Melbourne School of Engineering

Engineering Systems Design 2

Assignment One

Digital Systems

The maximum number of marks for this assignment is 60 (there is a bonus question worth 8 marks but your total mark will be capped at 60). It will contribute towards 2% of your assessment in this subject.

1. **[8 marks in total]**
   This question concerns the difference between analog and digital systems, and will require some research in the library and/or on the internet. Each part of this question requires a written answer of 4-5 sentences (at most half a page) in words understandable by a typical Arts student.

   (a) **[3 marks]** Explain how audio sound is recorded onto, stored on and played back from a compact disc (CD).

   (b) **[3 marks]** Explain how audio sound is recorded onto, stored on and played back from a vinyl record.

   (c) **[2 marks]** Which technology (digital CDs or analog vinyl records) do you think is superior? In what respects is it superior? Justify your choice.

2. **[6 marks in total]**
   Captain Janeway sends an away team to check out a completely new, uncharted planet, and they discover the ruins of an ancient pre-warp civilisation. Artifacts and pictures suggest the creators of the civilisation were four legged beings, that had one long tentacle coming out of their back, with the tentacle branching at the end with a number of grasping "fingers". Many of the artifacts contain inscriptions that appear to be mathematical, and the first to be translated (with limited away-team gear) comes out as the equation

   $$5\,x^2 - 50\,x + 125 = 0$$

   with the indicated solutions $x = 5$ and $x = 8$. The first solution looked right, but the second was puzzling. Seven-of-Nine (as she often does) quickly came up with an explanation: like that of humans, the base of the number system for this civilisation is equal to the total number of fingers. So how many fingers did these beings have?

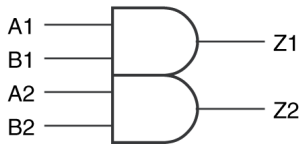

Figure 1: The BUT gate.

3. **[8 marks in total]**
   Some people think that there are four basic logic functions, AND, OR, NOT, and BUT. The figure below is a possible symbol for a 4-input, 2-output BUT gate.

   In this question you are to invent a useful, nontrivial function for the BUT gate to perform. The function should have something to do with the name (BUT). Keep in mind that due to the symmetry of the symbol, the function should be symmetric with respect to the A and B inputs of each section and with respect to sections 1 and 2.

   (a) **[4 mark]** Describe your BUT's function and write its truth table.

   (b) **[4 mark]** Find minimal sum-of-products expressions for the BUT-gate outputs.

4. **[8 marks in total]**
   A compact way of representing a function is simply to list the minterms corresponding to the rows of the truth-table that result in an output of 1. We do this using the notation

$$\begin{aligned} M(X,Y,Z) &= \Sigma_{X,Y,Z}(1,2,4,7) \\ &\equiv \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ. \end{aligned}$$

   Another example of this notation is

$$\begin{aligned} N(W,X,Y,Z) &= \Sigma_{W,X,Y,Z}(5,11,13,14) \\ &\equiv \overline{W}X\overline{Y}Z + W\overline{X}YZ + WX\overline{Y}Z + WXY\overline{Z}. \end{aligned}$$

   If there are some inputs for which we don't care about the corresponding output, rows 3 and 5 for example, then we will write $d(3,5)$ to denote this.

   Two functions F and G are dependent on the four Boolean variables A, B, C and D. The two functions are defined by the following expressions:

$$\begin{aligned} F &= \Sigma_{A,B,C,D}(0,1,2,3,5,7) \\ G &= \Sigma_{A,B,C,D}(5,7,12,13,14,15) \end{aligned}$$

   (a) **[2 marks]** Construct the K-map for F and derive a sum-of-products expression for F that has the fewest possible product terms.

   (b) **[2 marks]** Construct the K-map for G and derive a sum-of-products expression for G that has the fewest possible product terms.

(c) **[4 marks]** Revisit the K-maps for F and G and see if you can produce an implementation that requires fewer product terms in total (and thus fewer AND gates).

5. **[16 marks in total]**

In this question you will design a decoder for the (7,4) Hamming code. This is the circuit that you will implement in Workshop 3 so it is well worth putting in the time and effort to get your design right.

As discussed in Workshop 1, the (7,4) Hamming code takes four information bits $(b_4\, b_3\, b_2\, b_1)$ and adds three parity check bits $(p_3\, p_2\, p_1)$ to give a codeword

$$(c_7\, c_6\, c_5\, c_4\, c_3\, c_2\, c_1) = (b_4\, b_3\, b_2\, p_3\, b_1\, p_2\, p_1).$$

The ordering of the information bits and parity check bits in the final codeword is not critical however the ordering given above has some nice properties as we will see later.

The check bits $(p_3\, p_2\, p_1)$ are chosen as follows:

- p_3 is chosen so as to give an even number of 1s in the group $(c_7\, c_6\, c_5\, c_4) = (b_4\, b_3\, b_2\, p_3)$;
- p_2 is chosen so as to give an even number of 1s in the group $(c_7\, c_6\, c_3\, c_2) = (b_4\, b_3\, b_1\, p_2)$;
- p_1 is chosen so as to give an even number of 1s in the group $(c_7\, c_5\, c_3\, c_1) = (b_4\, b_2\, b_1\, p_1)$.

Note that the bits making up the first group $(c_7\, c_6\, c_5\, c_4)$ are those for which the binary representation of the codeword position has a 1 in the first bit: 7 (111), 6 (110), 5, (101) and 4 (100). Similarly, the second group consists of the bits in positions which have a 1 in the second bit: 7 (111), 6 (110), 3 (011) and 2 (010). The third group consists of bits which have a 1 in the third bit: 7 (111), 5 (101), 3 (011) and 1 (001). This is illustrated in the figure below.

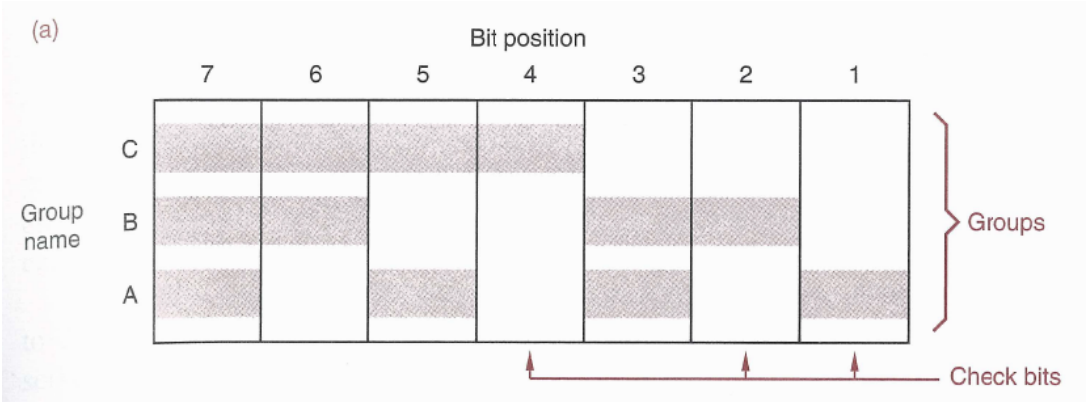

Figure 2: Parity-check matrix for (7,4) Hamming code. This is Figure 2-13 (a) in John F. Wakerly, Digital Design: Principles and Practices, 4th edition, Pearson Prentice Hall, 2006

The 7-bit codewords are then stored or transmitted over a communication channel. In this process, errors can occur so that the bits read or received may not be the same as

the original codeword. We will assume that of the seven bits in a codeword either none of the bits are in error or exactly one of the bits is in error. The structure of the codeword allows the decoder to work out exactly which bit, if any, is in error.

The decoder design can be broken down into two stages, the syndrome generator and the error corrector. In the first stage, the parity of each of the three code groups mentioned above is calculated. For each group we get an output that is 1 if the parity is incorrect (an odd number of 1s) and 0 if it is correct (an even number of 1s). The result is a 3-bit sequence $s_3\, s_2\, s_1$ called the syndrome. Note that s_3 corresponds to the code group that has p_3 in it (the top group in Fig. 2), s_2 corresponds to the code group that has p_2 in it (the middle group in Fig. 2), and s_1 corresponds to the code group that has p_1 in it (the bottom group in Fig. 2).

Looking carefully at Fig. 2 you should be able to see that the value of the syndrome $s_3 s_2 s_1$ tells us which bit, if any, is in error. The second stage of the decoding involves using the syndrome to correct bit errors. For example, if the syndrome is 111, then the parity of all three groups is incorrect and the only single bit error that causes this is an error in $c_7 = b_4$. This means that if the syndrome is 111 then the decoder should flip c_7. If the syndrome is 110, then the parity of the top group and the middle group is incorrect while the parity of the bottom group is correct. The only single bit error that causes this is an error in $c_6 = b_3$ so if the syndrome is 110 then the decoder should invert c_6. Similar reasoning can be applied to other values of the syndrome leading to the result that bit c_i is in error if the decimal representations of the 3-bit syndrome (treated as a 3-bit binary number) is i. Also observe that if the syndrome is 000 then none of the bits in the codeword are in error (since we have assumed that there is at most one error).

Your task is to design a digital circuit that takes the 7-bit codeword $c_7\, c_6\, c_5\, c_4\, c_3\, c_2\, c_1$ as input, and outputs the decoded information bits which we denote by $d_4\, d_3\, d_2\, d_1$. Your circuit design should follow the two stage process discussed above so that at the top level, your circuit has the structure shown in Figure 3.

(a) **[8 marks]** Design the **syndrome generator** block using no more than nine (2-input) XOR gates. More advanced designs might look to use only eight XOR gates and would pay attention to the maximum propagation delay of their circuit (the maximum number of gates that is traversed from any input to any output).

(b) **[8 marks]** Now design the **error corrector** block using no more than four 3-input AND gates, four (2-input) XOR gates and three inverters. Explain why the **error corrector** block does not have c_4, c_2 or c_1 as inputs.

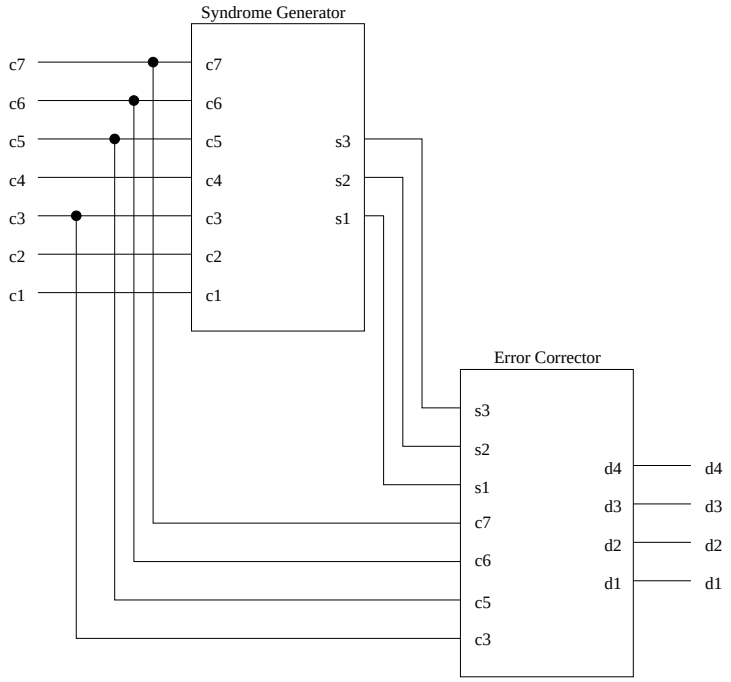

Figure 3: Top-level view of the (7,4) Hamming decoder

6. **[14 marks in total]**
A binary integer B in the range $0 \leq B < 2^7$ can be represented by 7 bits in "fixed-point" format, $B = B_6B_5B_4B_3B_2B_1B_0$. We can represent numbers in the same range with less precision using 6 bits in a floating-point notation, $F = M \cdot 2^E$, where M is a 4-bit mantissa $M = M_3M_2M_1M_0$ and E is a 2-bit exponent $E = E_1E_0$. In this system, the smallest integer is $0 \cdot 2^0$ and the largest is $15 \cdot 2^3$. We can write $B = M \cdot 2^E + T$ where the truncation error T satisfies $0 \leq T < 2^E$. For example:

   - if $B = 1011010$ we would choose $M = 1011$ and $E = 11$ and the truncation error would be $T = 10$;
   - if $B = 0110101$ we would choose $M = 1101$ and $E = 10$ and the truncation error would be $T = 1$.

   (a) **[4 marks]** Start by expressing the following binary integers B in the form $B = M \cdot 2^E + T$ where M, E and T are as specified above: (i) 1111111; (ii) 1111000; (iii) 0000000; (iv) 0001100; and (v) 1010101.

   (b) **[10 marks]** Design a combinational circuit which takes as input the 7-bit unsigned binary integer $B = B_6B_5B_4B_3B_2B_1B_0$ and which outputs the 4-bit mantissa $M = M_3M_2M_1M_0$ and the 2-bit exponent $E = E_1E_0$ of the floating-point representation of the number. You do not need to sketch the circuit, rather you should write down a sum-of-products expression for each of the six ouput variables in terms of the input variables. As always, you should explain the reasoning behind your design.

7. **BONUS QUESTION [8 marks in total]**

Suppose you wish to come up with a code for storing the twenty-six letters of the alphabet (lower case only). Each codeword must consist of a string of 0s and 1s so that it is suitable for storage and processing in a computer.

A fairly simple code is shown in Table 1. The code is a fixed-length code because every letter is represented by an equal number of bits — five bits per letter in this case. Note that we could not use fixed-length codewords with length less than five. Also note that it would be wasteful to use more than five bits per letter.

Table 1: A simple fixed-length code for the lower case letters

Letter	Codeword	Letter	Codeword
a	00000	n	01101
b	00001	o	01110
c	00010	p	01111
d	00011	q	10000
e	00100	r	10001
f	00101	s	10010
g	00110	t	10011
h	00111	u	10100
i	01000	v	10101
j	01001	w	10110
k	01010	x	10111
l	01011	y	11000
m	01100	z	11001

The question that now arises is whether or not we can come up with a code that is more efficient than this fixed-length code. By efficient we mean that, on average, the number of bits needed to code strings of letters is lower.

The first thing we can do is drop the requirement that each codeword is the same length and allow variable-length codes. We now have the freedom for some letters to have codewords with a small number of bits and others to have longer codewords. For example, since the letter e occurs much more frequently than the letter q, we would want to assign a short codeword to the letter e and a longer codeword to the letter q.

To help us with our task we can use the information in Table 2 which shows the relative frequencies of each letter in general English plain text (taken from *Cryptological Mathematics*, R. E. Lewand).

A Huffman code is a variable-length code that takes advantage of knowledge of the relative probabilities of the symbols that are to be coded.

Table 2: Relative frequencies of letters in general English plain text.

Letter	Probability	Letter	Probability
a	0.08167	n	0.06749
b	0.01492	o	0.07507
c	0.02782	p	0.01929
d	0.04253	q	0.00095
e	0.12702	r	0.05987
f	0.02228	s	0.06327
g	0.02015	t	0.09056
h	0.06094	u	0.02758
i	0.06966	v	0.00978
j	0.00153	w	0.02360
k	0.00772	x	0.00150
l	0.04025	y	0.01974
m	0.02406	z	0.00074

The first step in constructing a Huffman code is to build a code tree. This tree consists of branches which can split to form two new branches. The point at which a branch splits is called a node. A branch that does not split ends at a terminal node. Terminal nodes will correspond to the symbols that we wish to code.

We will start at the terminal nodes with one terminal node for each symbol and work backwards. We proceed to combine two branches into one in a special way until we only have one branch left.

To get familiar with constructing a Huffman code we will work through an example that is simpler than our twenty-six symbol source. Consider a source with only five symbols A, B, C, D and E with probabilities 0.1, 0.15, 0.1, 0.5 and 0.15 respectively. The steps required to build the code tree are shown in Figures 4 - 7.

Step 1: To begin we order the symbols by their probability of occurrence along with a corresponding branch for each symbol (terminal node). We then combine the two branches with the lowest probabilities into one new branch with probability equal to the sum of the original branch probabilities. Note also that we have labelled one branch (the upper branch) with a 1 and the other (the lower branch) with a 0.

Step 2: We now repeat the above process taking into account that the two lowest probability branches have been replaced by one new branch. The two lowest probability branches correspond to symbols B and E.

Step 3: Repeat the process.

Step 4: Repeat the process one more time after which we end up with a single branch with a probability of one.

The codes for each symbol can be obtained from the final tree by tracing a path from the last branch to the terminal nodes (the original symbols). The codeword assigned to a symbol is the sequence of 1s and 0s traversed on the path to that symbol. In our example we will have: A - 001, B - 011, C - 000, D - 1 and E - 010.

For parts (a) to (d) below we will work with a reduced alphabet with probabilities of occurrence as shown in Table 3.

Table 3: Relative frequencies of letters in general English plain text.

Letter	a	h	o	r	t	u	z
Probability	0.224	0.144	0.177	0.138	0.245	0.068	0.004

(a) **[1 mark]** Construct a fixed-length code for the seven lower case letters in Table 3.

(b) **[3 marks]** Construct a Huffman code for the seven lower case letters with probabilities as given in Table 3.

(c) **[1 mark]** The average number of bits per symbol required by the code is given by

$$\bar{l} = \sum_{i=1}^{M} p_i \, l_i$$

where M is the number of symbols, p_i is the probability of symbol i, and the l_i is the number of bits in the codeword for symbol i.

Calculate the average number of bits per symbol required by the Huffman code and compare this number to the number of bits per symbol required by your fixed-length code from (a).

(d) **[2 marks]** Do you notice anything special about the codewords in the Huffman code? (Hint: Think about how you would decode a string of 1s and 0s. This would be easy for a fixed-length code but is a little more difficult for a variable-length code such as the Huffman code.)

(e) **[1 mark]** Compare the average number of bits per symbol to the quantity known as entropy:

$$H = \sum_{i=1}^{M} -p_i \log_2 p_i$$

The entropy sets a lower limit on the required number of bits per symbol so your average codeword length should be bigger than the entropy.

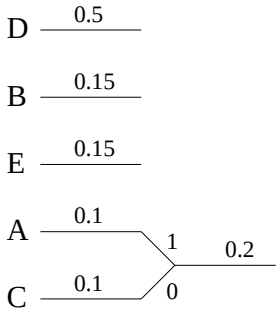

Figure 4:

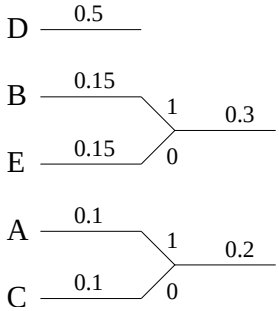

Figure 5:

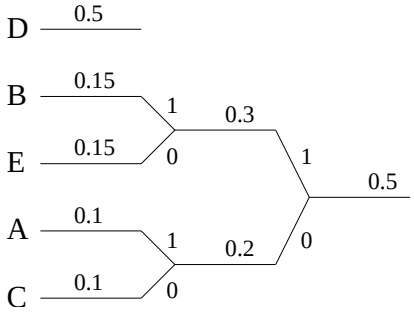

Figure 6:

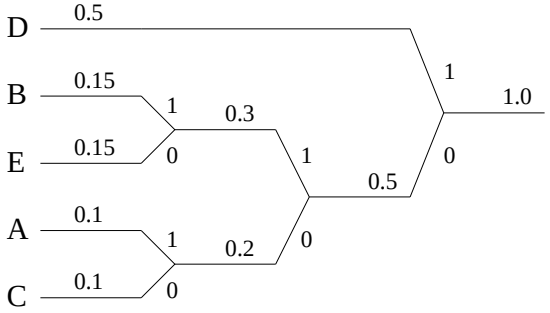

Figure 7: