Fabrice Kurmann
fkurmann@ucsc.edu
3 May, 2021

CSE 13S Spring 2021
Assignment 5: Hamming Codes
Design Document

Program Flow:

Encoder: Generate Hamming codes given input.
Accept command line options:
- -h - print out program info message
- -i - specify an infile other than stdin
- -o - specify an outfile for the encoded codes only other than stdout.

Decoder: Decode the Hamming codes made by the encoder.
Print statistics (bytes processed, uncorrected and corrected errors, error rate decimal to 6 digits of precision) to stderr.

Accept command line options:
- -h - print out program info message
- -i - specify an infile other than stdin
- -o - specify an outfile for the decoded codes other than stdout.
- -v - enables printing statistics to stderr

Matrices to store:
- Generator matrix (G)
- Transpose of parity checker matrix, (H^T)

Arrays/vectors to store:
- Error reference table
- Hamming code vector for input
- Decoded vector for output

The Bit Vector ADT:

Represents a one dimensional array of bits.
For n items, use (n/8) + 1 uint 8s

Bitwise operations are required to handle the individual bits.

ADT variables:

Length, store as a uint32_t
Vector, store as an array of uint8_t

Functions to implement:

bv_create - return pointer to a bit vector, each bit is initialized to zero.
To initialize vector to zero, it's byte contents must all be zero.
bv_delete - destructor, free memory
bv_length - return length in bits, not bytes

bv_set_bit - set a specific bit to 1(?), crucial not to change any other bits*
bv_clear_bit - set a specific bit to 0(?), only change desired bit
bv_get_bit - get a specific bit's value
bv_xor_bit - XOR comparison of bv element i and given binary value

Bit Vector of Length 14 bits:

$\underbrace{(14/8)+2 = 2 \text{ uint8s as vector:}}$ extra

0	0	0	0	0	0	0	0	0	0	0	0	0	0	x	x

set to 0 regardless

* changing specific bit: byte = index / 8
place in byte = index % 8

get bit 13: =

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	1	0	1	1	0	1	0	1	0	0	0	1	0	1

byte $13/8$ = 1. (truncate) → byte 1 (0 is first)

bit : 13 % 8 = 5 → points to 14th bit (at index 13)

Modify vector [byte_value] (bit value)

Breaking uint 8 into it's own array:
L setting a bit done by changing value of UINT.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

= 0

0	1	2	3	4	5	6	7
0	0	0	0	0	1	0	0

= 00100000_2 = 2^5 = 32

Change of bv(13) means setting 2nd UINT to original value + 2^5; 32.

---

* accessing specific bit

0	1	2	3	4	5	6	7
0	0	1	0	1	1	0	0

= 00110100_2 = 4 + 16 + 32 = 52 = uint value

To access bit 3: 52 → USE Logical shifts until 3 is bit 0
- if uint = odd: value = 1

Logical shifts $00110100_2 = 52$ else : value = 0

$00011010_2 = 2+8+16 = 26$

$00001101_2 = 1+4+8 = 13$

$00000110_2 = 2+4 = 6$ even, bit is 0

Logical shifts = using integer division on unit. - 3 times

---

* Using addition to conduct bitwise XOR: XOR 1 with bit 6

$0\boxed{0}110100_2 = 52$

$01000000_2 = 2^6 + 52 = (116/2^6) = 1.8 = 1$ so the

- Add UINT to byte in vector (set by bit value)

If SUM / $2^{\text{bit value in byte}}$ = 1 return true, else false

bv_print - print the bits, separated by spaces, as they are sorted in vector array.

(Bit vector is read *left to right)* Bit 0 is leftmost, the lsb, unlike standard reading.

↳ 7 shifts - check even/odd each time, print results one by one.

Bit Matrix ADT:

Represents a two dimensional array of bits - or an array of bit vectors.*

M*N matrix where M(r,c) = r*N + C

ADT variables:

Rows, store as a uint32_t

Columns, store as a uint32_t

vector, store as a bitVector.

C = 8

r = 4

	0	1	2	3	4	5	6	7
0	0	0	1	0	1	1	0	0
1	0	0	1	0	1	1	0	0
2	0	0	1	0	1	1	0	0
3	0	0	1	0	1	1	0	0

As a vector:

0	0	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1	1	0	0

$M_{2,3} = 2 \cdot 8 + 3 = 16 + 3 = 19$

Functions to implement:

bm_create - return pointer to a bit matrix, number of bits = rows*cols

To initialize vector to zero, it's byte contents must all be zero.

bm_delete - destructor, free memory

bm_rows - return num rows

bm_cols - return num cols

bm_set_bit - set a specific bit to 1(?), crucial not to change any other bits*

bm_clear_bit - set a specific bit to 0(?), only change desired bit

bm_get_bit - get a specific bit's value

These 3 functions will simply call the bit vector functions for the correct bit, showcased *above.*

*bm_from_data - make a 1 row bit matrix out of the first LENGTH number of bits in a matrix
bm_to_data - return the first 8 bits of a bit matrix as a uint8_t
Simply return the uint8_t in position 0 of the bit vector.

*bm_multiply - return multiply two bit matrices using matrix multiplication, return a new BM with the result.
NOTE, you multiply matrix A by matrix B MOD2!.

bm_print - print the bits, separated by spaces, as they are sorted in vector array.

The Hamming Code Module:

HAM_STATUS contains three outcomes:
HAM_OK = -3 (no errors)
HAM_ERR = -2 (Error not fixable)
HAM_CORRECT = -1 (Error fixed)

Ham functions:
ham_encode - Given a nibble from a bitMatrix, adds the Hamming bits to return a fill byte (uint8_t) of Hamming code.

ham_decode - Return a nibble (decoded from a coded byte of Ham Code) to the beginning of a specified bitMatrix. <u>Then also return the HAM STATUS from the decoding process.</u>

Systemic Hamming Code : (8, 4)

Note, in the bit vector these bits are reflected.

$P_0 = xor(D_0\ D_1\ D_3)$
$P_1 = xor(D_0\ D_2\ D_3)$
$P_2 = xor(D_1\ D_2\ D_3)$
$P_3 = xor(all)$

Index	7	6	5	4	3	2	1	0
Hamming code	P_3	P_2	P_1	P_0	D_3	D_2	D_1	D_0
Example for 6:	0	0	1	1	0	1	1	0
	According to rules				Nibble = 6			
Example for 1	0	0	1	1	0	0	0	1

Prelab Questions:

1) Lookup table:

Index (decimal of error syndrome		Bit that must be flipped
(0000_2)	0	No Error found
(0001_2)	1	4
(0010_2)	2	5

(0011_2)	3	Ham Error
(0100_2)	4	6
(0101_2)	5	Ham Error
(0110_2)	6	Ham Error
(0111_2)	7	3
(1000_2)	8	7
(1001_2)	9	Ham Error
(1010_2)	10	Ham Error
(1011_2)	11	2
(1100_2)	12	Ham Error
(1101_2)	13	1
(1110_2)	14	0
(1111_2)	15	Ham Error

2) Error decoding:

1) $(1100\,0111) \cdot H^T = [1 \;\; 2 \;\; 3 \;\; 3] \,\%\, 2 = [1 \;\; 0 \;\; 1 \;\; 1]$

$1101_2 = 13_{10}$ = Swap bit at 1

Output = $1100 \rightarrow [1011]$

2) $(0001\,1011) \cdot H^T = [2 \;\; 1 \;\; 2 \;\; 1] \,\%\, 2 = [0 \;\; 1 \;\; 0 \;\; 1]$

$1010_2 = 10_{10}$ = HAM ERROR

output = unknown.