Hamming correction

Theoretical background

Hamming codes are a class of error-detecting and error-correcting codes developed by Richard W. Hamming in the late 1940s. These codes ensure reliable data transmission in systems susceptible to noise and errors by introducing redundancy into the transmitted data, allowing the receiver to detect and correct errors.

Hamming codes are widely used in memory systems, data transmission, and storage devices, such as DRAM, where single-bit error correction is critical. They balance error correction capability and overhead, making them efficient for low-complexity systems.

This task focuses on (7,4) Hamming code where codeword is formed as:

$$c = [d_1, d_2, d_3, p_1, d_4, p_2, p_3]$$

where:

- d_1, d_2, d_3, d_4 are the data bits.
- p_1, p_2, p_3 are the parity bits.

The code can be defined using a parity-check matrix **H**, which checks for errors. The **H** matrix for a (7,4) Hamming code is:

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Each column of **H** corresponds to the binary representation of the bit positions (1 to 7).

To detect and correct errors, the syndrome **S** is calculated as:

$$S = H \cdot c^T$$

Where:

- c^T is the transposed received codeword.
- ***S*** is a 3-bit vector that indicates the position of the error (**S** ≠ **0**).

By solving for ***S***, you can identify and correct single-bit errors in the received codeword.

Goal

Your goal is to **create** an RTL module that receives 8-bit samples, determines if there are errors in them, corrects errors and sends the corrected samples. If there are no errors, the module should send the input samples.

MSB (8^{th} bit) in input samples should be ignored. Also, the 8th bit in output samples will be ignored by the judge during checking of the result.

Only one-bit errors may occur.

Provided module

You can edit only the *task_11.sv* file. If you wish, you may add other files for supportive submodules.

The module has the following inputs and outputs:

Table 1 List of inputs and outputs of the task_11 module.

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

Input and output interfaces

In one packet task receives **from 1 to 1024 samples** of input data (8 input values, 8 bits each). The last sample in a packet is indicated by signal *i_last.* The task should send the same amount of output samples, 8 bits each, and should indicate the last sample in the output packet by providing signal *o_last.*

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

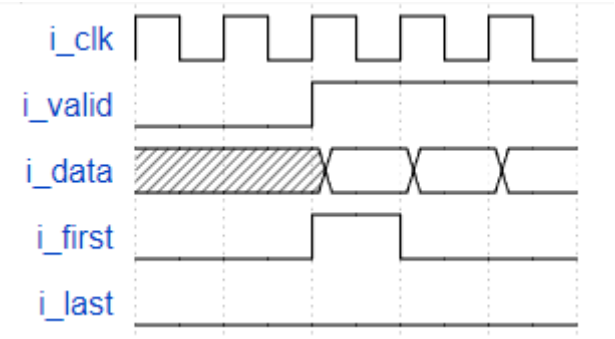

Figure 2 Input waveforms – the start of the packet.

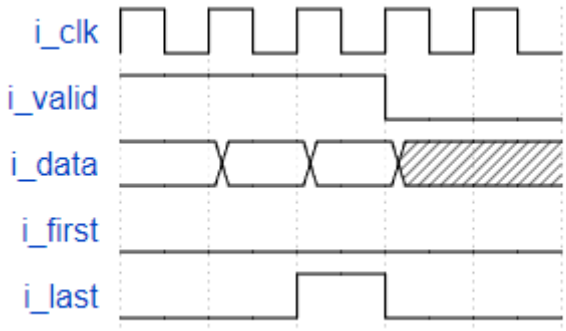

Figure 3 Input waveforms – the end of the packet.

To send data from the task in a proper way one needs to provide the *o_valid* signal that is synchronized with data on the *o_data* bus and o_last signal that is synchronized with the last sample in output packet.

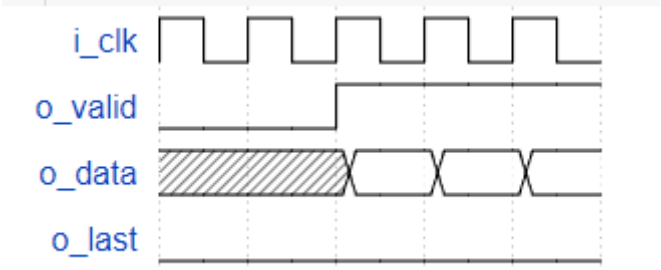

Figure 4 Output waveforms – the start of the packet.

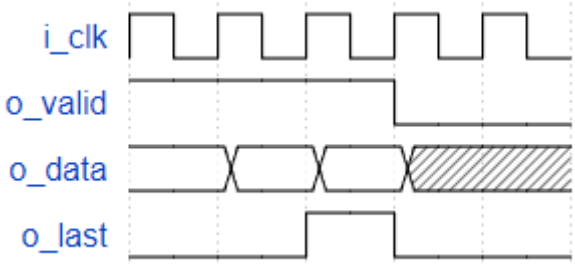

Figure 5 Output waveforms - the end of the packet.

Output data can be sent as a continuous stream, or it can have regular or irregular gaps. The only requirement is providing *o_valid* and *o_last* signals.

Evaluating the task

For the task to be considered correctly performed, the output byte must be bit-exact with the reference value.

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 **5 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 5 by the number of test vectors, so **you can earn up to 15 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).