 0000, then there are no errors and we could return HAM_OK.

PRELAB:

1.

2.

Index	Error Syndrome	row in H ^t	a. Decode 1110 0011. We do this by converting the binary into a matrix.
0	0000	HAM_OK	$(1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1) \cdot \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
1	0001	4	
2	0010	5	
3	0011	HAM_ERR	

4	0100	6	<p>That brings us to the decoded matrix: (1 0 1 1)</p> <p>If we look at the lookup graph, we see that that this points to the second row. Now we require to flip the second element of the code to become (1 0 0 0 0 1 1 1). Now we can POSITIVELY say that the message is the first 4 bits of the code: 1100.</p> <p>b. Decode 1101 1000. Multiplied becomes (0101), which is HAM_ERR. We can not decode this one</p>
5	0101	HAM_ERR	
6	0110	HAM_ERR	
7	0111	3	
8	1000	7	
9	1001	HAM_ERR	
10	1010	HAM_ERR	
11	1011	2	
12	1100	HAM_ERR	
13	1101	1	
14	1110	0	
15	1111	HAM_ERR	

Hamming encode

Allocate memory to Generated Matrix

return Code = bm_to_data

Hamming Decoding, Convert code into bit matrix

Code = bm_from_data()

Error syndrome = $c \cdot H^t$

If $e == 0$

 No error return Ham ok

Else

 lookup(e)

Encoder Main()

getopt() loop

Take in two possible arguments. Infile and outfile.

Create generator Matrix G // This is going to be a bit matrix

Create the generator matrix by using a series of for loops to create the 4x8 matrix.

while fgetc != EOF

byte = MSB 1100 0110 LSB

msg1 = low nibble of byte

msg2 = high nibble of byte

code 1 = ham_encode (G, message to encode); // 8 bits = 1 byte

code 2 = ham_encode (G, msg2); // 8 bits = 1 byte

We can use fputc // Writing a character OUT to a file stream
and free memory later

Decoder Main()

getopt loop()

create H transpose (H^t)

while fgetc != EOF

When we're decoding, every byte is a code.

How much code do we need to convert back to a single byte of data? 2.

get code 1

get code 2

msg1= decode (H^t, code 1) // nibble

msg2 = decode(H^t, code 2) //Nibble

Now we have to combine them and pack them together using pack_byte.

Pack the nibbles into a byte.

fputc(packed nibbles)