EECS 281: Homework #4 Due: Thursday, October 7, 2004

Name: ________________________ Email: ________________________

1. Convert the 24-bit number 0x414243 to mime base64: "QUJD"

First, set is to break 8-bit blocks into 6-bit blocks, and then convert: 0x414243 ⇒ b"**010000**01 0100**0010** **01**000011" ⇒ b"010000 010100 001001 000011" ⇒ "16 20 9 3" ⇒ "Q U J D"

2. Convert the base64 "T2s=" to ASCII: "Ok"

One final equal sign (=) means 16 bits, two final equal signs (==) mean 8 bits, and no equal signs mean 24 bits. So, "T2s=" ⇒ "T 2 s =" ⇒ "19 54 44" ⇒ "**010011 11**0110 1011**00**" because of the equal the last 2 bits are discarded ⇒ "01001111 01101011" ⇒ "0x4f 0x6b" ⇒ "O k". Their are no trailing null bits!

3. What is the parity of 0x414243 (even or odd)? odd

Convert to binary and then add: 0x414243 ⇒ "01000001 01000010 01000011" ⇒ 7 '1' bits and since seven is odd it is odd parity.

4. If 0x414243 is odd parity number then is it in error? "insufficient information"

Depends, the receiver must be told is even or odd numbers are bad.

5. Write a "single" C code statement of setting both bits 5 and 2 to 1 in the variable int a.

A quick solution is: "a = a | (1<<5) | (1<<2);" and we could clean this code up as follows: "a |= (1<<5) | (1<<2);" ⇒ "a |= 100000_2 | 0010_2 ;" ⇒ "a |= 100010_2;" (combine all constants 0x20 | 0x04 = 0x24) ⇒ "a |= 0x24;"

6. Write a "single" C code if statement of testing bits 5 and 2 in the variable int a are both true.

A quick solution is "if (a & (1<<5)) { if (a & (1<<2)) { ... } }" which is the same as "if (a&(1<<5) && a&(1<<2)) { ... }". The logical AND "&&" is not the same as the bitwise AND "&".

Note: this code is incorrect: "if ((a & (1<<5)) & (a & (1 << 2))) { ... }". Why? suppose "a=1111 1111". then, substitute the values: "if ((11111111 & 00100000) & (11111111 & 00000010)) { ... }" ⇒ "if ((00100000) & (00000010)) { ... }" ⇒ "if (00000000) { ...}" which is not what the question ask. So, you have to run small test values to make sure it is correct.

Another solution is "if (((a >> 5)&1) & ((a >> 2)&1)) { ... }" and since we know the bit will be shifted to the rightmost bit, we can drop the extra ands: "if ((a >> 5) & (a >> 2)) { ... }". But, still, all this shifting will slow the execution time down. So, can we exploit constants like problem 5?

Yes, the best solution is to see if we can fix up the second case: "if ((a & (1<<5)) & (a & (1 << 2))) { ... }". If we rearrange the masking bits like problem 5: "if (a & ((1<<5) | (1<<2)) == 100010_2) { ... }". ⇒ "if ((a &

0x24)==0x24) { ... }"

7. Write the C code function for a nand: unsigned int nand(unsigned int a, unsigned int b); no loops allowed. Example: nand(0x12, 0x35) is 0xffffffeff.

Since, a NAND is NOT(a AND y), we can write: unsigned int nand(unsigned int a, unsigned int b) { return ~(a & b); }

8. Write the C code function to count the number 1 bits in an integer: unsigned int bcount(unsigned int a); (note: multiply and divide not allowed). Example: bcount(0x1a) is 3.

<u>First solution:</u>

```
unsigned int bcount(unsigned int a) { int i, m=0; for(i=0; i<32; i++) { if (a & (1<<i)) { m++; } } return m; }
```

<u>Better solution:</u>

```
unsigned int bcount(unsigned int a) { int i, m=0; for(i=0; i<32; i++) { m+=a & 1; a>>=1; } return m; }
```

<u>Best code: independant of integer size</u>

```
unsigned int bcount(unsigned int a) { int m=0; while(a) { m+=a & 1; a>>=1; } return m; }
```

9. Write the C code function to return the bit position of the most significant bit: unsigned int bpos1(unsigned int a); (note: multiply and divide not allowed). Example: bpos1(16) is 4 and bpos1(17) is 4. How is this related to the log base 2 of a trunc(log2(17)) or ceil(log2(17))?

bpos1(16) is the same as "trunc(log2(17))"

<u>First solution:</u>

```
unsigned int bpos1(unsigned int a) { int i, m=0; for(i=0; i<32; i++) { if (a & (1<<i)) { m=i; } } return m; }
```

<u>Better solution:</u>

```
unsigned int bpos1(unsigned int a) { int i, m=0; for(i=0; i<32; i++) { if (a & 1) { m=i; } a>>=1; } return m; }
```

<u>Best code: independant of integer size</u>

```
unsigned int bpos1(unsigned int a) { int i, m=0; for(i=0; ; i++) { if (a & 1) { m=i; } a>>=1; } return m; }
```

10. Write the C code function to return 2**i : unsigned int pow2(unsigned int i); (note: multiply and divide not allowed). Example: pow2(3) is 8.

```
unsigned int pow2(unsigned int i) { return 1<<i; }
```

11. What is the hamming distance of 0xAF and 0377 (show work)? ________________

bcount(0xAF ^ 0377) ⇒ bcount(Hexidecimal 0xAF XOR Octal 0377) ⇒ bcount(10101111 XOR 11111111) ⇒ bcount(01010000) ⇒ 2.

12. Write the C code function to compute the hamming distance: int H(unsigned int a, unsigned int b); Example H(3, 5) is 2.

```
int H(unsigned int a, unsigned int b) { return bcount(a ^ b); }
```

13. What is the hamming distance of 0 and 5? 2___ 5 and 7? 1___ 0 and 7? 3___

14. Draw the n-cube of the code set 0, 5, 7 . What is the minimum distance between all these codes? What level of detection or correction does the code set 0, 5, 7 have?

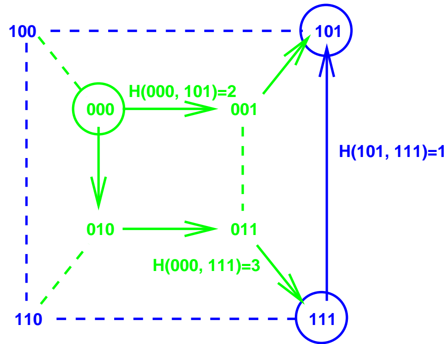


Figure 1: 3-cube showing codes

The Figure 1 clearly shows that the shortest distance between any of the code is 1: Min(H(0,5)⇒2, H(0,7)⇒3, H(5,7)⇒1) ⇒ 1.

With a minimum distance of 1 you cannot claim you can detect the error. If we think about all the possiblities of bit flipping from one code to another then flipping one bit in code 5 can become code 7 which is also legal. So, a minimum hamming distance of 2 between any combination of legal codes, allows us to have "single error detection" (i.e. Min(H(a,b))=2). Adding a parity bit to a number gives us a hamming distance of 2.

For example, suppose, we have a set of 2-bit numbers, { 00, 01, 10, 11 } which we wish to transmit. Clearly, the hamming distance is 1. Now, in order to increase the Hamming distance we must add an extra bit. So let's add an odd parity to each 2-bit number, { 00**1**, 01**0**, 10**0**, 11**1** } which is the same as in decimal { 1, 2, 4, 7 }. We can see that any combination of of these new codes will give a Hamming distance of 2.

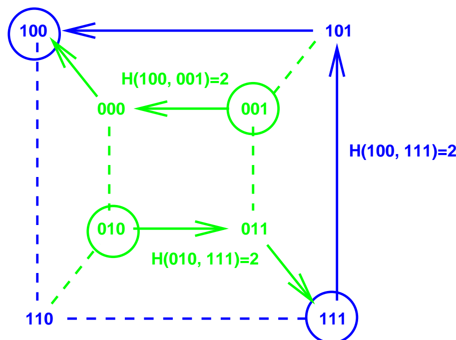


Figure 2: 3-cube showing 2-bit codes with parity

In general, if the minimum distance is 1 then the code is unique and cannot detect or correct certain codes. If

the minimum distance is 2 then "single error detection". If 3 then "single error correction" which means not only can we detect a error but also correct it. If 4 then "single error correction plus double error detection". If 5 then "double error correction" and so on.

15. Give the n-cube, k-map, SOP of the f(a,b,c) minterms for (4, 6), then give the minimize SOP, then draw the logic gate schematic.

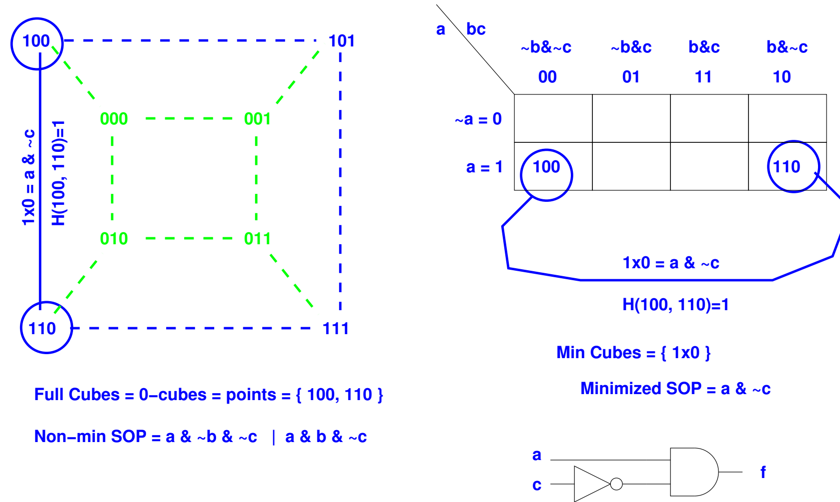


Figure 3: f(a,b,c) minterms for (4, 6)

16. Give the SOP of the f(a,b,c) minterms for NOT(4, 6), then give the minimize SOP. Is it smaller than problem 15?

Ignore this problem, it should have read the NOT(f(a,b,c)).

17. Give the n-cube, k-map and SOP of the f(a,b,c) minterms for (0, 3, 5, 6), then give the minimize SOP. Why didn't it get smaller?

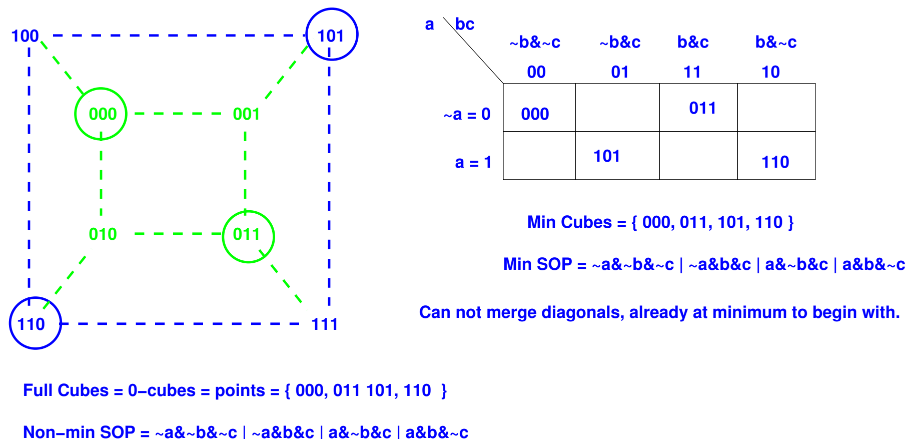


Figure 4: f(a,b,c) minterms for (0, 3, 5, 6)

18. Give the k-map and SOP before and after minimizing the f(a,b,c) minterms for (0, 3, 5, 6)?

Unchanged, because you cannot merge diagonals, you are already at minimum.

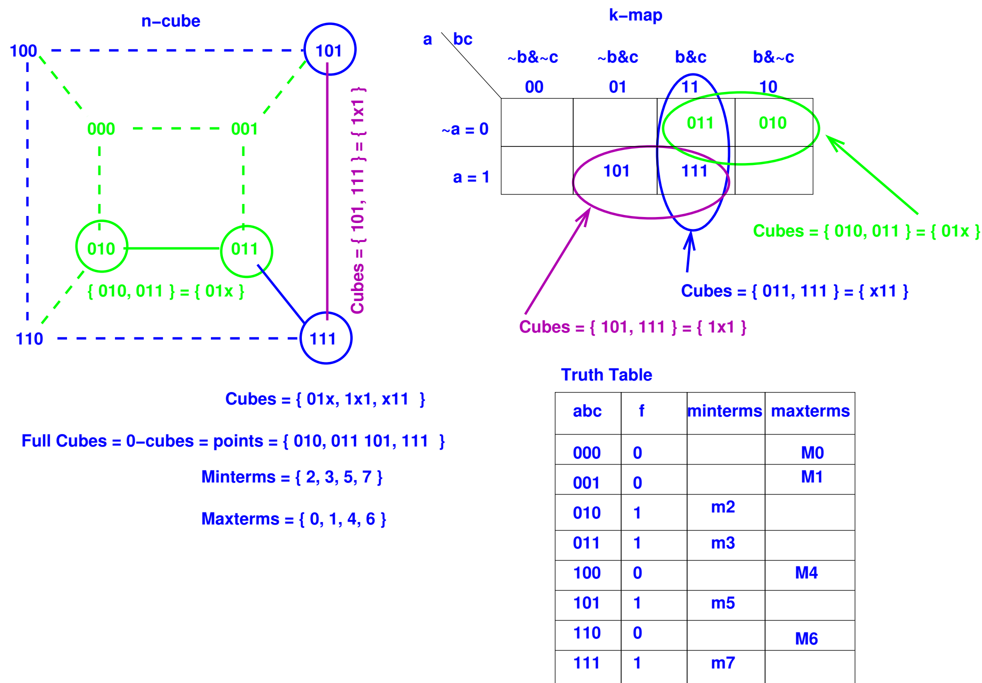
19. Minimize the f(a,b,c,d) minterms for (0, 5, 8, 10, 13). Give n-cube, k-map and SOP.

covered in class, see http://bear.ces.cwru.edu/eecs_281/eecs_281_20041007c.jpg

20. Minimize the f(a,b,c,d) minterms for (0, 5, 8, 10, 13) and a Don't Care minterm of 2. Give n-cube, k-map and SOP. Is it smaller than problem 19?

covered in class, see http://bear.ces.cwru.edu/eecs_281/eecs_281_20041007d.jpg

21. Give the truth table, minterms, maxterms, n-cube, and k-map of 01x, 1x1, x11 :



abc	f	minterms	maxterms
000	0		M0
001	0		M1
010	1	m2	
011	1	m3	
100	0		M4
101	1	m5	
110	0		M6
111	1	m7	

Figure 5: f(a,b,c) minterms for (2, 3, 5, 7)