MoonCompress

Theoretical background

On the far side of the Moon, we must transmit a lot of data. To save some bandwidth, our engineers came up with a simple compression algorithm. Since we don't want to lose important data, our algorithm is lossless. Like all compression algorithms (especially lossless ones), our lunar one is also based on assigning shorter codes to symbols/characters with a higher probability, i.e. occurring more often in the sequence to be compressed.

Goal

Your task is to create a module that will receive a data packet, store it, compress it using our algorithm, and send it back to the Judge. So, as mentioned earlier, you need to determine the frequency of occurrence of each symbol/character in the input string received from the Judge, then sort them according to decreasing probability of occurrence and then assign the input symbols to the codes from the table (1:1) – Table 2 Lunar Coding Table (see at the end of the document). It may happen that the frequency of occurrence will be the same for some symbols, then we perform additional sorting, and symbols with lower value are placed higher. Once you have established the mapping of input symbols to lunar codes, you can transcode the whole input data. The final task is to send them back to the Judge and wait for the verdict.

In this task, a character is 8-bit wide, so you can encode 256 different characters, but here we limit ourselves to a 64-character subset. Hence, only 64 entries are in the coding table. The judge can generate an input data of any length (max 4096B).

Example 1

Let's suppose you get input data like this: input_data = [3, 9, 3, 2, 4, 2, 9, 9, 1]

Frequency table			After sorting	
Count	Symbol		Count	Symbol
2	3		3	9
3	9		2	2
2	2		2	3
1	4		1	1
1	1		1	4

Now, the code mapping.

Symbol	Code
9	0000
2	0001
3	0010
1	0011
4	0100

And finally, the output data

output_data = [0010, 0000, 0010, 0001, 0100, 0001, 0000, 0000, 0011]

for this input_data = [3, 9, 3, 2, 4, 2, 9, 9, 1]

The output data length is 36 bits (5 bytes), and the value should be placed in the output 16 bits header (MSB goes first). Different symbol codes are marked with colors, unused with 'x'.

len_h	len_l	byte 0	byte 1	byte 2	byte 3	byte 4
00000000	00100100	00100000	00100001	01000001	00000000	0011xxxx

Example 2

Here you have another example of how to prepare data to be sent to the Judge. In this case we have different length of output codes and some of them can cross the byte boundaries.

Let's say we have such output after sorting and code mapping.

output_data = [0010, 10001, 0010,10100, 0000]

In this case we have 5 coded symbols (marked as yellow and green) that are grouped in 8 bits. They are 22 bits in length (unused bits here are marked with 'x'). The length (in bits) of the output data are placed in the first two transmitted bytes (the header), first is MSB then LSB.

len_h	len_l	byte 0	byte 1	byte 2
00000000	00010110	00101000	10010101	00000xx

Example 3

Here we have much longer input.

input_data = [30, 50, 20, 40, 50, 40, 50, 17, 16,15, 14, 13, 12, 11, 11, 12, 13, 14,15,16, 17]

Frequency table			After sorting		
Count	Symbol		Count	Symbol	Code
1	30		3	50	0000
3	50		2	11	0001
1	20		2	12	0010
2	40		2	13	0011
2	17		2	14	0100
2	16		2	15	0101
2	15		2	16	0110
2	14		2	17	0111
2	13		2	40	10000
2	12		1	20	10001
2	11		1	30	10010

And finally, the output data with length of 88bits

```
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--
|len_h   | len_l  | byte 0 | byte 1 | byte 2 | byte 3 | byte 4 | byte 5 | byte 6 | byte 7 |
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--
|00000000 01011000| 10010000 01000110 00000001 00000000 01110110 01010100 00110010 00010001
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--
--+--------+--------+---------+
  | byte 8 | byte 9 | byte 10 |
--+--------+--------+---------+
   00100011 01000101 01100111 |
--+--------+--------+---------+
```

For such input

input_data = [30, 50, 20, 40, 50, 40, 50, 17, 16,15, 14, 13, 12, 11, 11, 12, 13, 14,15,16, 17]

Provided module

You can edit only the *task_12.sv* file. If you wish, you may add other files for supportive submodules.

You only get the outline of the ***task_12*** module. Inputs and outputs are declared as shown below in Table 1.

Table 1 List of inputs and outputs of the task_12 module.

Signal name	Signal type	Bit length
i_clk	Input	1 bit
i_rst	Input	1 bit
i_valid	Input	1 bit
i_first	Input	1 bit
i_last	Input	1 bit
i_data	Input	8-bit
o_valid	Output	1 bit
o_last	Output	1 bit
o_data	Output	8-bit

Input and output interfaces

In one packet the task receives a sequence of data on the *i_data* port. The maximal amount of data is 4KB. The beginning of the data block is marked by the *i_first* signal and the end by the *i_last* signal. Throughout the data transfer, the *i_valid* signal is high.

Task receives data in the way shown in the figure 1below:

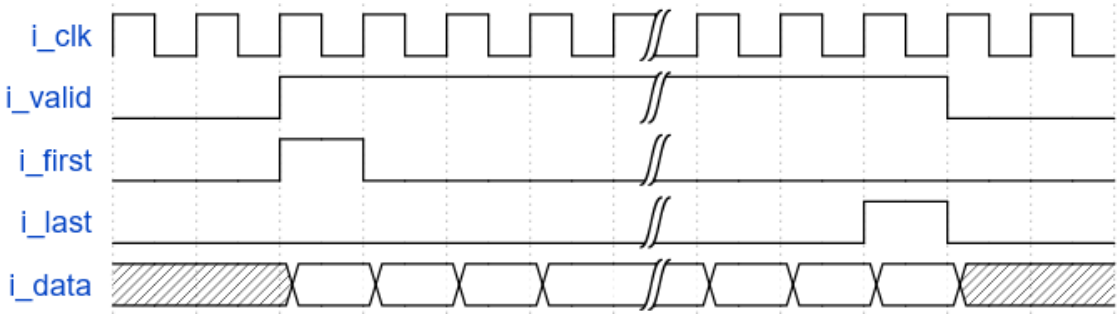

Figure 1 Input waveforms.

The task should return a sequence of data using the *o_data* port. Along with the data the *o_valid* signal should be set, and on the last cycle of data the *o_last* signal should go high.

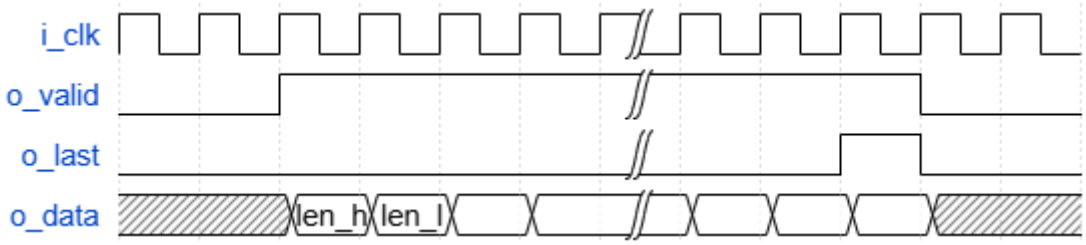

Figure 2 Output waveforms.

Evaluating the task

The evaluation is done by comparing the sequence returned by the task with the expected sequence calculated by the reference model in the Judge. If the sequences are the same, the task is marked as correctly solved.

In Development mode, the task will be tested using a single set of data that is randomized on each run. In Evaluation mode, the task will be tested using 3 fixed sets of data, which remain the same across all runs and are identical for every team (for more information, check chapter 2.3.1 **Testing modes** in the **FPGA_Hackathon_2025_Project_Guide** document).

You earn 7 points for correctly processing each test vector. In Development mode, this value simply indicates that the task passed the test. Evaluation mode, your total base score is calculated by multiplying 7 by the number of test vectors, so you can earn up to 21 points.

Additional points may be awarded for **resource utilization** (for more information, check chapter 2.3.2 **Bonus Points** in the **FPGA_Hackathon_2025_Project_Guide** document).

Lunar Coding Table

Table 2 Lunar Coding Table (from most likely to least likely).

Probability	Code
most probable/often	0000
	0001
	0010
	0011
	0100
	0101
	0110
	0111
	10000
	10001
	10010
	10011
	10100
	10101
	10110
	10111
	1100000
	1100001
	1100010
	1100011
	1100100
	1100101
	1100110
	1100111
	1101000
	1101001
	1101010
	1101011
	1101100
	1101101
	1101110
	1101111
	11100000
	11100001
	11100010
	11100011
	11100100
	11100101
	11100110

	11100111
	11101000
	11101001
	11101010
	11101011
	11101100
	11101101
	11101110
	11101111
	11110000
	11110001
	11110010
	11110011
	11110100
	11110101
	11110110
	11110111
	11111000
	11111001
	11111010
	11111011
	11111100
	11111101
	11111110
least probable/often	11111111