Hamming Code

- Harsh Rajiv Agarwal
B19EE036

What is Error Correcting Code ?

When bits are transmitted over the computer network, they are subject to get corrupted due to interference and network problems. The corrupted bits leads to spurious data being received by the receiver and are called errors.

Error-correcting codes (ECC) are a sequence of numbers generated by specific algorithms for detecting and removing errors in data that has been transmitted over noisy channels. Error correcting codes ascertain the exact number of bits that has been corrupted and the location of the corrupted bits, within the limitations in algorithm.

ECCs can be broadly categorized into two types:

1. Block codes – The message is divided into fixed-sized blocks of bits, to which redundant bits are added for error detection or correction. e.g. Reed-Solomon Code, Golay, BCH, Multidimensional parity, and Hamming codes

2. Convolutional codes – The message comprises of data streams of arbitrary length and parity symbols are generated by the sliding application of a Boolean function to the data stream.

What is Hamming Code ?

Hamming code is a block code that is capable of detecting up to two simultaneous bit errors and correcting single-bit errors. It was developed by R.W. Hamming for error correction.

Hamming codes are a class of binary linear codes. It is executed by encoding the original message with a series of redundant bits in positions of the powers of two and the number of occurrences of these extra bits is determined according to an inequality. This specific positioning makes it possible to detect and correct the errors of transmission i.e the receivers of the message can perform recalculations to detect errors and find the bit position of the erroneous bit if it exists. Here, redundant bits are called as parity bits.

This scheme involves two main steps:

1. Encoding
2. Decoding (Error detection and correction)

Encoding

This process consists of two main steps:

1. **Calculation of the number of parity bits:** Let the message contain ‘m’ bits. We add ‘p’ parity bits to this message in order to create a codeword of ‘m+p’ bits. The value of ‘p’ is decided by the following inequality: $2^p \geq m + p + 1$. The idea behind this inequality is that an error in any of the m+p bit positions is denoted by m+p. An additional bit is added which denotes the no error condition. Now, ‘p’ bits can indicate 2^p states, 2^p must be at least equal m+p+1.

2. **Calculating the values of each parity bit & Positioning them:** The above ‘p’ parity bits are placed at positions that are powers of 2 (i.e. 1, 2, 4, 8, 16, 32, 64...). This can be implemented in two configurations:

    1. Even parity - The total number of 1’s in the relevant binary position is made even.

    2. Odd parity - The total number of 1’s in the relevant binary position is made odd.

Encoding

If we follow even parity, P1 is given "1" if the number of 1's in all data bits in positions of the least significant positions is even (3, 5, 7, 9, 11 and so on). P1 is given "0" for the opposite arrangement.

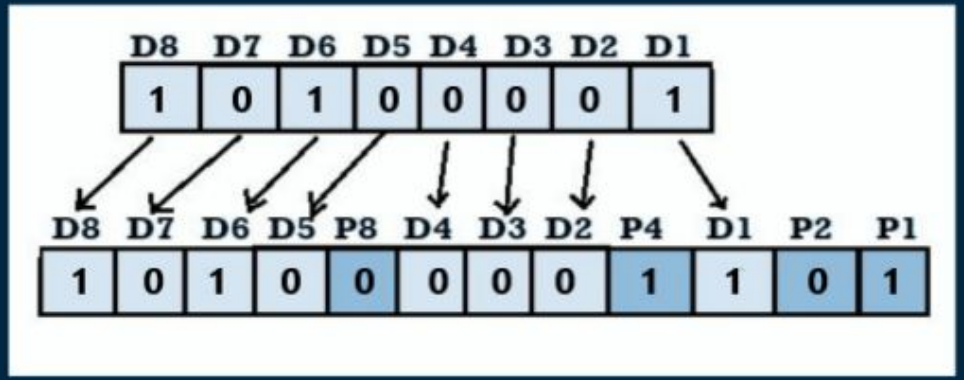
P2 is given "1" if the number of 1's in all data bits in positions of the second least significant positions is even (3, 6, 7, 10, 11 and so on). P2 is given "0" for the opposite arrangement

P4 is given "1" if the number of 1's in all data bits in positions of the third least significant positions is even (5–7, 12–15, 20–23 and so on). P4 is given "0" for the opposite arrangement.

All parity bits are calculated using XOR Gates.

The image alongside explains the encoding

scheme.



Decoding

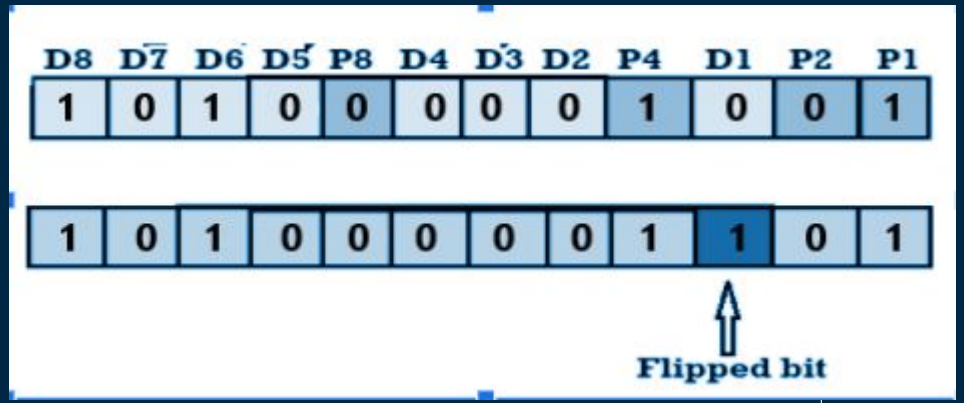
As mentioned before the p number of redundant bits are placed at positions of powers of 2 (1, 2, 4, 8, 16, 32, 64….). Using the same formula as in encoding, the number of redundant bits are placed. 2^p ≥ m + p + 1.

C1 = parity(1, 3, 5, 7, 9, 11 and so on)

C2 = parity(2, 3, 6, 7, 10, 11 and so on)

C3 = parity(4–7, 12–15, 20–23 and so on)

All parity bits are calculated using XOR Gates.



Identifying the Position of the Error:

By following the same method we can calculate XOR of parity positions and their value. We then take the bits of the results of these and take the decimal equivalent. If it is 0, that implies that there is no error. Otherwise, the decimal value gives the bit position which has error. For example, if C4.C3.C1.C1 = 0111, it implies that the data bit at position 7, decimal equivalent of 0111, has error. So the erroneous bit must be flipped to get the correct message.

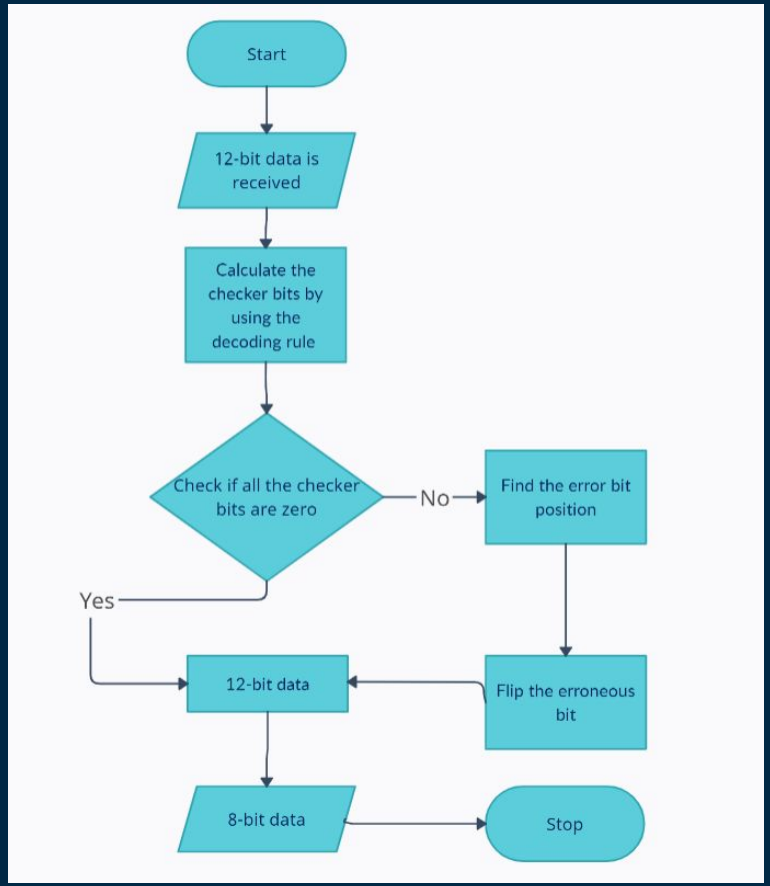
Algorithm

The Hamming Code is simply the use of extra parity bits to allow the identification of an error:

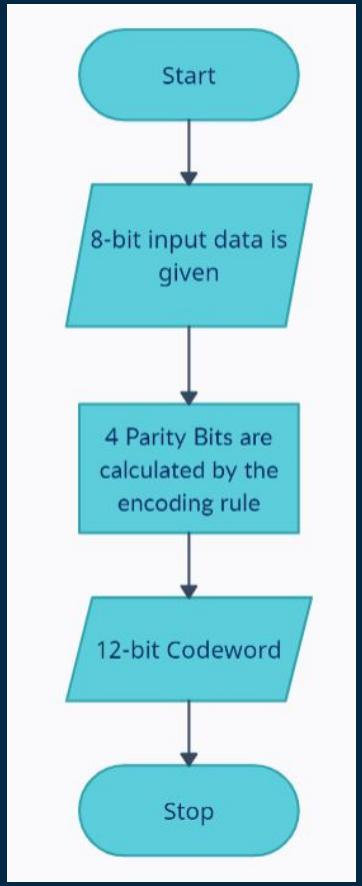
1. Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).
2. All the bit positions that are a power of 2 are marked as parity bits (1, 2, 4, 8, etc).
3. All the other bit positions are marked as data bits.
4. Since we check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd.
5. Set a parity bit to 0 if the total number of ones in the positions it checks is even.
6. Now, checker bits are calculated. For calculating C1, we use count one and skip one method which implies starting from P1, we skip P2 and count D1 and so on. We then check the parity of all these bits which is C1 itself.
7. Similar strategy is used for calculating C2, C3 and C4.
8. If every checker bit is 0, this is the no error condition. This implies that the transmitted codeword is free of errors.
9. In contrast, we arrange them the checker bits in descending order(C4C3C2C1) and the position formed is corrected by flipping the state

Flowchart

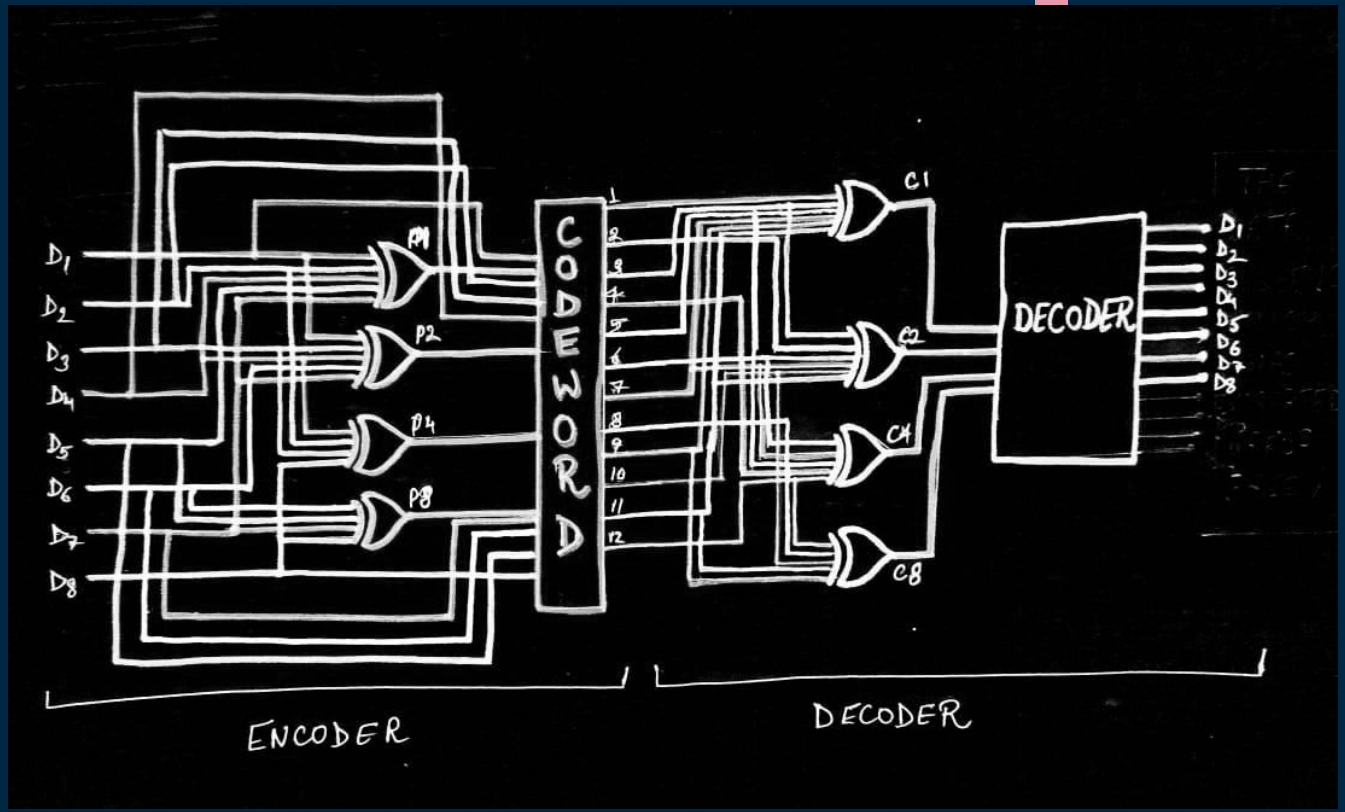
DECODING SCHEME



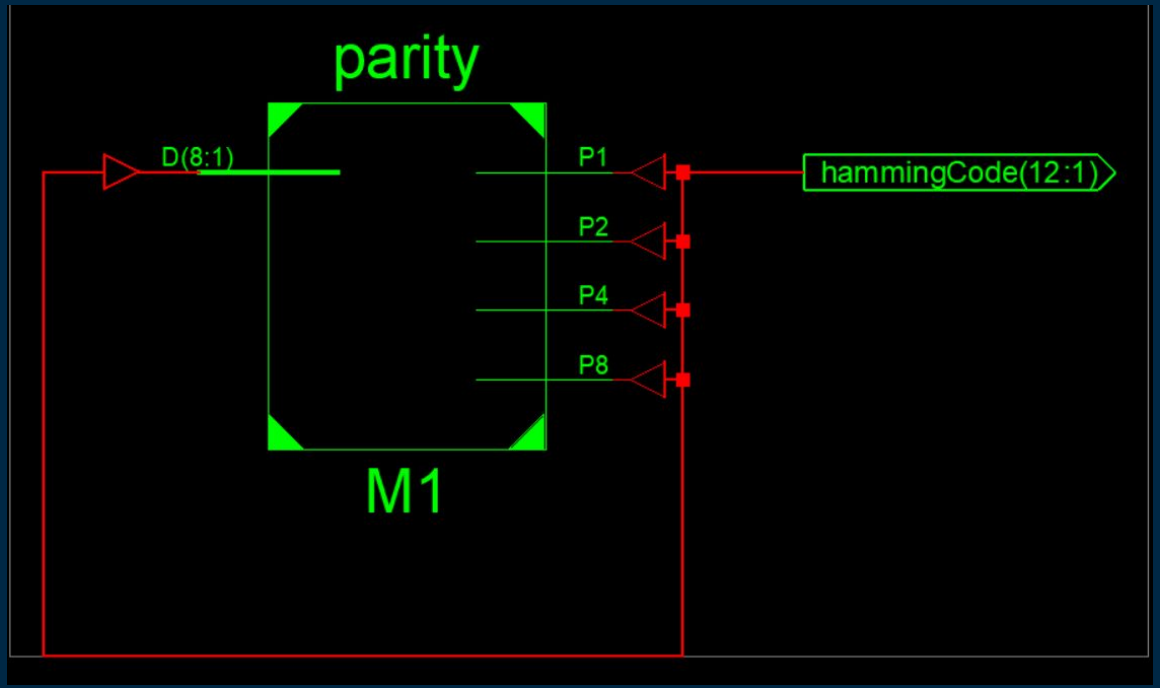
ENCODING SCHEME



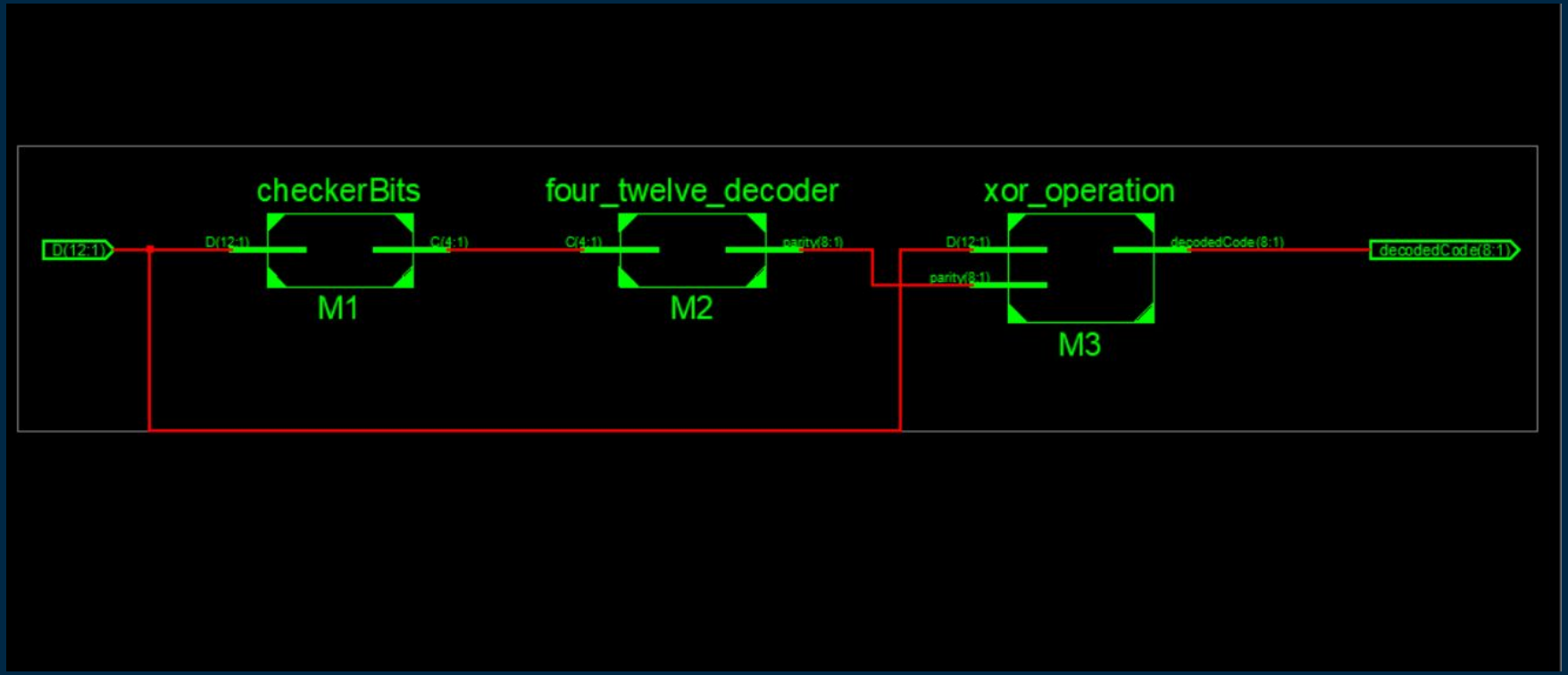
Circuit Diagram



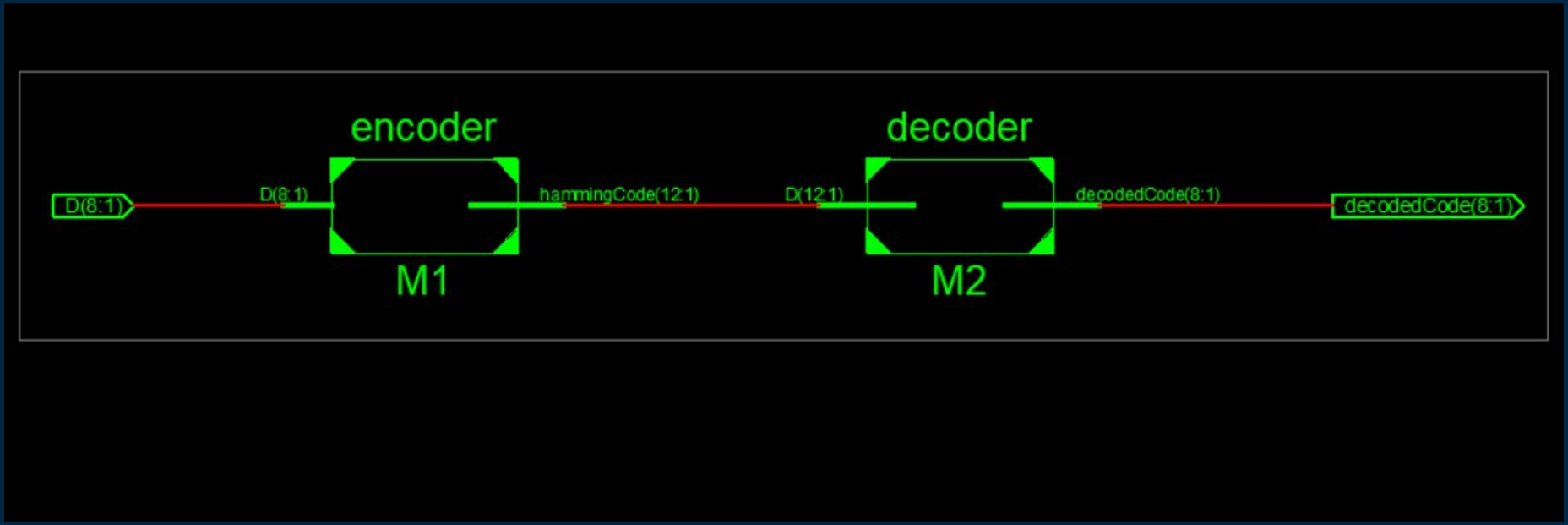
RTL Scheme – Encoder



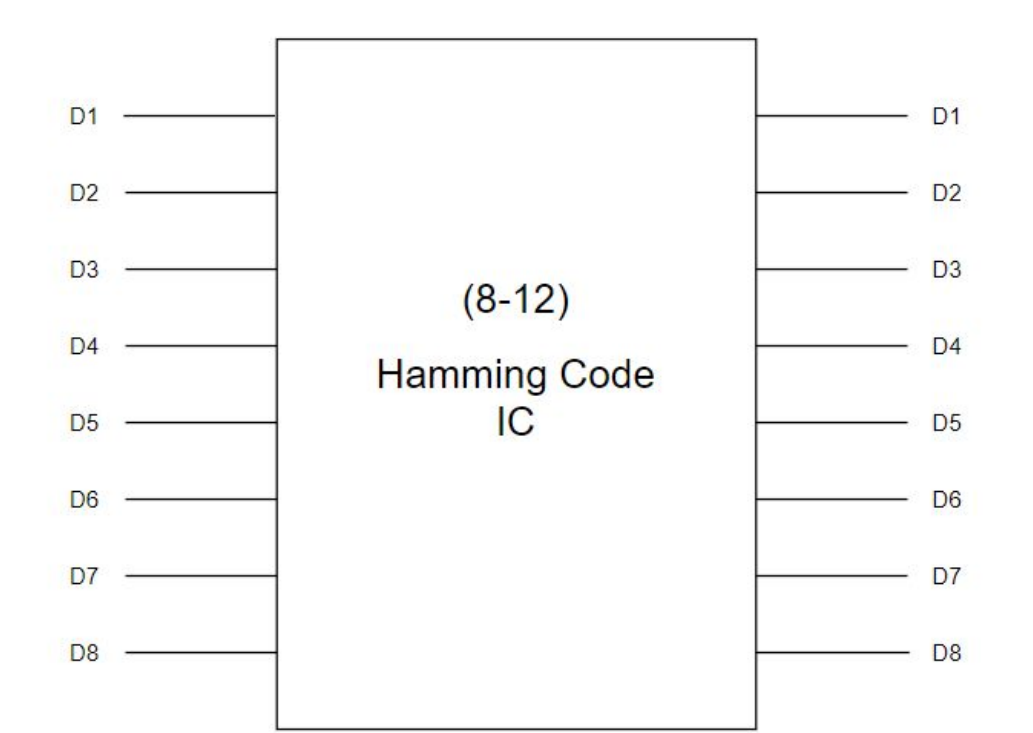
RTL Scheme – Decoder



RTL Scheme – Hamming Code



Pin Structure



Verilog Code – Encoder

```
module encoder(input [8:1]D, output [12:1]hammingCode);
        wire P1,P2,P4,P8;
        parity M1(D, P1,P2,P4,P8);
        assign hammingCode[1]=P1;
        assign hammingCode[2]=P2;
        assign hammingCode[3]=D[1];
        assign hammingCode[4]=P4;
        assign hammingCode[5]=D[2];
        assign hammingCode[6]=D[3];
        assign hammingCode[7]=D[4];
        assign hammingCode[8]=P8;
        assign hammingCode[9]=D[5];
        assign hammingCode[10]=D[6];
        assign hammingCode[11]=D[7];
        assign hammingCode[12]=D[8];
endmodule
```

```
module parity(input [8:1]D, output P1, output
P2, output P4, output P8);

        assign P1=D[1]^D[2]^D[4]^D[5]^D[7];

        assign P2=D[1]^D[3]^D[4]^D[6]^D[7];

        assign P4=D[2]^D[3]^D[4]^D[8];

        assign P8=D[5]^D[6]^D[7]^D[8];

endmodule
```

Verilog Code – Decoder

```
module decoder(input [12:1]D, output [8:1]decodedCode);
        wire [4:1]C;
        wire [8:1]parity;
        checkerBits M1(D, C);
        four_twelve_decoder M2(C, parity);
        xor_operation M3(D,parity,decodedCode);
endmodule
module checkerBits(input [12:1]D, output [4:1]C);
        assign C[1]=D[1]^D[3]^D[5]^D[7]^D[9]^D[11];
        assign C[2]=D[2]^D[3]^D[6]^D[7]^D[10]^D[11];
        assign C[3]=D[4]^D[5]^D[6]^D[7]^D[12];
        assign C[4]=D[8]^D[9]^D[10]^D[11]^D[12];
endmodule
```

```
module four_twelve_decoder(input [4:1]C, output
[8:1]parity);

        assign parity[1]=(~C[4])&(~C[3])&(C[2])&(C[1]);

        assign parity[2]=(~C[4])&(C[3])&(~C[2])&(C[1]);

        assign parity[3]=(~C[4])&(C[3])&(C[2])&(~C[1]);

        assign parity[4]=(~C[4])&(C[3])&(C[2])&(C[1]);

        assign parity[5]=(C[4])&(~C[3])&(~C[2])&(C[1]);

        assign parity[6]=(C[4])&(~C[3])&(C[2])&(~C[1]);

        assign parity[7]=(C[4])&(~C[3])&(C[2])&(C[1]);

        assign parity[8]=(C[4])&(C[3])&(~C[2])&(~C[1]);

endmodule
```

Verilog Code – Decoder

```
module xor_operation(input [12:1]D, input [8:1]parity, output [8:1]decodedCode);

        assign decodedCode[1]=D[3]^parity[1];

        assign decodedCode[2]=D[5]^parity[2];

        assign decodedCode[3]=D[6]^parity[3];

        assign decodedCode[4]=D[7]^parity[4];

        assign decodedCode[5]=D[9]^parity[5];

        assign decodedCode[6]=D[10]^parity[6];

        assign decodedCode[7]=D[11]^parity[7];

        assign decodedCode[8]=D[12]^parity[8];

endmodule
```

Verilog Code – Encoder + Decoder

```
module hamming(input [8:1]D,output [8:1]decodedCode);

        wire [12:1]encode;

        encoder M1(D,encode);

        decoder M2(encode,decodedCode);

endmodule
```

Testbench – Encoder

```
module test_encoder;
        reg [8:1] D;
        wire [12:1] hammingCode;
        encoder uut ( .D(D), .hammingCode(hammingCode));
        initial
        begin
                D=8'b10100001;
              #50 D=8'b10100010;
              #50 D=8'b10101101;
              #50 D=8'b10101110;
              #50 D=8'b10011001;
              #50 D=8'b10011010;
              #50 D=8'b10010101;
              #50 D=8'b10010110;
              #50 D=8'b01100001;
              #50 D=8'b01100010;
              #50 D=8'b01101101;
              #50 D=8'b01101110;
              #50 D=8'b01011001;
              #50 D=8'b01011010;
              #50 D=8'b01010101;
              #50 D=8'b01010110;

        end
        initial #850 $finish;
endmodule
```

Testbench – Decoder

```
module test_decoder;
        reg [12:1] D;
        wire [8:1] decodedCode;
        decoder uut (.D(D), .decodedCode(decodedCode) );

        initial
        begin
                        D=12'b101100001101;
                #50 D=12'b100000010011;
                #50 D=12'b111001101100;
                #50 D=12'b001001110010;
                #50 D=12'b100101000001;
                #50 D=12'b100101001011;
                #50 D=12'b100100000100;
                #50 D=12'b100101111010;
                #50 D=12'b011001000110;
                #50 D=12'b011000111000;
                #50 D=12'b011001110111;
                #50 D=12'b111001111000;
                #50 D=12'b110101001110;
                #50 D=12'b000101010000;
                #50 D=12'b011100101111;
                #50 D=12'010000110001;
        end
        initial #850 $finish;
endmodule
```

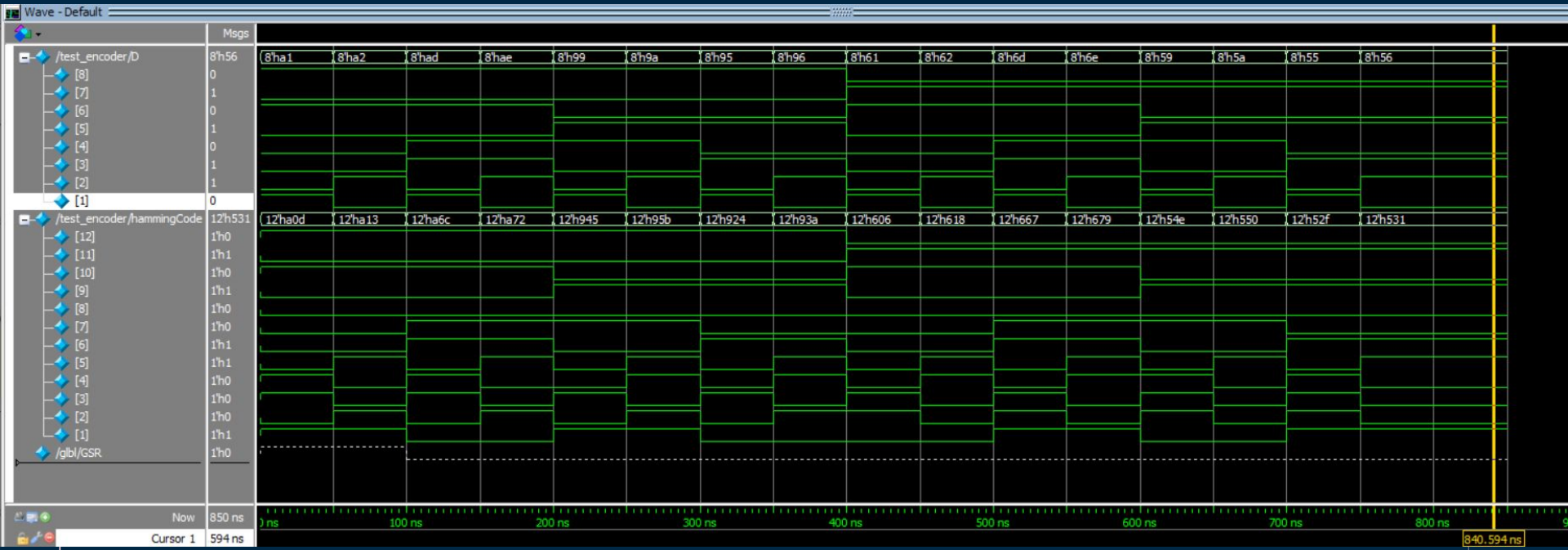
Truth Table – Encoder

DATA BITS	ENCODED BITS
10100001	101000001101
10100010	101000010011
10101101	101001101100
10101110	101001110010
10011001	100101011011
10011010	100100100100
10010101	100100111010
10010110	101000001101
01100001	011000000110
01100010	011000011000
01101101	011001100111
01101110	011001111000
01011001	010101001110
01011010	010101010000
01010101	010100101111
01010110	010100110001

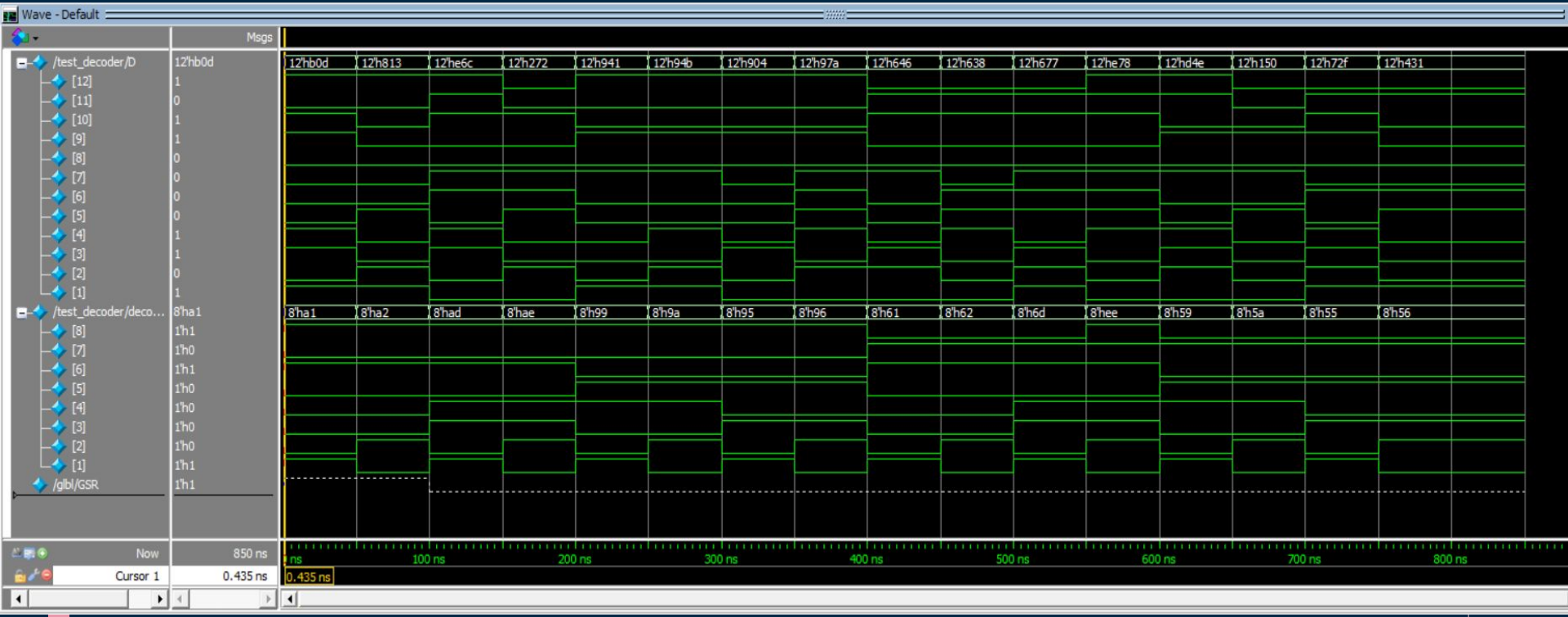
Truth Table – Decoder

ERROR INPUT	CORRECTED INPUT	FINAL OUTPUT
101100001101	101000001101	10100001
100000010011	101000010011	10100010
111001101100	101001101100	10101101
001001110010	101001110010	10101110
100101000001	100101000101	10011001
100101001011	100101011011	10011010
100100000100	100100100100	10010101
100101111010	100100111010	10010110
011001000110	011000000110	01100001
011001100111	011000011000	01100010
011001100111	011001100111	01101101
111001111000	011001111000	01101110
110101001110	010101001110	01011001
000101010000	010101010000	01011010
011100101111	010100101111	01010101
010000110001	010100110001	01010110

Simulation Waveform – Encoder



Simulation Waveform – Decoder



Improvisations in Hamming Code Scheme

1. The basic hamming code can detect and correct single bit error only. By adding another parity bit P9 to the coded word the Hamming code can also be used to detect double bit errors. If we include this additional parity bit to the 12-bit coded word then it becomes a 13-bit coded word. This method is called SEDC-DED algorithm.
   This is done while decoding the codeword. The additional bit "P9" is the XOR operation of all the bits in the codeword.
   P9 = D1^D2^D3^D4^D5^D6^D7^D8^D9^D10^D11^D12
   C = P8P4P2P1
   There are three conditions:
   1. If C=0 and P9=0
   There is no error in the transmitted codeword, so the codeword is taken as valid information.
   2. If C ≠ 0 and P9= 1
   A single bit error occurred that can be detected and corrected.
   3. If C ≠ 0 and P9= 0
   Double bit error occurred that cannot be corrected, so the codeword is taken as invalid information

Improvisations in Hamming Code Scheme

2. The number of redundancy bits, 'r' to be appended to n-bit data is obtained such that the relation $(2^{(r-1)} - 1) \geq n$ is satisfied. The number of redundancy bits in this method is same as that for Hamming code for some values of n. But in some cases, it will be just one more redundancy bit than needed in the Hamming code. For 7-bit data, the number of redundancy bits required will be 4.
The redundancy bits that are to be calculated and appended at bit positions 8, 9, 10, & 11 i.e at the end.
The bit at bit position 8 is selected such that there is even parity at bit positions 1, 3, 5, 7 and bit position 8.
The bit at bit position 9 is selected such that there is even parity at bit positions 2, 3, 6, 7 and bit position 9.
The bit at bit position 10 is selected such that there is even parity at bit positions 4, 5, 6, 7 and bit position 10.
The bit at bit position 11 is selected such that there is even parity considering only the redundancy bits.
The decoding scheme is similar to that of hamming code except the calculation of parity bits is done in a different way.

Improvisations in Hamming Code Scheme

After error detection & correction, if any, the redundancy bits can be removed easily and there is no need to reassemble the data bits. This improved method can be scaled easily for larger data lengths. As an example, let us consider a 56-bit data stream, which can contain eight 7-bit ASCII characters and need 7 redundancy bits. In spite of an extra redundancy bit when compared to the Hamming code, the number of bits involved in the process of calculation of redundancy bits is much less than in the Hamming code. it involves a total of 172 bits in the process of calculation of these 7 redundancy bits. In comparison, in the Hamming code it involves a total of 182 bits in the process of calculation of 6 redundancy bits.

This method improves the number of computations that have to be done and also prevents the re-assembling of data bits once the receiver receives them. Scaling Hamming Code for larger data lengths results in a lot of overhead due to interspersing the redundancy bits and their removal later. This method is suitable for transmission of large size data bit-streams as long as there is likelihood of at the most single-bit error during transmission.

Applications of Hamming Code

Here are some common applications of using Hamming code:

1. Satellites
2. Computer Memory
3. Modems PlasmaCAM
4. Open connectors
5. Shielding wire
6. Embedded Processor