

FEC (Forward Error Correction) by code Hamming.

Theoretical overview:

a. Calculating the Hamming Code

The key to the Hamming Code is the use of extra parity bits to allow the identification of a single error. Create the code word as follows:

1. Mark all bit positions that are powers of two as parity bits.
(positions 1, 2, 4, 8, 16, 32, 64, etc.)
2. All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
3. Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.
Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc.
(3,5,7,9,11,13,15,...)
Position 2: check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc. (3,6,7,10,11,14,15,...)
Position 4: check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc. (5,6,7,12,13,14,15,20,21,22,23,...)
Position 8: check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, etc. (9-15,24-31,40-47,...)
Position 16: check 16 bits, skip 16 bits, check 16 bits, skip 16 bits, etc. (17-31,48-63,80-95,...)
etc.
4. Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

b. Here is an example

A byte of data: 10011010.

Create the data word, leaving spaces for the parity bits:

_ _ 1 _ 0 0 1 _ 1 0 1 0

Calculate the parity for each parity bit

(a? represents the bit position being set):

- Position 1 (p_1) checks bits 3, 5, 7, 9, 11:

? _ 1 _ 0 0 1 _ 1 0 1 0.

Even parity so set position 1 to a 0: 0 _ 1 _ 0 0 1 _ 1 0 1 0

- Position 2 (p_2) checks bits 3, 6, 7, 10, 11:

0 ? 1 _ 0 0 1 _ 1 0 1 0.

Odd parity so set position 2 to a 1: 0 1 1 _ 0 0 1 _ 1 0 1 0

- Position 4 (p_3) checks bits 5, 6, 7, 12:

0 1 1 ? 0 0 1 _ 1 0 1 0.

Odd parity so set position 4 to a 1: 0 1 1 1 0 0 1 _ 1 0 1 0

- Position 8 (p_4) checks bits 9, 10, 11, 12:

0 1 1 1 0 0 1 ? 1 0 1 0.

Even parity so set position 8 to a 0: 0 1 1 1 0 0 1 0 1 0 1 0

- Code word: 011100101010.

c. Finding and fixing a bad bit

The above example created a code word of 011100101010. Suppose the word that was received was 011100101110 instead. Then the receiver could calculate which bit was wrong and correct it. The method is to verify each check bit. Write down all the incorrect parity bits.

Position 1 (c_1) checks bits 1, 3, 5, 7, 9, 11.

Position 2 (c_2) checks bits 2, 3, 6, 7, 10, 11.

Position 4 (c_3) checks bits 4, 5, 6, 7, 12.

Position 8 (c_4) checks bits 8, 9, 10, 11, 12.

Doing so, you will discover that parity bits 2 and 8 are incorrect. It is not an accident that $2 + 8 = 10$, and that bit position 10 is the location of the bad bit. In general, check each parity bit, and add the positions that are wrong, this will give you the location of the bad bit.

Practical part:

Write the RTL model of 8 bit to 12 bit synchronous parallel encoder and 12 bit to 8 bit synchronous parallel decoder on code Hamming. Stages and the task:

1. Encoder RTL.
2. Decoder RTL.
3. Co-simulation of encoder and decoder (Test bench).
4. TB of joint model provides the possibility of random errors in the data stream between encoder and decoder.

Initial requirements

Encoder side:

- Clock frequency is 50 MHz
- Asynchronous reset
- Synchronous data input and data enable
- Synchronous data output with data valid signal

Decoder side:

- Clock frequency is 50 MHz
- Asynchronous reset
- Synchronous data input and data enable
- Synchronous data output with data valid and error flag.

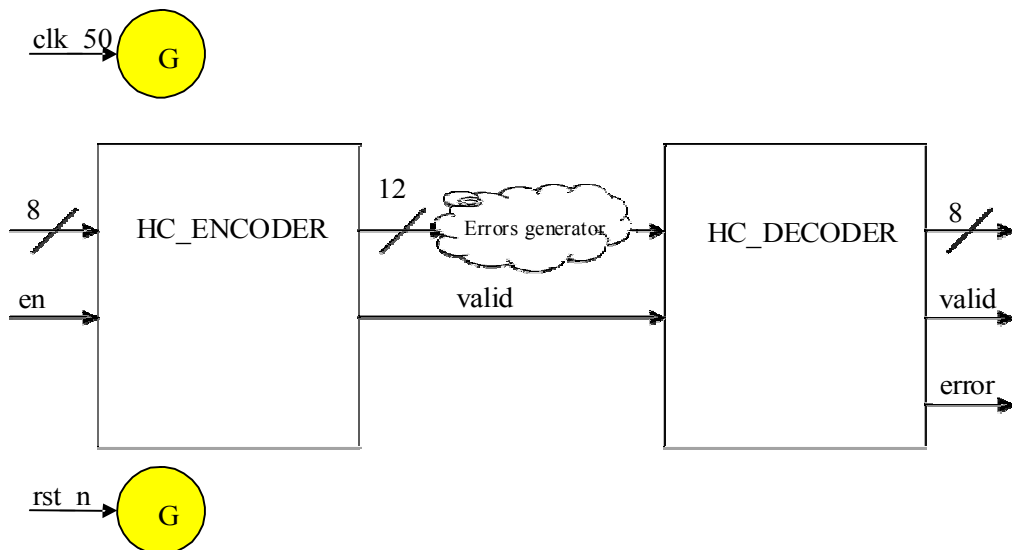


Fig 1:Test_bench block diagram (G – global resource for all devices)

Where HC_ENCODER and HC_DECODER are VHDL components, and Error generator is part of test bench (see example).

Random error injection example.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.math_real.all;
-----
random_error_injection: process is
    variable seed1 : integer :=100;
    variable seed2 : integer :=105;
    variable rand_v : real;
    variable bit_position : integer range data_bus'high downto 0 := 0;
begin
    wait on data_from_encoder;
    for i in 0 to N loop -- where N is integer constant
        uniform(seed1, seed2, rand_v); -- from math_real library
        -- returns value of real type with range 0.0 to 1.0
        -- needs updating of seed1 and seed2 values for each call
        bit_position:= integer(real(data_bus'high) * rand_v);
        data_to_decod <= data_from_encoder;
        data_to_decod(bit_position)<=not data_from_edcoder(bit_position);
        wait until rising_edge(clk);
        seed1 := seed1 + 1;
        seed2 := seed2 +1;
    end loop;
    wait;
end process random_error_injection;
```