Seung Jun Baek

text	binary code
A	0
B	101
C	100
D	111
E	1101
F	1100

Table 1: Huffman coding table

1 Outline

In this assignment, you are asked to design a verilog module for a simple **Huffman decoder**. First, I will explain what Huffman codes is.

Huffman coding is a method for data compression. Suppose you have a large text file containing alphabet letters. How do you store the file using binary codes? We can use ASCII code to map each alphabet letter into a binary number. Since each letter in a ASCII table takes 7 bits, the size of file is (the number of characters) times (7 bits). For example, if we would like to store a file containing 10,000 alphabet letters, the file size will be 70,000 bits.

But this is usually inefficient, *because not all letters are equal*. For example, the letter "e" occurs most frequently in the English literature. Letters such as "a", "s", "t" are also frequently appearing letters. By contrast, letters such as "q", "z", "j" appears relatively infrequently. So the idea of compression is that, **assign a binary code of small number of bits to frequently appearing letter** (and assign bigger number of bits of binary code to infrequently appearing letters). How to optimally assign a binary code to each letter depending on its occurrence frequency – this is what Huffman codes is about. Some experiments show that Huffman coding can compress English plaintexts at a ratio of 67%.

Now, suppose there is a file with character data. The file contains a sequence of characters, where there are 6 kinds of characters given by A, B, C, D, E and F. It contains 100,000 characters, and the occurrence frequency of characters is given as follows. A: 45 %, B: 13 %, C: 12 %, D: 16 %,E: 9 %, F: 5 %. For example, 45% of the characters in the file are A, and 13% of the characters are B, etc.

Based on this statistics, we can construct **Huffman coding table**, given by Table 1. Table 1 says that, since A occurs most frequently, we should use short code "0" (1 bit) to represent A. By contrast, since E and F occurs least frequently, we should use long code "1101" and "1100" respectively (4 bits) to represent E and F.

Huffman encoding: Huffman encoding is a process of changing input texts to binary codes, given a Huffman coding table. Suppose we are given an input text BACAABEDF, and change this into binary code based on Table 1.

- Input text: BACAABEDF
- According to the table, B is mapped to 101, A is mapped to 0, C is mapped to 100, etc. So the sequence of letters is mapped onto: 101 0 100 0 0 101 1101 111 1100
- Output binary code: 1010100001011101111 1100. This is simply a concatenation of the binary codes for the letters.

Huffman decoding: Huffman decoding is a process of changing input binary codes to output texts, given a Huffman coding table. Huffman decoding is a reverse process of Huffman encoding.

- Input binary code: 10101000010111011111100
- According to the table, 101 is mapped to B, 0 is mapped to A, 100 is mapped to C, etc.
- Output text: BACAABEDF.

2 Objective

The objective of this assignment is as follows. You are given Table 1. Also you will be given input bit sequence which is synchronized to a clock signal. **Your goal is to decode the input bit sequence, that is perform Huffman decoding, and find out the output text.**

For example, using the above example, suppose the input to your module is given by 10101000010111011111100 where each bit is synchronized to the clock. The output of your module should be BACAABEDF (more details later).

3 Specification

3.1 module `huffman_decoder`

Module `huffman_decoder` is a module to carry out Huffman decoding based on Table 1. The input and output signals are given as follows:

- input `x`. 1-bit input signal. This represents the input binary codes for Huffman decoding.
- output `y`. This signal is a 3-bit bus. The output signal must produce values according to Table 2. For example, if you detect the binary code "1101" which represent "E" (according to Table 1), your output should be "101" according to Table 2. The "Null" in Table 2 means that, if you have nothing to output, you should produce 000 at the output – this is called Null.

text	output (y)
Null	000
A	001
B	010
C	011
D	100
E	101
F	110

Table 2: Output codes

- input `clk`. 1-bit input clock signal.
- input `reset`. 1-bit reset signal. If `reset` is 0, your module operates normally. When `reset` goes to 1, **your output should be reset to 3'b000**.

The following are the requirements of this module: **IF YOU DO NOT FOLLOW THE REQUIREMENTS, YOU WILL NOT GET FULL CREDIT, AND MAY EVEN GET ZERO POINTS.**

1. You must design a sequential logic/finite state machine (FSM).
2. FSM must be synchronized to **positive edge** of the clock signal.

4 What to submit

- You must submit a file, named `huffman_decoder.v`, and the file must contain the implementation of modules `huffman_decoder`.
- Upload your file at Blackboard before deadline (no late submission accepted).

5 Comments

- You don't have to find the minimum (optimal) design of your module. That is, **as long as the outputs are correct, the homework will get full credit**.
- You may implement as many submodules as you like, if necessary. In that case, all the modules must be contained within the `huffman_decoder.v` file.

6 How to test your module

In the blackboard, I have uploaded `huff_main.v` so that you can test your module. The file contains `main` module. The `main` module instantiates `huffman_decoder` module, and feeds the test input signal `x` to the module. The test results can be monitored using `gtkwave` tool by looking at output `y` signals.

You can run the following in your command line to compile and simulate the source files.

- `iverilog -o h3.out huff_main.v huffman_decoder.v`
- `vvp h3.out`
- `gtkwave h3_output.vcd`

Below, a more detailed explanation of the testing is provided.

6.1 Modifying `huff_main.v` file

If you look at the `huff_main.v` file, you will see that there are three examples of Huffman-coded texts: FEEDCAFE, DEADBEEF and DECAFACE. The variable `x_test` is the input bit sequence of Huffman code into your Huffman decoder module. That part is given as follows in line 21 of `huff_main.v` file:

```
// INPUT_LEN: 27 answer: FEEDCAFE
//assign x_test = `INPUT_LEN'b1100_1101_1101_111_100_0_1100_1101;

// INPUT_LEN: 26 answer: DEADBEEF
//assign x_test = `INPUT_LEN'b111_1101_0_111_101_1101_1101_1100;

// INPUT_LEN: 23 answer: DECAFACE
assign x_test = `INPUT_LEN'b111_1101_100_0_1100_0_100_1101;
```

If you want to test one of the texts for decoding, you can comment the other two and uncomment the `assign` statement that you want to test. In the above example, a Huffman code for text DECAFACE is used for the testing. The Huffman code is given by

$$11111011000110001001101 \qquad (1)$$

because 111=D, 1101=E, 100=C, 0=A, 1100=F, 0=A, 100=C, 1101=E. You can see that `x_test` is exactly that Huffman code given by Expression (1).

Also, for each of will also see that, in the beginning part of the file

```
//`define INPUT_LEN 27 // for FEEDCAFE
//`define INPUT_LEN 26 // for DEADBEEF
`define INPUT_LEN 23 // for DECAFACE
```

This is called the macro (exactly same as the macro `#define` in C language). The macro is need to properly set the length of the variable `x_test`. For the case of DECAFCE, macro `` `INPUT_LEN `` is set to 23 because (1) is 23 bits.

So if you want to use DECAFACE as your test case, you need to

1. uncomment `` `define INPUT_LEN 23 // for DECAFACE `` and comment the other two
2. uncomment the assignment for `x_test`, which is

   ```
   // INPUT_LEN: 23 answer: DECAFACE
   assign x_test = `INPUT_LEN'b111_1101_100_0_1100_0_100_1101;
   ```

   and comment the other two.

6.2 checking the output

6.2.1 Checking in GTKVIEW

To help with your debugging, `huff_main.v` file contains logic which converts the module output to ASCII characters. This will help you to view your output from the module as readable characters in your GTKVIEW simulation. This can be done as follows.

- In your GTKVIEW window, select the signal `ascii_out` of main module.
- Drag the signal to waveform window.
- Right-click the signal name and select "Data Format" then "ASCII".

In this way, you can view the output as characters A through F. Also the 'Null' output from the module will be shown as the character "–". Refer to the screenshots uploaded at BB.

6.2.2 Checking at command prompt

Also the main module outputs the results to the command prompt. Using `$write` and `$display` commands provided by verilog, we can

If everything is correct, you should see the following output at the command prompt.

```
The answer is
---D---E--CA---FA--C---E
```

Note that there are several "–" between the characters. **IMPORTANT: Your output should exactly match the above output pattern, including "–"s.** Here are the answers for other two cases:

```
The answer is
----F---E---E--D--CA---F---E

The answer is
---D---EA--D--B---E---E---F
```

7 Grading

- 7 points (full) if your module works correctly, that is,
  1. if `huffman_decoder` is correctly designed. That is, the command prompt output in Section 6.2.2 should exactly match the answer.

The rest of case is 0 points, i.e., if you do not submit (or late), or if your file does not compile correctly, or produces wrong results.