Aaron Chamberlain
CSE 310
Winter 2016

Midterm #1

1.1) Define gray code and explain why it is used (hint: consider a set of physical toggle switches and a human operator controlling these switches).

Gray code is a special binary numeral system where each pair of successive values differs by one bit. It's use is applied to many fields, but particularly in error correction and simplifying analog to digital input conversion. An example table of natural binary numbers and gray code is provided.

Decimal Value	Natural Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Imagine the input to a computer is a series of 3 switches, where each switch represents a bit of the above sequences and the "off" position represents a 0 while the "on" position represents a 1. With a natural binary system, to increment from 5 to 6 one must two switches and to change from 3 to 4 requires the user to flip all 3 switches. Not only does this increase the probability of having incorrect user input, but it also causes problems with digital logic systems. Likely, the user will not perform these changes within nanosecond precision, thus causing the input to have transitory state that is not what the user desires to input. Gray code solves this problem by making it so that a change of input value by 1 only changes one bit of the input, and allows the logic designer to implement this change in value however he so desires.

Research Source:

1) Last modified by: Harvey M. (2016) Gray Code. Retrieved Feb. 16, 2016 from the World Wide Web. https://en.wikipedia.org/wiki/Gray_code

1.2) Consider a 5 bit gray code.
1.2.1) How many entities can 5 bits encode? -explain your reasoning, do not just state an answer.

Given that Gray Code is a binary numeral system, it will encode the same number of values as a natural binary system, with the difference being in how the two are incremented. Given this, we may use the well known equation where n-bits will represent 2^n values. This can logically be deduced by the fact that each of the n bits can only have 2 possible values, 0 or 1. So in this case, 5-bits will represent 2^5 values, or 32 values (0-31 for integers). A “proof by exhaustion” for this case can be seen in the form of a table below:

Integer	Binary	Integer	Binary	Integer	Binary	Integer	Binary
0	00000	8	01000	16	10000	24	11000
1	00001	9	01001	17	10001	25	11001
2	00010	10	01010	18	10010	26	11010
3	00011	11	01011	19	10011	27	11011
4	00100	12	01100	20	10100	28	11100
5	00101	13	01101	21	10101	29	11101
6	00110	14	01110	22	10110	30	11110
7	00111	15	01111	23	10111	31	11111

In this case it can be seen that, as expected, 5 bits will encode from 0 to 31 in decimal integer values. Although this table denotes the natural binary numbers, as notes above, there is no change of limit for gray code due to the fact that both are binary numeral systems. The only difference to such a table in Gray Code would be how the binary values are incremented.

1.2.2) Assume the ordinary binary representation of a hex digit (e.g., 0xA is binary 1010). Write a 5 digit gray code sequence from 0 to the largest number that can be encoded in five bits. Then write both the ordinary binary and ordinary hexadecimal representation corresponding to each number in that sequence.

5-Bit Gray Code	Natural Binary	Hexadecimal
00000	00000	0x0
00001	00001	0x1
00011	00010	0x2
00010	00011	0x3
00110	00100	0x4
00111	00101	0x5
00101	00110	0x6
00100	00111	0x7
01100	01000	0x8
01101	01001	0x9
01111	01010	0xA
01110	01011	0xB

5-Bit Gray Code	**Natural Binary**	**Hexadecimal**
01010	01100	0xC
01011	01101	0xD
01001	01110	0xE
01000	01111	0xF
11000	10000	1x0
11001	10001	1x1
11011	10010	1x2
11010	10011	1x3
11110	10100	1x4
11111	10101	1x5
11101	10110	1x6
11100	10111	1x7
10100	11000	1x8
10101	11001	1x9
10111	11010	1xA
10110	11011	1xB
10010	11100	1xC
10011	11101	1xD
10001	11110	1xE
10000	11111	1xF

1.3) Consider ordinary unsigned addition for a 3-bit gray code values - i.e., the sum of two 3-bit gray code values as a gray code value.
1.3.1) Given all possible 3-bit input values for a 3-bit gray code, what are the possible output values as gray codes? How are these gray codes interpreted as "ordinary" numbers?

As discussed above, a 3-bit sequence can represent 2^3, or 8 distinct values. If we assume that we will not ignore the overflow that may occur from the maximum of these possible values (111 & 111), we will arrive at a 6-bit resultant value. Given this, that means that there 2^6 possible, or 64 possible combinations. Each of those values will be listed in the table below.

In this context, Gray Code represents an incremental state and is not directly converted to another radix. In other words, each subsequent gray code sequence can be simply thought of as the previous state + 1 in decimal.

6-Bit Gray Code pt. 1	6-Bit Gray Code pt. 2	Decimal Representation	Decimal Rep. Pt. 2
000000	110000	1	33
000001	110001	2	34
000011	110011	3	35
000010	110010	4	36
000110	110110	5	37
000111	110111	6	38
000101	110101	7	39
000100	110100	8	40
001100	111100	9	41
001101	111101	10	42
001111	111111	11	43
001110	111110	12	44
001010	111010	13	45
001011	111011	14	46
001001	111001	15	47
001000	111000	16	48
011000	101000	17	49
011001	101001	18	50
011011	101011	19	51
011010	101010	20	52
011110	101110	21	53
011111	101111	22	54
011101	101101	23	55
011100	101100	24	56
010100	100100	25	57
010101	100101	26	58
010111	100111	27	59
010110	100110	28	60

6-Bit Gray Code pt. 1	6-Bit Gray Code pt. 2	Decimal Representation	Decimal Rep. Pt. 2
010010	100010	29	61
010011	100011	30	62
010001	100001	31	63
010000	100000	32	64

1.3.2) If you are constructing a full adder, there are two additional values: carry in and carry out. Extend your solution to 1.3.1. include these two additional values, and display the result as a truth table using 1 and 0 entries.

1.3.3) Using the truth table in 1.3.2, construct the Karnaugh map for the 3-bit gray code full adder. Does this Karnaugh map reveal any hazards?

1.3.4) Write an appropriate Verilog program to implement the 3-bit gray code adder.
To address this problem, a full adder module will be implemented that can then be instantiated and applied to each bit of a sequence in a separate 3-Bit Adder Module.

```
//file: full_adder.v
module full_adder (a,b,c,sum,carry);

output sum;
output carry;

input a;
input b;
input c;

assign sum = a^b^c;
assign carry = (a&b) | (b&c) | (c&a);

endmodule

//file: 3_bit_adder.v
//instantiates the full adder defined above, applies it to each bit.
module 3_bit_adder (a,b,sum,carry);

output [2:0] sum;
output carry;

input [2:0] a;
input [2:0] b;

wire [1:0]s;
```

```
full_adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
full_adder u1 (a[1],b[1],s[0],sum[1],s[1]);
full_adder u2 (a[2],b[2],s[1],sum[2],carry);

endmodule
```

1.3.5) Combine appropriate 3-bit gray code full adders to produce a single 6-bit gray code full adder.

My reasoning in defining the full_adder module above in a single bit instance is so that it can be repeated and applied to any n-bit sequence. In the case of a 6-bit gray code adder, I don't have to redefine this lower level module at all. Instead, all I must do is widen the inputs and outputs to allow for correct representation of all values and then carry the logic across all bits. The Inputs and Outputs a, b, and sum will be widened to [n-1:0] and the wire s will be widened to [n-2:0]. The revised verilog module can be found below.

```
//file: 6_bit_adder.v
//instantiates the full adder defined above, applies it to each bit.
module 6_bit_adder (a,b,sum,carry);

output [5:0] sum;
output carry;

input [5:0] a;
input [5:0] b;

wire [4:0]s;

full_adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
full_adder u1 (a[1],b[1],s[0],sum[1],s[1]);
full_adder u2 (a[2],b[2],s[1],sum[2],s[2]);
full_adder u3 (a[3],b[3],s[2],sum[3],s[3]);
full_adder u4 (a[4],b[4],s[3],sum[4],s[4]);
full_adder u5 (a[5],b[5],s[4],sum[5],carry);

endmodule
```

1.4) Consider an anti-gray code which maximizes hamming distance for adjacent values in a 4-bit binary sequence, starting at 0.

1.4.1) Design and write such a sequence in binary representation.

The following table shows the process of generating Anti-Gray Code, with the result on the far right column.

3-Bit Gray	3-Bit Left Shift	Double List	4-Bit Anti-Gray Code
000	0000	0000	0000

3-Bit Gray	3-Bit Left Shift	Double List	4-Bit Anti-Gray Code
001	0010	0000	1111
011	0110	0010	0010
010	0100	0010	1101
110	1100	0110	0110
111	1110	0110	1001
101	1010	0100	0100
100	1000	0100	1011
		1100	1100
		1100	0011
		1110	1110
		1110	0001
		1010	1010
		1010	0101
		1000	1000
		1000	0111

Research Source:
1) Anonymous author (2012). Anti-gray codes, How to derive them. Retrieved Feb. 16, 2016 from the World Wide Web. http://godsnotwheregodsnot.blogspot.com/2012/08/anti-gray-codes-how-to-derive-them.html

1.4.2) What is the average hamming distance for the values in this sequence?

The hamming distance can be calculated by enumerating the number of bits that differ in two strings of equal length. For example, the hamming distance between 0000 and 1111 is 4 because every bit of the two sequences differ from one another.

The average hamming distance can be calculated manually for a sequence this small. The 16 values of this sequence form 8 pairs where the hamming distance for each respective pair is: 4, 4, 4, 4, 4, 4, 4, 4. Given no variance in the hamming distance values, the average distance is 4.

Research Source:
1) Last Modified by: Zorro1024 (2016). Hamming Distance. Retrieved Feb. 16 2016 from the World Wide Web. https://en.wikipedia.org/wiki/Hamming_distance

1.4.3) What real-world applications would such a sequence have?

The sequence's properties make it so that they are the furthest hamming distance away from one another. Another way of putting this might be that each value is maximally distinct from one another. Of the practical uses that come to mind, the most seems to be in relation to using random values that are as distinct as possible. For example, imagine you are creating a graph with 4 different colors for the data points. One method to get these random colors could be to use a hash, but this could possibly generate two very similar values such as grey and light grey. Using an anti-gray code in this situation would produce the exact opposite of another color.

2) We have discussed 1-hot code.

2.1) Define 1-hot code and explain why it is used (hint: consider a set of physical toggle switches and a human operator controlling these switches).

1-Hot code encoding takes place in a set of binary bits where at any given time only one bit may be high (1) and the rest must be in a low state (0). All possible combinations in a 3-bit group would be the following: 100, 010, 001, for example. This type of encoding is typically used within state machines, because it has a constant hardware cost. To determine a devices state, only 1 bit must be read, or in this case only one flip-flop must be polled. To change the state of the machine, only 2 flip-flops must be accessed.

Consider a machine that has 8 physical switches, each switch controls a state. To switch from state 1 (10000000) to state 2 (01000000), the user must one change the first switches position, and then flip the 2nd switch from the left high. When this bit is high the digital logic of the system would perform only the operations of this state. In this way, it is easy to ensure a valid state of the machine at all times.

2.2) Display the 3-bit regular binary and the 1-hot code that represents the same value.

Binary	**One-Hot**
000	10000000
001	01000000
010	00100000
011	00010000
100	00001000
101	00000100
110	00000010
111	00000001

2.3) 3-bits correspond to what radix?

The radix of any value denotes the number of possible values that any single bit can have. The most commonly used radix is radix 10, or decimal. In this case any one digit can be the values 0 to 9.

Since 1-Hot Code is denoted using only the values 0 or 1, it's radix is 2, or binary. No matter how many bits are in a sequence of numbers, 3-bits in this case, the radix will be the same.

2.4) A 1-cold code is the same as a 1-hot except that the 1-cold has a low where the 1-hot has a high, and vice versa. How does one convert a 1-hot code to a 1-cold code?

As expressed in the question, 1-Hot code differs from 1-Cold code only in the value that they chose to represent the state. For example, in a 4-bit state machine representing the 2nd state would be written as 0010, whereas in 1-Cold it would be written as 1101.

The easiest way to achieve this result is to simply run the 1-Hot code through a NOT gate for every bit. In this way, the entire sequence is inverted and obtains the result.

2.5) Write a truth table that shows the conversion of 3-bit ordinary to a 1-cold code.

Note that because of issues related to not having available certain characters, as well as limited space, the following non-standard syntax and symbology is observed: 1) The ‘ represents the NOT logic gate, or the unary input inverter. This is standardly written as NOT A or ~A. 2) A group of logical values with no operators between them implies the AND gate. For example, ABC means A AND B AND C, or A&B&C.

Had there been more space, the standards would have been used. The entire 1-Cold representation is the equivalent of values Q0 through Q7 in one set. For example, the 1-Cold value derived from the first row is 11111110.

A	**B**	**C**	**A’**	**B’**	**C’**	**(ABC)’ (Q0)**	**(ABC’)’ (Q1)**	**(AB’C)’ (Q2)**	**(AB’C’)’ (Q3)**	**(A’BC)’ (Q4)**	**(A’BC’)’ (Q5)**	**(A’B’C)’ (Q6)**	**(A’B’C’)’ (Q7)**
0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	0	1	1	1	0	1	1	1	1	1	1	0	1
0	1	0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	1	0	0	1	1	1	1	0	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	1	0	1	0	1	1	0	1	1	1	1	1
1	1	0	0	0	1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	1	1	1	1	1	1	1

2.6) Write a Verilog program that converts a 3-bit binary number into 1-cold code.

//A simple decoder module that takes a 3-bin in and outputs 8-bit 1-Cold Code

```
module dec (bin, ocout);
  input [2:0] bin;
  output [7:0] ocout;
  reg [7:0] bin;
  always @(bin)

  begin
    //based off the binary input, select the appropriate 1-C out.
    case (bin)
      3'b000 : ocout = 8'b11111110;
      3'b001 : ocout = 8'b11111101;
      3'b010 : ocout = 8'b11111011;
      3'b011 : ocout = 8'b11110111;
      3'b100 : ocout = 8'b11101111;
      3'b101 : ocout = 8'b11011111;
      3'b110 : ocout = 8'b10111111;
      default : ocout = 8'b01111111;
    endcase
  end
endmodule
```

2.7) Are the any hazards in your circuit? If so, how would you correct these?

In the way I have implemented the circuit, there are no hazards, because I am simply selecting a state based of the entire input. Therefore only once all 3 input registers have been set will my logic evaluate to it’s equivalent 1-Cold Code output, which will also behave synchronously.

Had I implemented it in the same way that the truth table from 2.5 carries out this conversion, there would be a hazard. All three of the inputs run through a NOT or inverter gate before going into the final NAND gate. Therefore there will be a tiny delay between when the non-inverted and the inverted values reach the NAND gate. This will result in either a Static-1 or a Static-0 hazard, depending on which of the resultant Q-Bits being observed. To solve this, we can add in redundant logic before the inputs reach the AND gate.