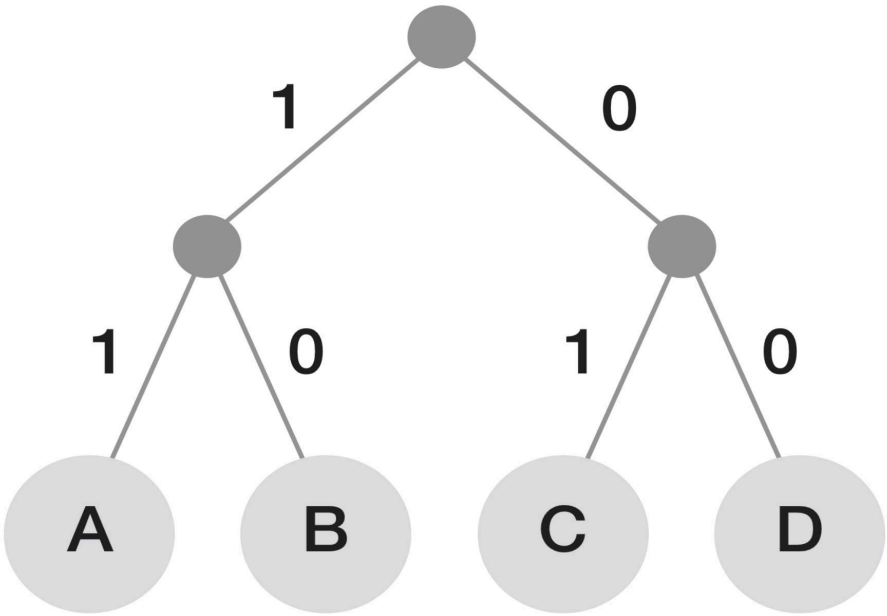
EE224 DIGITAL ELECTRONICS

Team 12

HUFFMAN CODING



Authors

200260015	Devashish Shah
200260023	Kaushik Dhandekar
200260030	Arth Mendhe
200260035	Parmy Parmar
20d170032	Reet Mhaske
200260049	Shashwat Chakraborty

ABSTRACT

This report presents the circuits conceptualised and designed to perform **compression** of a large string of data using **Huffman coding,** which enables efficient storage of data. This coding system uses codes of different lengths to represent characters depending on the frequency of using that character, unlike standard encoding systems that use codes of a fixed length. The algorithm for generating a Huffman code is discussed below.

The main purpose of the circuit can be demonstrated thus. A user inputs a string of characters. The ASCII equivalents of these are stored in a "Temporary memory".We assume this is achieved by an external circuitry.
These stored ASCII equivalents are then accessed twice, once to count frequencies and make the Huffman code and for a second time to convert the ASCII into Huffman code and store the string in Permanent Memory.

To achieve the latter, our circuit performs the following functions in order:

1. While receiving the ASCII codes for the first time from temporary memory, it identifies the character from the ASCII equivalent (Circuit C1). Since the ASCII codes are 8-bit codes, we use an 8 to 3 ASCII decoder for the same.
2. During the same run, it counts the number of occurrences of each character (Circuit C2). This is achieved by using universal counters.
3. After having counted the occurrences, it identifies the least occurring character, assigns it a code of the longest length. It iteratively finds the least occurring character from the remaining ones, at each iteration assigning a code of length 1bit less than the previous code. The most frequently occurring character gets a code of length 1.

   To identify the least occurring character, we use the fact that if the counters storing the counts of occurrences of each character are made to start counting DOWN the clock pulses from their current state, the one corresponding to the least occurring character will count down to zero first (Circuit C2)Thus the Huffman codes have been computed and stored in registers during the first access of the ASCII codes stored in the temporary memory.

4. During the second access of the ASCII codes stored, we identify the characters using Circuit C1. On identifying a particular character, the code corresponding to it gets serially stored into the permanent memory. (Circuit C3)

To achieve this, we use a temporary register, into which the Huffman code to be stored is parallelly loaded, and serially shifted into the permanent memory.

We note that if we are using a 4 bit register, which stores a 2 bit code, then we need to ensure that only the bits relevant to the code are stored in the memory. For example, a 4 bit register supposed to store "10" will actually be storing "1000". We realise that all the Huffman codes generated above, *end with 0** (except for the maximum length code which ends in 1). We use this fact, and stop shifting the code in the permanent memory when either the last bit shifted into the memory is 0 or the maximum length of code has been shifted.

This is achieved with the use of a feedback circuit and a 555 timer in the monostable configuration.

Circuit 3 uses a clock with frequency sufficiently larger than (an integral multiple) that of the clock used to read the data from the temporary memory in circuit C1.Both clocks are synchronised to have an initial simultaneous positive edge. This ensures that the code is shifted into the permanent memory before the next ASCII code from the temporary memory is read.

Thus the compressed data has been stored in the permanent memory.

5. To retrieve the data from the permanent memory, we read bitwise data from it. We continue serially shifting through the data until a match with a Huffman code is found. Once a match is found, we output the ASCII of the corresponding character using a 3 to 8 ASCII encoder. (Circuit C4)

**Note:* We assume a simplified case of all frequencies of occurrences being distinct. Accordingly, for n distinct characters, the character occurring the maximum number of times is assigned the code 0, the next one is assigned 10, the next gets 110, and so on, until the second most rare occurrence gets 1111…110(n bits). The least occurring character gets 1111…1 (n bits).The Huffman code allows to have characters being encoded using the same length of code. Here we assume such a case does not occur for simplicity.

The above circuits have been elaborated in the later sections, and ways to extend the circuits have also been discussed.

WHAT IS HUFFMAN CODING?

A Huffman code is an optimal prefix code commonly used for **lossless data compression** in computer science and information theory. Unlike most coding conventions like the ASCII or BCD that encode characters using fixed-length codes, Huffman code has the added benefit of assigning codes of variable lengths depending on the frequency of occurrence of characters. The characters that occur more frequently are assigned shorter codes. This leads to efficient compression of data and saves storage space. Huffman codes can be easily represented in the form of a Huffman Tree, as has been explained below.

HUFFMAN ALGORITHM

Given a string of data comprised of characters, the following steps are followed to generate the Huffman tree:

- Start with as many leaves, one representing each character.
- Arrange the characters in the descending order of their frequencies of occurrence.
- Repeat the following until all characters have been exhausted and all leaves have been assigned parents.
    - Find two characters with the least occurrence frequencies. Replace these characters by their single concatenation and treat this as a new character, assigning it a frequency equal to the sum of the frequencies of the original two characters.
    - In the Huffman tree, this new concatenated character acts as a parent of the two original characters (the original characters are branches of this concatenation).
- The above code generates a tree, with the most frequent characters near the root of the tree and the least frequent near the leaves.

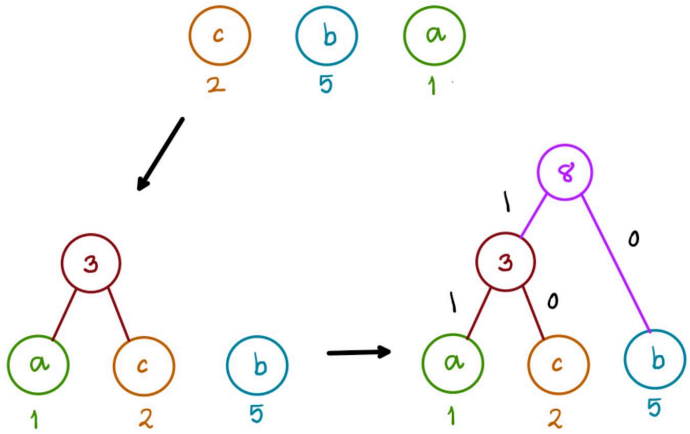
Once the Huffman tree has been generated, it is traversed to generate a dictionary that maps the symbols to binary codes as follows:

A. Start from the root of the tree.

B. If the node is not a leaf node, label the branch to the left child as *0* and the branch to the right child as *1*. Repeat the process at both the left child and the right child.

Example:
Consider the string of characters **cbbcbbab**. “a” occurs once, “b” occurs five times and “c” occurs twice. Thus,

- Frequency of “a” = 1
- Frequency of “b” = 5
- Frequency of “c” = 2

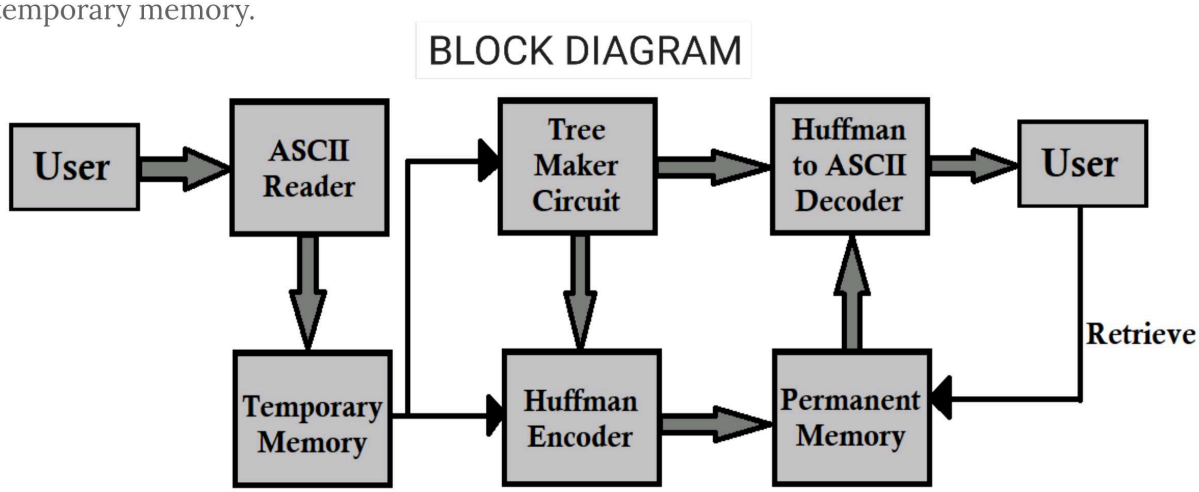


We start with the two least frequent characters, which are “a” and “c”. Add their frequencies and assign that number (1+2 = 3, in our case) to a new node. This generates the tree. The code for “a” is 11, that for “b” to 10, and for “c” is 0. BCD or equivalent coding systems would need 2-bit codes to represent each of the 3 characters.

To store the 8 occurrences of the characters as above, an ordinary system would require 8*2 =16 bits. On the other hand, Huffman encoding requires 1*2 (for ‘a’) + 5*1 (for ‘b’) + 2*2 (for ‘c’) = 11 bits. The space saved becomes prominent when the number of distinct characters to be encoded is larger.

THE PROBLEM STATEMENT AND BLOCK DIAGRAM

Consider a computer that takes inputs from the user in the form of strings composed of the characters "a," "b," and "c." The temporary memory of the computer stores the data in ASCII format. When the user hits the "Save File" option, the data compression system first counts the number of occurrences of each character and generates a Huffman tree. It then performs an ASCII to Huffman code conversion and stores the Huffman equivalent of the string in the permanent memory. To extract data from the permanent memory, the Huffman equivalents of the characters are converted to their ASCII and stored in the temporary memory.



The design of the data compression system is as follows:
This project focuses on the essential internal workings of the four intermediate blocks where all the processing happens, namely:
1) **ASCII Reader** (it is a decoder that reads characters from ASCII)
2) **Tree Maker Circuit** (for a given message in ASCII, it generates the Huffman codes of its characters)
3) **Huffman Encoder** (it stores the message into permanent memory in Huffman code)
4) **Huffman to ASCII Decoder** (Whenever we need our data back, this circuit retrieve it from the permanent memory from Huffman code to ASCII)

The following assumptions have been made while designing the circuits:

1. The Temporary Memory can read out and store one byte (≡ *one character*) in synchronisation with the system's clock.
2. Data will be serially written (bitwise) onto Permanent Memory and will be read out in a similar fashion.
3. All the working of memories are not explicitly shown in the report.

4. We work with three characters only. It suffices to understand the fundamentals of such a circuit since it can be modified to encode any number of bits.
5. All characters have different frequencies of occurrence and if $freq(a_1) < freq(a_2) ... < freq(a_n)$ $then$ $freq(a_n) > \sum_{i=1}^{n-1} freq(a_{n-1})$. This constraint simplifies the structure of our Huffman tree but, at the same time, is general enough to cover a large set of cases.

THE COMPONENT CIRCUITS

1. ASCII READER

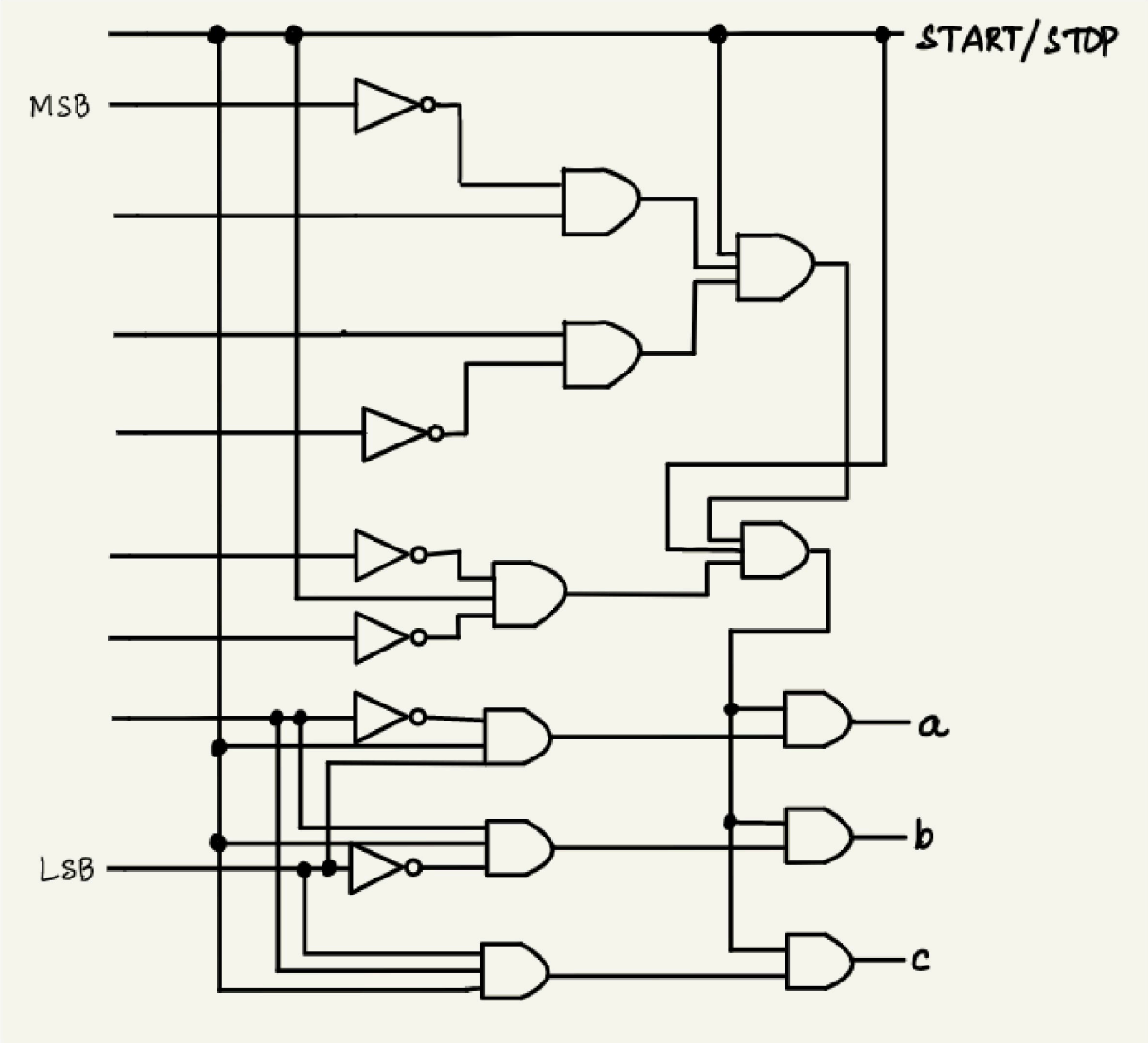
This circuit acts as a 8 to 3 ASCII decoder. It takes ASCII input from the temporary memory (1 byte at a time) and makes the corresponding output pin HIGH. The ASCII codes are as follows:

- a ≡ 01100001
- b ≡ 01100010
- c ≡ 01100011

The circuit action begins when the START/STOP line becomes HIGH. Let's analyse the situation case by case:

- When the decoder gets 01100001 ("a") as its input, the top most line on the output side becomes HIGH. This line is labelled "a".
- Similarly, when the input is 01100010 ("b"), the second line from the top becomes HIGH. This line is labelled as "b".
- When the input is 01100011 ("c"), the bottom-most line becomes HIGH. We call this line "c".

The ASCII code is assumed to be read in chunks of 8 bits,by a clock of frequency CLK_1 and accordingly short width pulses appear on the the lines corresponding to characters.

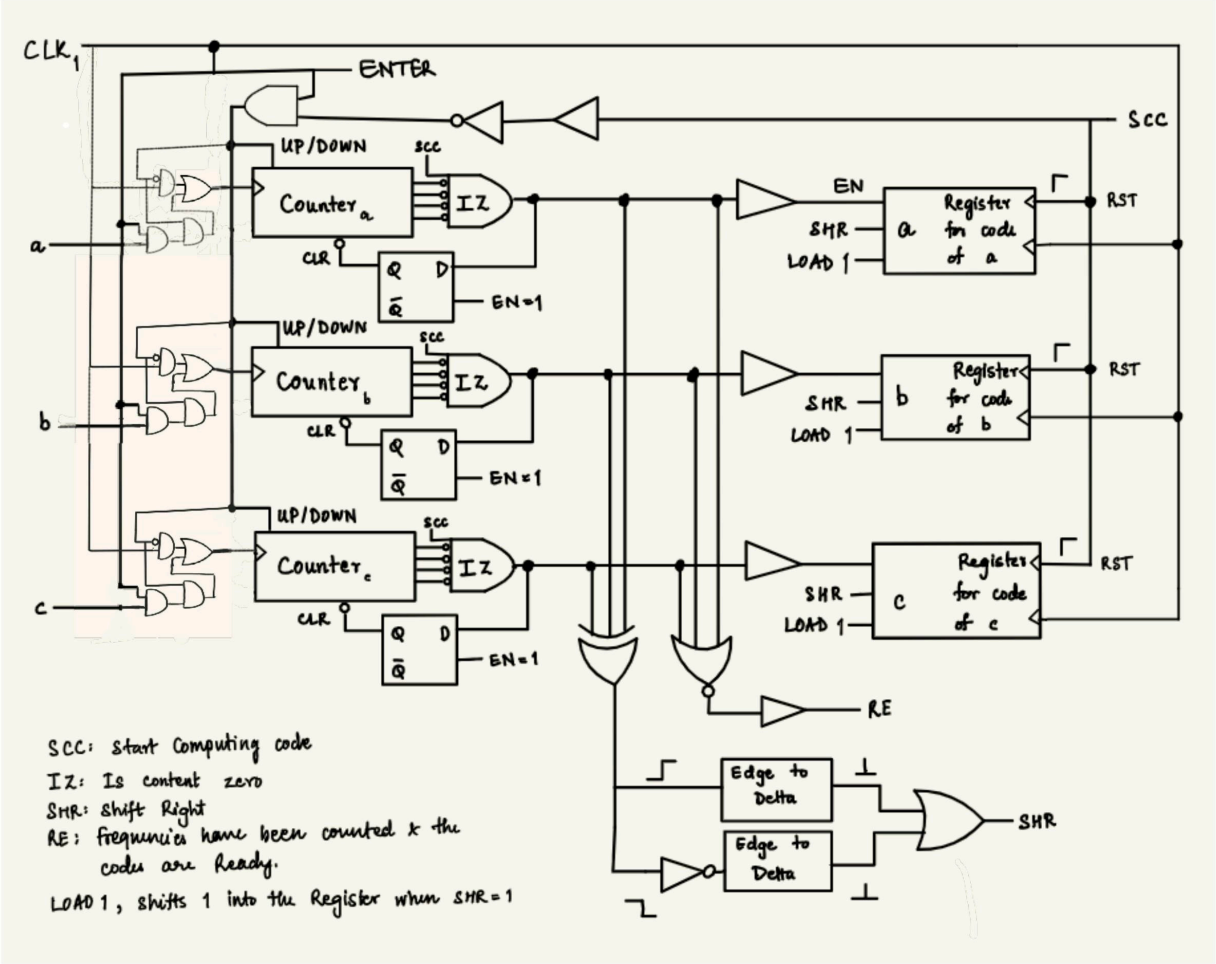


The ASCII equivalent codes are assumed to appear at the lines after a clock frequency equal to that of Circuit 4

Circuit C1

2. TREE MAKER CIRCUIT

The next step in compressing a data file will be the generation of an efficient code-the Huffman code for each character. The circuit shown below is used to make the Huffman tree for the three characters based on their frequencies of occurrence. The inputs line labelled a, b, and c are taken from circuit C1.



Circuit C2

The working of the above circuit can be summarised as follows:

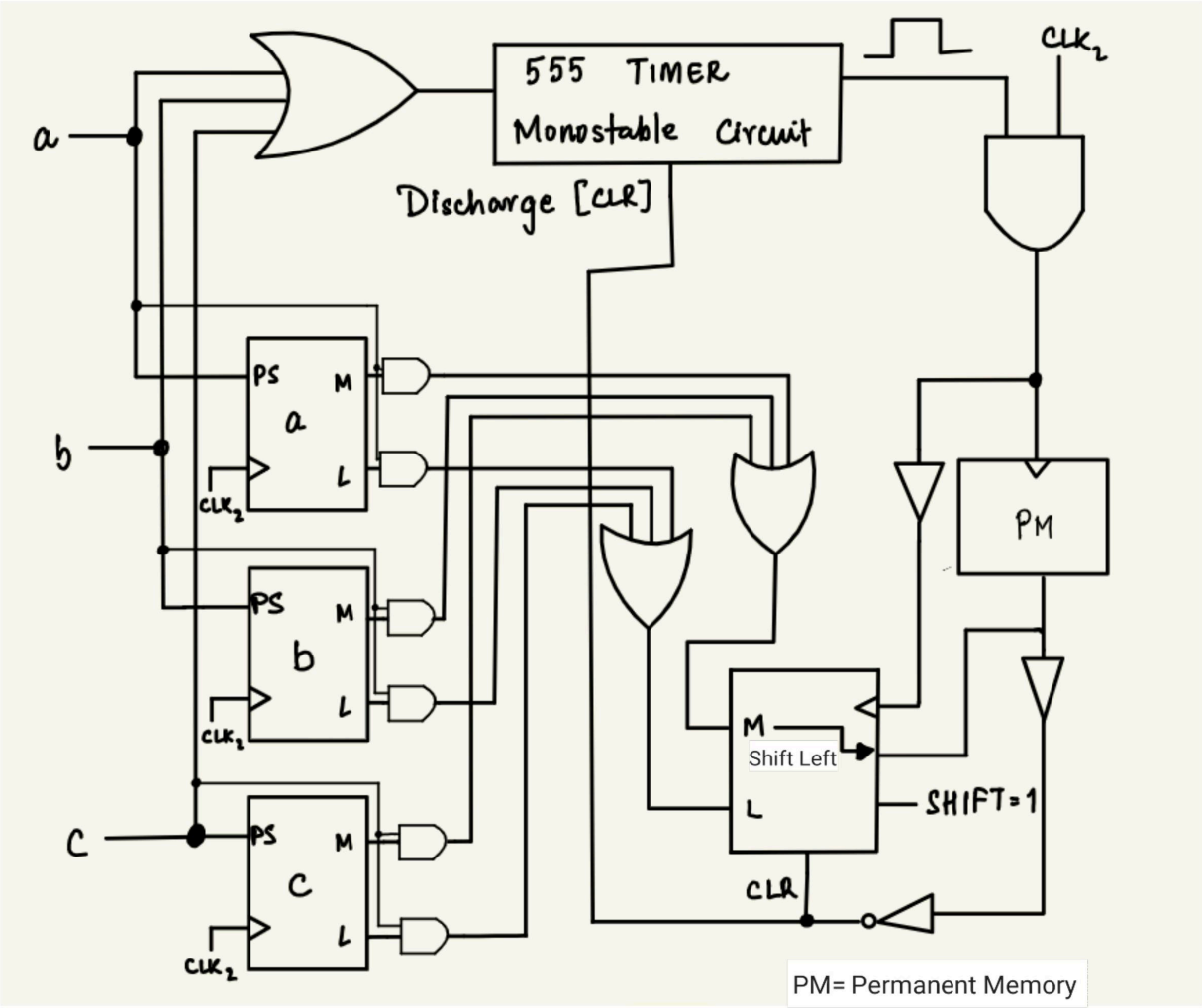
- Lines “a”, “b” or “c” have a high pulse when the corresponding character stored in temporary memory is read by circuit C1. This happens in synchronisation with the system clock of Circuit 1 as long as the enter line is High , and it enables the inputs to the counters.
- As the message (string) is read the corresponding character frequencies are entered in the respective counters.
- Once all frequencies have been stored in the respective counters, the SCC (Start Computing Codes) is set to HIGH. This disables the counters from taking inputs and start down counting with subsequent clock cycles.
- The first counter to become zero will correspond to the character with the lowest frequency. As soon as this happens, the EX-OR will toggle from 0 to 1, generating a pulse output using an edge detector (high pass filter). Simultaneously, when the first counter goes to 0, it is cleared using feedback, and its down counting stops.
- When the next counter (corresponding to the second-lowest frequency) goes to zero while down counting, the EX-OR gate toggles again, generating a pulse once more. So, every time a counter goes to 0, it generates a pulse which is fed to the SHR (shift right) line of each of the three registers for “a”, “b”, and “c”.
- A buffer gate is added to the Enable pin of each resistor to ensure that the SHR is delayed by one pulse for each enabled register.
- This process of finding minimum frequency character continues till all the counters trickle down to 0, and when this happens, all three registers will have the corresponding Huffman codes.

The working of the above circuit can be understood by considering the original example of **cbbcbbab** as the stored message. As the characters are read out of the temporary memory, the respective lines go high whenever a character is detected, thus registering a count in the corresponding counter. Once all frequencies have been counted (1,5,2) the counters start down counting. The circuit functions such that the three registers store the Huffman codes. Here “a” is assigned 11, “c” is 10, and “b” is 00.

The constraint regarding the frequency values (mentioned earlier) ensures that a block can be built to assign just one bit-0 to “a” in this case. Equivalently, while storing into permanent memory we can use optimal Huffman code.

3. HUFFMAN ENCODER

This circuit uses the sequence of characters stored in Temporary Memory in ASCII to the Permanent Memory (PM) in Huffman codes generated from our Tree Making circuit, which is the main aim of our project, i.e., efficiently storing data permanently and retrieving it whenever we want (done in the following circuit).



Circuit C3

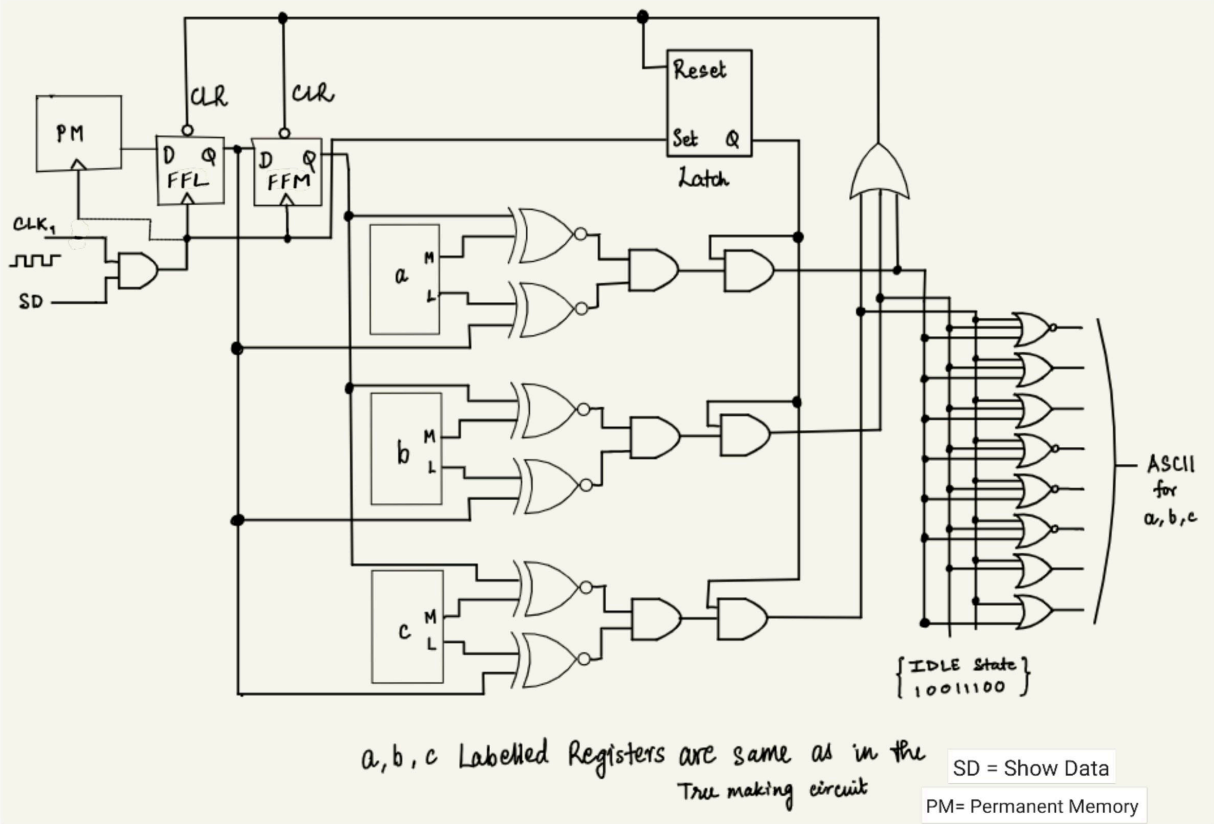
NOTE: Registers labelled a, b and c are 2-bit registers derived from circuit C2. $M \equiv MSB$ *and* $L \equiv LSB$.

Working of the above circuit summarised:

- The “a”, “b”, “c” lines become high whenever the corresponding code is read from the temporary memory. 2 bit registers named “a”, “b”, “c” are the same as the ones in the tree making circuit which contains Huffman codes for respective characters.
- As soon as any of the input lines becomes high, it makes the parallel shift (PS) line of the respective register high and shifts the content to the temporary register.
- Now, the temporary register is ready to shift its contents serially to the permanent memory where they will be stored efficiently. This is done in synchronisation with the positive edge of the clock pulse.
- The clock for the temporary register is a logical AND of two circuits. One is the monostable circuit whose stable state is LOW and it becomes HIGH whenever input comes and remains in that state for 2 cycles of clock 2. This duration of 2 cycles is decided by the maximum bit length of our Huffman codes. This is needed to clear the monostable timer once the maximum length code (which will have all 1s) comes in input and gets transferred to permanent memory.
- All other Huffman codes end with a 0, which will clear the monostable timer once 0 is detected after a delay provided by the buffer to make sure that 0 will also get shifted to permanent memory.

4. HUFFMAN TO ASCII DECODER

The final block of the full circuit uses data stored in registers “a”, “b”, and “c” of Circuit C2 to **Decode** the data stored in the Permanent Memory (PM) in Huffman code into its original form- ASCII.



Circuit C4

NOTE: Registers labelled a, b and c are 2-bit registers derived from circuit C2. $M \equiv MSB$ *and* $L \equiv LSB$.

The functioning of the circuit can be summarised as follows:

- Registers labelled “a”, “b” and “c” are the permanent registers of circuit C2, which store the **Huffman** codes for each of “a”, “b” and “c”.
- PM contains our message in the Huffman encoded form.
- The two D flip-flops after PM compare the incoming bits being read with the code bits stored in the permanent registers..
- If the contents of the F/F do not match any of the codes, the output of the circuit is

in an idle state, which is 1001110, and as soon as the code gets matched, the ASCII equivalent of the concerned bit is seen at the output.

Working of the circuit with an example:

- Data from the PM comes out serially, MSB comes out first, for example, suppose “a” = 0, “b” = 10, “c” = 11, and the message is **aabcc**, then the string stored in PM is 001011, then the serial input to the comparing flip-flops will go as 001011, from left to right in subsequent clock pulses. The data shifted into F/Fs get compared with the codes of “a”, “b”, and “c” using EX-NOR gates.
- Now there are two cases of comparison, one when the code is 0 (1-bit) and the second for all other codes (2-bits). As decided, we stick to such a convention that all the Huffman codes start with 1 except the lowest digit code, which is just 0.
- Whenever we press enter to start decoding bits are serially shifted into the F/Fs with subsequent clock pulses. (bits are shifted from L to M)
- Once any of the codes get matched with the content of the flip-flops, it will make the corresponding outputs of that alphabet visible in ASCII. Simultaneously, it will reset the latch and preset the comparing flip-flops so that the circuit is ready for the next alphabet afresh.

PROSPECTS

The circuit blocks are designed such that they can be extended to encode a larger number of characters. Certain modifications to the tree-making circuit can be thought upon in order to relax the constraints on the frequency distribution. Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others that find optimal prefix codes (while, for example, putting different restrictions on the output). One can explore these as alternatives to the algorithm considered in the project.

REFERENCES

Huffman code, Wikipedia:Huffman coding

Digital Computer Electronics by Albert Paul Malvino and Jerald Brown

Prof Naveen Garg’s Lecture on Data Compression, IITD : Data Compression

GROUP 12: EE224, Digital Electronics

Contribution Statement:

Circuit Design, diagrams, and ideation by:

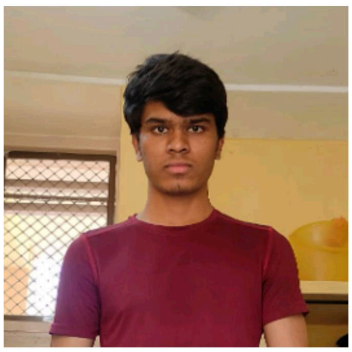
200260023 Kaushik Dhandekar

20d170032 Reet Mhaske

200260049 Shashwat Chakraborty

200260015 Devashish Shah

Report and writeup by:



200260030 Arth Mendhe

200260035 Parmy Parmar