

Implementation of Hamming code using VLSI

Nutan Shep¹, Mrs. P.H. Bhagat²

Department of Electronics & Telecommunication

Dr.B.A.M.U,Aurangabad

Government College of Engineering

Aurangabad (M.S.),India-431005

Abstract-This paper tries to explain the implementation of hamming code using VLSI. In the present world the field of communication has got many applications, and in every field the data is encoded at the transmitter and transferred on a communication channel and received at the receiver after it is decoded. During the transmission of data it might get corrupted because of some noise on the channel. So it is necessary for the receiver to have some function which can detect the error in the received data. Hamming code is one of such forward error correcting code which has got many applications. In this paper the algorithm for hamming code is discussed and then implementation of it in verilog is done to get the results. Hamming code is an improvement over parity check method. Here a code is implemented in verilog in which 4-bit of information data is transmitted with 3-redundancy bits. In order to find the value of these redundancy bits a code is written in verilog which will be simulated in Xilinx 9.1 software. The result of simulation and test bench waveforms are also shown.

Keywords: VLSI, verilog, xilinx, redundancy, parity, Hamming.

I. INTRODUCTION

One of the major issues in the field of communication is the secure and error free transmission of data from transmitter to receiver. And for error free transmission there are number of technologies. One of the technologies which is used for correcting forward error is the Hamming code technology. Around 1947 Richard W. Hamming developed this technology to detect and correct single bit errors in transmitted data. In Hamming code error detection and correction technique to get error free data at destination, we encrypt information data according to even and odd parity method before transmission of information at source end.[3][1]

In telecommunication, Hamming codes are a family of linear error correcting codes. Hamming codes can detect upto two and correct upto one bit errors. By contrast the simple parity code cannot correct errors, and can detect only odd number of errors. Hamming code are special in that they are perfect codes, that is they achieve the highest possible rate for codes with their block length and minimum distance. Because of the simplicity of hamming codes, they are widely used in computer memory. In this context one often uses an extended hamming code with one extra parity bit. Extended hamming code achieve a distance of 4, which allows the decoder to distinguish between the situation in which at most one bit error occurred and the situation in which two bit error

occurred. In this sense, extended hamming codes are single error correcting and double-error detecting and often referred to as SECDED.[3][4]

In this paper we have written Verilog code for finding error location and correct the bit which is corrupted. At the destination, we receive 7-bit of data with 4 redundancy bits. This received data may be corrupted due to noise. To remove this noise we find the address of the error bit then correct them. To find the location of error bit and correct them we write code in Verilog language. This paper is organized as follows: the concept of hamming code along with the application of it, Verilog language, Why Verilog is preferred over VHDL, Implementation of hamming code, Performance and experimental results, conclusion and references.

II. HAMMING CODE

Hamming code are the linear block code which are invented by Richard.W. Hamming. They are an improvement over simple parity code method. Hamming codes are valid only when the hamming distance between the bits is less then or equal to one. By contrast, the simple parity code cannot correct errors, and can only detect an odd number of errors. They are the type of binary codes. The idea of hamming distance is the central concept in coding the error control. The hamming distance between the two words (of the same size) is the number of differences between the corresponding bits. The hamming distance can easily be found if we apply the Xor operation on the two words and count the number of 1s in the result. The hamming distance is a value always greater then zero. If we find the hamming distance between any two words it will be the result of the Xoring of the two bits. Like the hamming distance between d (000,011) is 2 because 000 xor 011 is 011(two 1s) and the hamming distance between d(10101,11110) is 3 because 10101 xor 11110 is 01011(three 1s).[3][7]

Hamming code method works only on two methods(even parity, odd parity) for generating redundancy bits. The number of redundancy bits are generated using a formula. The number of redundancy depends on the number of information data bits. The formula is:

$$2^r = D+r+1 \dots \dots \dots (1)$$

Here, r = number of redundancy bits

D = number of information data bits

If we calculate the number of redundancy bits for an 4 bit of information then it comes to be 3 redundancy bit. Redundancy bits are those extra bits which are required to detect and correct errors. The redundancy bits are added to the information bit at the transmitter and removed at the receiver. The receiver is able to detect the error and correct it because of the redundancy bits. Hamming codes are used as forward error correcting codes in the Bluetooth standard, and to protect data stored in semiconductor memories. Hamming codes are generally used in computing, telecommunication, and other applications including data compression, and turbo codes. They are also used for low cost and low power applications.

III. VERILOG LANGUAGE

Verilog is one fo the two most common Hardware Description languages(HDL) used by integrated circuits(IC)designers. The other one is VHDL. Verilog can be used to describe designs at four levels of abstraction. They are the algorithmic level, the register transfer level, the gate level and the switch level.[6]

A. USING VERILOG FOR SIMULATION

The basic structure of Verilog in which all hardware components and testbenches are described is called a module. Language constructs, in accordance to Verilog syntax and semantics form the inside of a module. These constructs are designed to facilitate the description of hardware components for simulation, synthesis, and specification of testbenches to specify test data and monitor circuit responses. A module that encloses a design’s description can be described to test the module under design, in which case it is regarded as the testbench of the design.

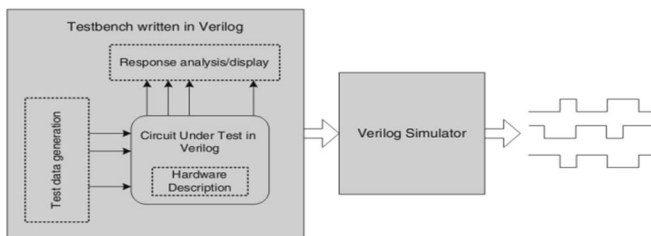


Figure 1 Simulation in verilog

The above model consists of a design with a verilog testbench. Verilog constructs (shown by dotted lines) of the Verilog model being tested are responsible for the description of its hardware, while language constructs used in a testbench are in charge of providing appropriate input data or applying data stored in a text file to the module being tested, and analysis or display of its outputs. Simulation output is generated in the form of a waveform for visual inspection or data files for record or for machine readability.[5][6]

B. USING VERILOG FOR SYNTHESIS

After a design passes basic the functional validations, it must be synthesized into a netlist of components of a target library. The target library is the specification of the hardware that the design is being synthesized to. Verilog constructs used in the Verilog description of a design for its verification or those for timing checks and timing specifications are not synthesizable. A verilog design that is to be synthesized must use language constructs that have a clear hardware correspondence. The figure below shows a block diagram specifying the synthesis process. Circuit being synthesized and specification of the target library are the inputs of a synthesis tool. The outputs of synthesis are a netlist of components of the target library, and timing specification and other physical details of the synthesized design. Often synthesis tools have an option to generate this netlist in Verilog.

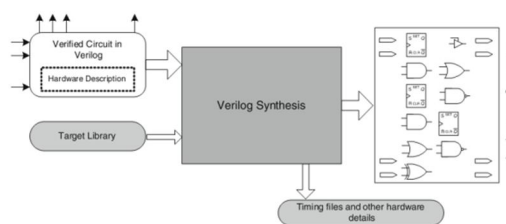


Fig 2 synthesis of a verilog design

C. POST-SYNTHESIS SIMULATION

When the netlist is provided by the synthesis tool that uses verilog for the description of the netlist components(Fig 1), the same testbench prepared for the pre-synthesis simulation can be used with this gate-level description. This simulation, which is often regarded as post-synthesis simulation, uses timing information generated by the synthesis tool and yields simulation results with detailed timing. Since the same testbench of the high-level design is applied to the gate-level description, the resulted waveform or printed data must be the same.[5][6]

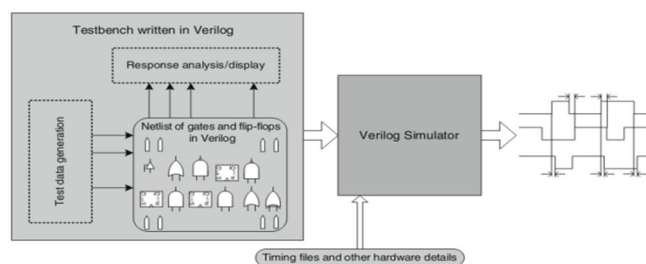


Fig 3 Post synthesis simulation

This can be seen when comparing Fig 1 and 3, while the only difference is that the post-synthesis simulation includes timing details.

IV.WHY VERILOG PREFERRED OVER VHDL

We are preferring verilog language over VHDL because of the

following reasons.

1. Verilog means verification of logic. It is built in C mostly easy to learn.
2. It gives the simple and effective way of describing the digital circuits. It is intended for modeling, simulation and analysis.
3. They are highly portable and self documenting .
4. They provides many descriptive style like structural, register transfer level and behavioral.
5. Design described in HDL are technology independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.
6. The way the code is written will greatly effect size and speed of the synthesized circuit.[5][8]

V. IMPLEMENTATION OF HAMMING CODE

In this section , the design of hamming code encoder and decoder in Verilog language will be done. Here we have used the (7,4) algorithm. The algorithm is called a (7, 4) code, because it requires seven bits to encoded four bits of data. The extra three bits are parity bits. Each of the three parity bits are parity for three of the four data bits, and no two parity bits are for the same three data bits. All of the parity bits are even parity. Example:

Given: data bits d1, d2, d3, and d4

A (7, 4) Hamming code may define parity bits p1, p2, and p3 as

$$p1 = d2 + d3 + d \tag{2}$$

$$p2 = d1 + d3 + d4 \tag{3}$$

$$p3 = d1 + d2 + d4 \tag{4}$$

There's a fourth equation for a parity bit that may be used in Hamming codes:

$$p4 = d1 + d2 + d3 \tag{5}$$

Valid Hamming codes may use any three of the above four parity bit definitions. Valid Hamming codes may also place the parity bits in any location within the block of 7 data and parity bits. Two Hamming codes with different parity bits or parity bits in a different bit position are considered equivalent. They will produce different results, but they are still Hamming codes. One method for transforming four bits of data into a seven bit Hamming code word is to use a 4x7 generator matrix [G].[2][4]

Define d to be the 1x4 vector [d1 d2 d3 d4]

It's possible to create a 4x7 generator matrix [G] such that the product modulo 2 of d and [G] (d[G]) is the desired 1x7 Hamming code word. Here's how it's done:

Step 1. Represent each data bit with a column vector as follows:

$$d1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$d2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$d3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$d4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Step 2. Represent each parity bit with a column vector containing a 1 in the row corresponding to each data bit included in the computation and a zero in all other rows. Using the parity bit definitions from the example above:[7][2]

$$p1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$p2 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$p3 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Step 3. Create a generator matrix, [G], by arranging the column vectors from the previous steps into a 4x7 matrix such that the columns are ordered to match their corresponding bits in a code word. To create a generator that produces code words with the bits ordered p1, p2, p3, d1, d2, d3, d4 (3 parity bits followed by 4 data bits) use the vectors from the previous steps and arrange them into the following columns [p1 p2 p3 d1 d2 d3 d4] .[1][2]

The results are following 4x7 generator matrix:

$$p1 \ p2 \ p3 \ d1 \ d2 \ d3 \ d4$$

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

Arranging the columns in any other order will just change the positions of bits in the code word.

Example:

Encode the data value 1010 using the Hamming code defined by the matrix G (above).

$$[1 \ 0 \ 1 \ 0] * \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$[1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0]$$

So 1010 encodes to 1011010. Equivalent Hamming codes represented by different generator matrices will produce different results.

In a world without errors decoding a Hamming code word would be very easy. Just throw out the parity bits. The encoding example produced a 7 bit code word. Its parity bits are 101 and its data bits are 1010. If you receive a 1011010, just decode it as 1010. But what happens if you receive a code word with an error and one or more of the parity bits are wrong?[7][1]

Suppose the Hamming code defined by the matrix G in the example above is being used and the code word 1011011 is received. How is that word decoded? The first step is to check the parity bits to determine if there is an error.

Arithmetically, parity may be checked as follows:

$$p1 = d2 + d3 + d4 = 0 + 1 + 1 = 0$$

$$p2 = d1 + d3 + d4 = 1 + 1 + 1 = 1$$

$$p3 = d1 + d2 + d4 = 1 + 0 + 1 = 0$$

In this case every parity bit is wrong. p1, p2, and p3 should have been 010, but we received 101. Parity may also be validated using matrix operations. A 3x7 parity check matrix [H] may be constructed such that row 1 contains 1s in the position of the first parity bit and all of the data bits that are included in its parity calculation. Row 2 contains 1s in the position of the second parity bit and all of the data bits that are included in its parity calculation. Row 3 contains 1s in the position of the third parity bit and all of the data bits that are included in its parity calculation.

Example:

Using the code from example above, the matrix H may be defined as follows:

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

Multiplying the 3x7 matrix [H] by a 7x1 matrix representing the encoded data produces a 3x1 matrix called the "syndrome". There are two useful properties of the syndrome. If the syndrome is all zeros, the encoded data is error free. If the syndrome has a non-zero value, flipping the encoded bit that is in the position of the column in [H] that matches the syndrome will result in a valid code word.[2]

Example: Using the parity check matrix from the example above we can correct and verify the code word 1011011.

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

A column of all 1s is not the column of all 0s, so there is a parity error. Looking back at the matrix [H], you will see that the seventh column is all 1s, so the seventh bit is the errored bit. Changing the seventh bit produces the code word 1011010.

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

Sure enough 1011010 is a valid code word. As I stated at the top of this section remove the parity bits to get the encoded value. In this case 1011011 was likely transmitted as 1011010, which encodes 1010.

VI. PERFORMANCE AND EXPERIMENTAL RESULT

We have written a code in verilog for hamming[7,4] code. The code is running successfully. In this code we are initializing a data of 4 bit and calculating its parity bit by the above general algorithm. It requires to calculate the parity bit by matrix multiplication and modulo-2 addition. The parity bit required for four bit data is three bit. These bits are calculated and the code is written using case statement.[5][6]

We have done encoding and decoding of the data in the same code. A variable syndrome is used to decode the data and to calculate the parity bit. Syndrome is calculated by multiplying a 3x7 matrix[H] by a 7x1 matrix representing the encoded data produces a 3x1 matrix, called 'syndrome'. The syndrome has two useful properties. First if the syndrome is all zeros,

the encoded data is error free. But if the syndrome has a non-zero value, then flip the encoded bit that is in the position of the column in [H] that matches the syndrome will result in a valid code word.[6]

Before viewing the test bench waveform we will show the input/output ports of hamming encoder and decoder.

TABLE 1
INPUT/OUTPUT PORTS

Signal	Description
Data_in	Input data
Edc_in	Calculates the parity bit
Data_out	Outputs data
error	It is 0 when syndrome is 0
syndrome	Connection between hardware elements

The test bench waveform below shows that we have given an input data of 8 bit. The data is 01010101 i.e 8'h55 in hexadecimal. The figure below shows only the value of the data.

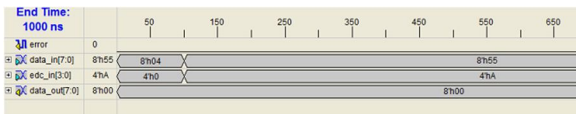


Fig 4 Test bench waveform

The figure below shows the in detail bits of input data. Like the 8th bit contains 0, 7th bit contains 1, 6th bit contains 0 and so on.

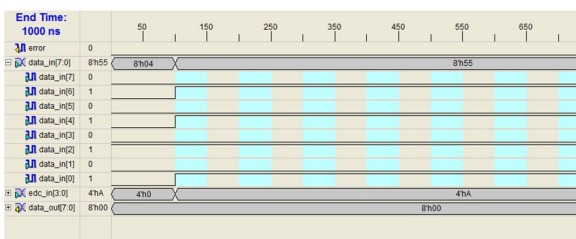


Fig 5 Input data

In the next figure we are giving the bits to edc_in variable which helps in calculating the parity bits. The variable edc_in contains 1010 i.e 4'hA in hexadecimal.

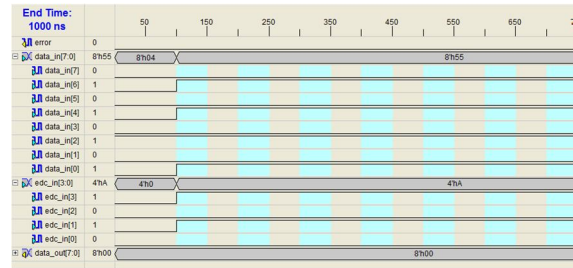


Fig 6 Given output data

The figure below gives the simulation data result of the test bench waveform. It shows that when we send the input data 8'h55 to the input variable the syndrome calculates the parity bit and transmits the data bit and in the result we get output data bit as 8'h55.

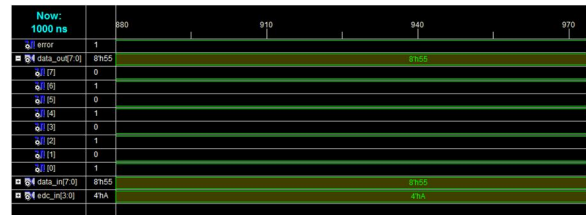


Fig 7 simulation result

VII. CONCLUSION

Thus we have tried to implement hamming code in verilog language and shown the output results. Hamming codes have improved the way of communication by detecting and correcting errors. We have also given the reason that why we are preferring verilog language over VHDL.

ACKNOWLEDGEMENT

I gratefully acknowledge all the people who assisted me during this work. It is a pleasure to thank to all the members who encouraged me for this journal presentation, whose words are great encouragement to me.

I enjoyed and learnt many more things from this event. Finally, I will welcome all the suggestions from anybody which will definitely be nice one. If you found any mistake or error please suggest for the same.

Finally, I gratefully acknowledge to all Institute members because of them I got inspiration to write and present this journal in this event.

REFERENCES

- [1] Data communication and networking, Behrouz A. Forouzan, 4th edition, Tata McGrawHill publication.
- [2] [http://hamming\(7,4\)code](http://hamming(7,4)code) Discussion and implementation.
- [3] Hardware implementation of a single bit error code correction by constant IANA, Gheorghe SERBAN, Ion TUTANESCU, Petre ANGHE-LESCU.
- [4] Information Theory Coding and Cryptography by Ranjan Bose.
- [5] <http://www.xilinx.com/training/xilinx-training-courses.pdf>.
- [6] Verilog HDL: A guide to digital design and synthesis, second edition by Sameer Palnitkar.
- [7] Hamming W. Richard. Coding and information and theory, Prentice-Hall chapter 3.
- [8] ISE 9.1 Quick Start Tutorial, available at <http://www.xilinx.com/itp/xilinx9/books/docs/qst/q>.