

Hamming error correcting code generator and receiver

EEE 458: VLSI 2 Laboratory

Submitted By:

Name	Student ID
Shadman Shahid	1506179
Shahed-E-Zumrat	1506097
Shamsuddin Ahamed	1506160
Didarul Islam	0806075



Department of Electrical and Electronic Engineering
Bangladesh University of Engineering and Technology

Abstract

In modern data communication systems, error free data transmission is essential for fast and efficient communication. Errors may creep into a transmitted data due to various reasons. However, a robust and efficient system, must be able to handle such adversities properly. That is why error correcting codes are essential for an error-free data communication system. In this regard, forward error correcting codes (FEC) are better for a system, as the receiver system is prepared to handle errors. Moreover, FECs prevent the redundancy of transmitting erroneous data back to the transmitters end for correction. With this end in view, in this project, we have implemented a simple error correcting scheme, based upon the least Hamming distance principle.

Contents

1	Introduction	4
1.1	Error correcting Codes	5
1.2	Generating Codeword	7
1.3	Correcting error in received word	7
2	Behavioral Design: Coding the functional algorithm	9
2.0.1	Module enc	10
2.0.2	Module dec	11
3	Results	13
3.1	Code of Testbench	13
3.2	Functional verification	15
3.3	Schematic generation	16
3.4	Timing, Power and Area considerations	18
3.5	Layout generation	20
3.6	Design Rule Check	22
4	Conclusion	22

List of Figures

1	Major steps in the ASIC design flow process.	4
2	A typical CAD system [1]	5
3	Waveform showing the encoder block input and output. For $d[2:0]=110$, the $Cdata=110110$. For $d[2:0]=001$, the $Cdata=001110$	15
4	Waveform showing the decoder block input r and the error vector output, e . For $r[5:0] = 100011$, $e = 001000$. For $r[5:0] = 111110$, $e = 001000$	15
5	Waveform showing the decoder block input r and output $Cout$. For $r[5:0] = 100011$, $Cdata = 101011$. For $r[5:0] = 000001$, $Cdata = 000001$	16
6	The schematic generated from the behavioral verilog code after RTL synthesis.	16
7	The schematic of the encoder block generated from the behavioral verilog code after RTL synthesis.	17
8	The schematic of the decoder block generated from the behavioral verilog code after RTL synthesis.	18
9	The timing report and the respective slack time generated.	18
10	The timing details including delays and slack time of the different cells and modules. The slack time here is shown to be less than the required time.	19

11	(a) The list of all the generated gates from the behavioral verilog code after RTL synthesis. 35 gates are needed. (b) Area occupied by the gates in the two modules. (c) Amount of switching and leakage power consumed by the two modules.	19
12	(a) The floorplan of the developed circuit developed after adding the power mesh. Blue lines indicate metal 1 and yellow lines indicate metal 4. (b) The layout of the Hamming-encoder-decoder IC after placing and optimizing the standard cells. The routing wire lengths were minimized (c) The layout of the IC after placing the clock tree. (d) Layout after placing metal fillers. (e) Completed layout of the system displayed in Cadence Virtuoso Suite after the necessary DRC checks.	20
13	: The DRC (Design Rule Check) window, showing zero errors and a completed process.	22

List of Tables

1	Some error correcting schemes and the respective (n,k) values.	6
2	The eight possible codewords for corresponding datawords. It can be noted the the 3 most significant bits in each codeword is the same as the dataword.	13
3	The four received words from Fig. 5 tabulated along with the respective error vectors and corrected codewords. It can be seen that the generated codewords are obtained from table 2. the datawords are: 101 , 011 , 000 and 110	13

1 Introduction

A very-large-scale-integration (VLSI) method dictates the process of forming a complex logical circuit systems in an integrated circuit (IC). A standard VLSI design process, also known as 'Application specific IC' (ASIC) design process includes steps for system specification, system architectural design, algorithm development or behavioral design followed by system synthesis including physical design (PnR - Place and Route), shown in Fig. 1. For physical implementation further steps are needed for fabrication and packaging. Among these steps, this report depicts a complete design process up until the layout generation and verification step, excluding fabrication and packaging steps. **Cadence Innovus Solution** is used in this experiment as a design tool.

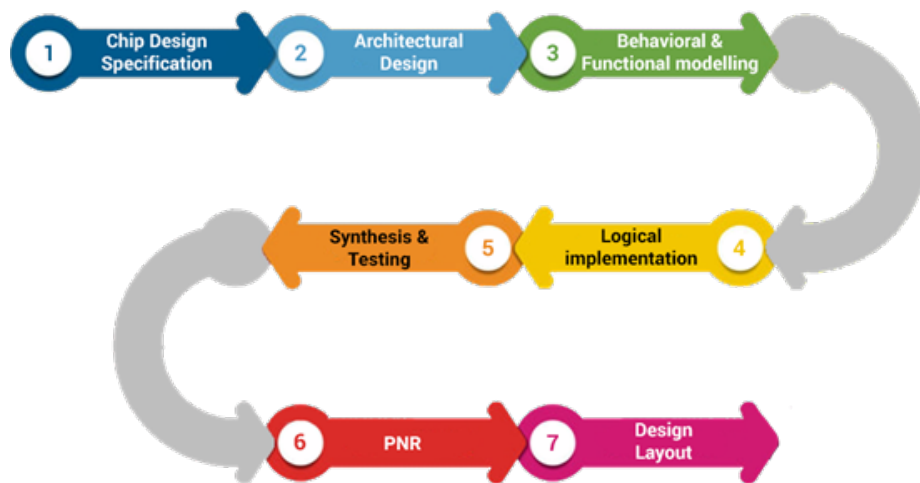


Figure 1: Major steps in the ASIC design flow process.

The front end design consists of a system specification step in which a design is conceptualized using a hardware description language (HDL) such as Verilog in this case. After proper modelling and physical synthesis of the digital subsystem, the system is validated by analyzing the timing diagram from NC-Sim in Cadence Inclusive Unified Simulator (IUS). The flow chart of a typical computer aided design (CAD) system is given in Fig. 2 A proper testbench for testing the devised Verilog code has also been developed. // This report is organized as follows: the latter part of Section 1 explains the algorithm and theory of the function to be performed by our system. The behavioral design steps coded in verilog, via the designated design software package, i.e., Cadence, are discussed in Section 2. Section 3 documents and verifies the results. The timing performance, circuit level and physical design steps are automatically performed by the design software are verified in this section. The section ends with a summary of the key performance parameters and device specifications. Section 4 presents an overview of the entire report.

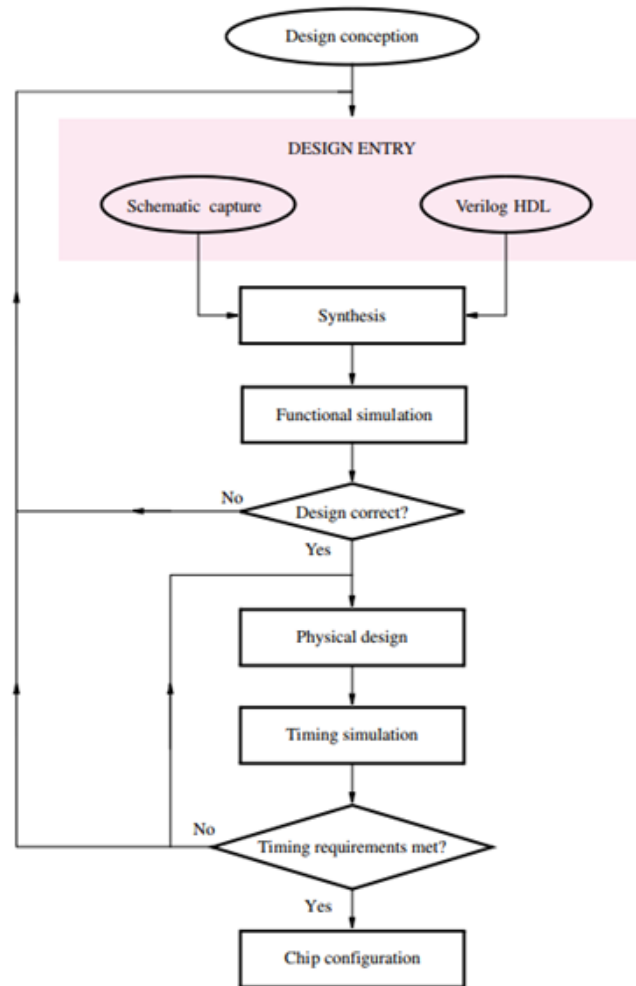


Figure 2: A typical CAD system [1]

1.1 Error correcting Codes

A **Forward error correcting code** (FEC) is a coding scheme, where error in a transmitted bit string can be corrected at the receiver end, without referring it back to the transmitter. Thus, there isn't any back propagation of information. Hence, this scheme is a '**Forward**' coding scheme. The key ideas behind this particular error correcting coding scheme is based on the **Hamming distance** concept.

When transmitting data, the data is not transmitted in raw form. It is encoded and converted to a longer string with more bits appended to it. A data bit string (known as **dataword**) is converted to a unique codeword before being transmitted. A codeword (c) is the unit of bits that can be decoded independently. Codeword of length n , transmits k bit dataword. So, **Codeword = Dataword + Check Digits**. The number of **check digits**, m is thus $n-k$. This is the kind of FEC known as **block code**.

Hamming distance between the two unmatched sequences is defined as the number of unmatched digits between two bit sequences of equal length. In the field of error correcting code, the number of mismatches between digits is in general known as **distance**. For example **100110** and **111110** has a hamming distance of 2 as the 4th and 5th bits of the two sequences do not match. **Hamming space** is the set of all possible 2^N binary strings of length N . The hamming space for $N = 6$ consists of 2^6 numbers ranging from **000000** to **111111**. Thus, **100110** and **110011** are both members of the hamming space for $N = 6$.

For correcting a minimum of t errors the minimum distance between two unique code-words, δ_{\min} , is given by $\delta_{\min} = 2t + 1$. The relation between the number of check digits m and the length of the code word, n for $t = 1$, is given by

$$2^m \geq \sum_{j=0}^t \binom{n}{j}$$

$$2^m \geq \binom{n}{0} + \binom{n}{1} \quad (1)$$

This limit on m here, is known as the **Hamming bound**. Only for a few code schemes, this inequality becomes an equality. Here, for $k=3$, after simplifying the initial expression we obtain the following relation. Only for single error correcting codes (i.e., $t = 1$), this condition is a **necessary** and **sufficient** condition.

$$2^{n-3} \geq n + 1$$

$$2^n \geq 8(n + 1) \quad (2)$$

Equation. 2 is satisfied for $n = 6$. For a 3 bit dataword, to detect a maximum of single error ($t = 1$), the codeword must be of 6 bits. This particular scheme is a (6,3) coding scheme. The following table shows different coding schemes with the codeword length, dataword length and code rate.

	n	k	Code	Code Rate (n/k)
Single-error correcting, $t=1$	4	1	(4,1)	0.25
Minimum code separation 3	5	2	(5,2)	0.4
	6	3	(6,3)	0.5
	7	4	(7,4)	0.57
	15	11	(15,11)	0.73

Table 1: Some error correcting schemes and the respective (n,k) values.

1.2 Generating Codeword

A 6 bit codeword is generated from a 3 bit dataword. The codeword is formed by a linear combination of the data bits. An n bit long codeword \mathbf{c} , and k bit long dataword \mathbf{d} is defined as follows.

$$\mathbf{c} = (c_1, c_2, \dots, c_n)$$

$$\mathbf{d} = (d_1, d_2, \dots, d_k)$$

For a special case in which codeword bits from c_1 up to c_k is same as corresponding databits from d_1 up to d_k . For code word bits from c_{k+1} to c_n are defined by a linear combination of all the data bits, d_1, d_2, \dots, d_k . Such a scheme is known as **systematic code**. For a dataword **110**, a the generated codeword may be, **110111** where the check digits **111** are a linear combination of the databits. To transform the dataword to a codeword a **generator matrix**, \mathbf{G} , is defined. This generator matrix is later used at the receiver end to decode the received word and detect errors.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & h_{11} & h_{21} & \dots & h_{m1} \\ 0 & 1 & 0 & \dots & 0 & h_{12} & h_{22} & \dots & h_{m2} \\ & & & \vdots & & & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & h_{1k} & h_{2k} & \dots & h_{mk} \end{bmatrix} \quad (4)$$

$\underbrace{\hspace{10em}}_{\mathbf{I}_k(k \times k)} \quad \underbrace{\hspace{10em}}_{\mathbf{P}(k \times m)}$

The first portion of the generator matrix is an **identity matrix** of size k . The latter portion of the generator Matrix ($\mathbf{P}(k \times m)$) generates those bits which form the check digits of the code word. Thus, by knowing the dataword, we can calculate the check digits if we know \mathbf{G} .

$$\begin{aligned} \mathbf{c} &= \mathbf{dG} \\ &= \mathbf{d} [\mathbf{I}_k \quad \mathbf{P}] \\ &= [\mathbf{d} \quad \mathbf{dP}] \end{aligned} \quad (5)$$

Thus, for a given \mathbf{P} matrix, we can generate the codeword \mathbf{c} from a dataword.

1.3 Correcting error in received word

The received word of n bits is transformed, to give the error word of n bits. The error word has **1s** in those bits where there is an error and **0s** for the correctly transmitted bits. For example, if **001110** is a valid codeword, and **001100** is a received word, then the error word \mathbf{e} is **000010** as the 2nd least significant bit is erroneously transmitted. And, **001110** is the

corrected codeword. The received word \mathbf{r} is transformed by the \mathbf{H} matrix,

$$\mathbf{H}^T = \begin{bmatrix} \mathbf{P} \\ \mathbf{I}_m \end{bmatrix} \quad \text{where,} \quad \mathbf{P} = \begin{bmatrix} h_{11} & h_{21} & \cdots & h_{m1} \\ h_{12} & h_{22} & \cdots & h_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ h_{1k} & h_{2k} & \cdots & h_{mk} \end{bmatrix} \quad \text{and} \quad \mathbf{I}_m = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

On transforming \mathbf{r} , a non-zero row vector \mathbf{s} of length \mathbf{k} , is generated. This vector is referred to in literature as **syndrome**.

$$\mathbf{s} = \mathbf{r}\mathbf{H}^T \quad (6)$$

The \mathbf{s} bit string is compared with the rows of the matrix \mathbf{H}^T . The row to which \mathbf{s} matches completely, specifies the location of the error bit in \mathbf{r} . Thus, a bitwise XOR operation (i.e modulo -2 addition) between the error vector \mathbf{e} and received word \mathbf{r} will generate the corrected codeword. This codeword is obtained from the least hamming distance between the valid codewords from the received word.

2 Behavioral Design: Coding the functional algorithm

To implement an error correcting scheme, the designed IC could perform two major functions.

1. **Encoding:** Generate desired codeword for a dataword, performed by module `enc`
2. **Decoding:** Spotting the error and finding the codeword from a received word, performed by module `dec`

Thus, in developing the algorithm for this (6,3) systematic linear coding scheme, we had to develop two modules, `enc` and `dec` for these two objectives. Thereafter, these two modules were utilised in the top module `Hamm_enc_dec`. In the top module, the 3-bit dataword `d`, 6-bit received word `r` and the clock was taken as inputs while `Cdata`, `Cout` and `e` bit strings are the outputs. `Cdata` is the codeword generated by encoding the dataword `d`. `Cout` and `e` are the corrected codeword and the error vector for the received word `r`. The module below shows the top module code. Also, in this module the parameters `I0`, `I1`, `I2`, `P0` and `P1` and `P2` are initialized as the columns of the generator matrix G . The H^T matrix is also initialized from these vectors.

```
module Hamm_enc_dec (e , Cout , Cdata , r , d , clk );
```

```
    input  clk ;
    input  [2:0] d ;
    input  [5:0] r ;
    output [5:0] e , Cout , Cdata ;
    parameter [2:0] I0 = 4 , I1 = 2 , I2 = 1 ;
    parameter [2:0] P0 = 3'b101 , P1 = 3'b011 , P2 = 3'b110 ;
    wire [5:0] H3 , H2 , H1 ;

    assign H3 = { P0 , I0 } ;
    assign H2 = { P1 , I1 } ;
    assign H1 = { P2 , I2 } ;

    enc E1 ( Cdata , d , I0 , I1 , I2 , P0 , P1 , P2 , clk ) ;
    dec D1 ( S , e , Cout , r , H3 , H2 , H1 , clk ) ;
```

```
endmodule
```

2.0.1 Module enc

This module took as input the 3 bit dataword, the **generator matrix G** and a clock input. The generator matrix G for a (6,3) scheme is a (3×6) matrix. So, each column of this matrix was given as input in this module, (I0, I1, I2, P0, P1, P2). For a particular dataword, \mathbf{d} , the codeword \mathbf{c} is generated as \mathbf{dG} . For this project we used the following matrix as G .

$$\mathbf{G} = \begin{matrix} & \begin{matrix} 1 & 0 & 0 & 1 & 0 & 1 \end{matrix} \\ \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{matrix} \\ \begin{matrix} (I0) \\ (I1) \\ (I2) \end{matrix} & \begin{matrix} (P0) \\ (P1) \\ (P2) \end{matrix} \end{matrix} \quad (7)$$

For a particular $\mathbf{d=110}$, we get the codeword

$$\begin{aligned} \mathbf{c} &= \mathbf{dG} \\ &= [1 \ 1 \ 0] \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{aligned}$$

$$\therefore \mathbf{c} = [1 \ 1 \ 0 \ 1 \ 1 \ 0]$$

The above operation is implemented in the code snippet below.

```

module enc (Cen , d , I0 , I1 , I2 , P0 , P1 , P2 , clk );
input clk ;
input [2:0]d , I0 , I1 , I2 , P0 , P1 , P2 ;
output [5:0]Cen ;
reg [5:0]c ;

always@ ( posedge clk )
begin
    c = {^(d&I0) ,^( d&I1 ) ,^( d&I2 ) ,^( d&P0 ) ,^( d&P1 ) ,^( d&P2 )};
end
assign Cen = c ;
endmodule

```

2.0.2 Module dec

The received word \mathbf{r} is taken as input and is transformed into an error vector \mathbf{e} by the \mathbf{H}^T vector. The \mathbf{H}^T is obtained from the generator matrix as

$$\begin{aligned}\mathbf{H}^T &= \begin{bmatrix} P0 & P1 & P2 \\ I0 & I1 & I2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

Each of the columns of \mathbf{H}^T is taken as input. Thereafter the syndrome vector is obtained. The calculation for a sample received word of $\mathbf{r}=\mathbf{100011}$ is given below.

$$\begin{aligned}\mathbf{s} &= \mathbf{r}\mathbf{H}^T \\ &= [1 \ 0 \ 0 \ 0 \ 1 \ 1] \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \therefore \mathbf{s} &= [1 \ 1 \ 0]\end{aligned}$$

For a received word of $\mathbf{100011}$ we obtain a syndrome of $\mathbf{s}=\mathbf{110}$. Now this syndrome matches the 3rd row of \mathbf{H}^T .

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

This corresponds to an error vector, $\mathbf{e}=\mathbf{001000}$. Then the codeword closest (in terms of Hamming distance) to this received word, is obtained as $\mathbf{c}=\mathbf{e}\oplus\mathbf{r}=\mathbf{101011}$. Here, the XOR operation is the same as modulo-2 addition. The code snippet of the decoder module is given in the next page. This module finds the codeword which has the least hamming distance from the received word.

```

module dec(error , c, r, H3,H2,H1, clk);
input clk;
input [5:0]r, H3, H2, H1;
output reg [2:0]S;
output [5:0]error , c;
reg [5:0]e = 6'b000000;

always@(posedge clk)
begin

    S = {^(r & H3),^(r & H2),^(r & H1)};

    if(S == {H3[0],H2[0],H1[0]}) e = 6'b000001;
    else if(S == {H3[1],H2[1],H1[1]}) e = 6'b000010;
    else if(S == {H3[2],H2[2],H1[2]}) e = 6'b000100;
    else if(S == {H3[3],H2[3],H1[3]}) e = 6'b001000;
    else if(S == {H3[4],H2[4],H1[4]}) e = 6'b010000;
    else if(S == {H3[5],H2[5],H1[5]}) e = 6'b100000;
    else if (S == 3'b111) e = 6'b100010;

end

assign error = e;
assign c = e^r;
endmodule

```

3 Results

The encoder block transforms the dataword into codeword as per Eqn. 5. The transformation is depicted in the table Table 2. The transformation of some sample received words as shown in Fig. 4 and Fig. 5 is given in Table 3.

d	c
000	000000
001	001110
010	010011
011	011101
100	100101
101	101011
110	110110
111	111000

Table 2: The eight possible codewords for corresponding datawords. It can be noted the the 3 most significant bits in each codeword is the same as the dataword.

r	e	c
100011	001000	101011
011100	000001	011101
000001	000001	000001
111110	001000	110110

Table 3: The four received words from Fig. 5 tabulated along with the respective error vectors and corrected codewords. It can be seen that the generated codewords are obtained from table 2. the datawords are: **101**, **011**, **000** and **110**.

3.1 Code of Testbench

The code of the testbench is given below. the system works after the positive edge of the clock pulse. The timing diagram is attached in the figures that follow. By, comparing the timing diagram with the table mentioned in the preceding section.

```

module Hamm_stim;
  reg [5:0] r;
  wire [2:0] d;
  Hamm_enc_dec H1(e, Cout, Cdata, r, d);

  initial
    begin
      clk = 1'b0;
      forever
        begin
          #5 clk = ~clk;
        end
      end

  initial
    begin
      $shm_open("shm.db", 1);
      $shm_probe("AS");
      #100 $finish;
      #100 $shm_close();

    end
  //simulate the Input Signals
  initial
    begin
      #0 r <= 6'b000000;
        d <= 0;
      #5 r <= 6'b100011;
        d <= 3'b110;
      #15 r = 6'b011100;
        d = 3'b001;
        forever
          begin
            #20 r = ~r;
              d = ~d;
          end
        end
    end
endmodule

```

3.2 Functional verification

The verilog code of the system, is simulated and visualized using **Sim Vision**. The timing diagram analysis shows that the code is working correctly and that it performs the required functions correctly. The two blocks are verified by visualizing the timing diagrams in Fig. 13 and 4. From the lookup table, we can find the corresponding codeword for a given

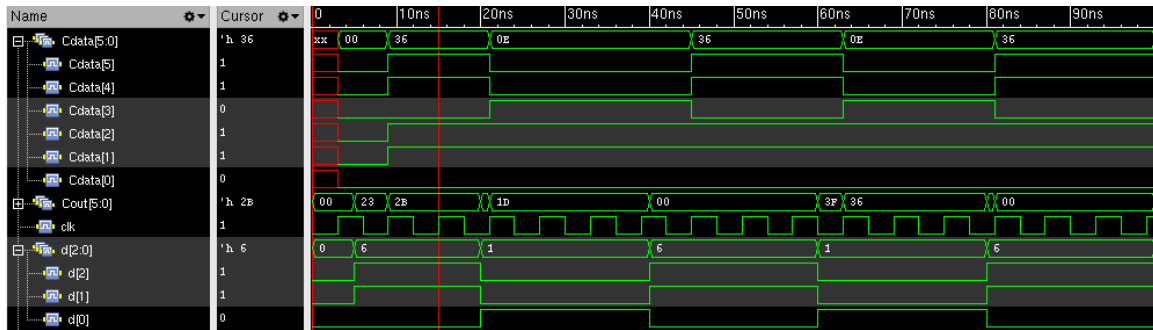


Figure 3: Waveform showing the encoder block input and output. For $d[2:0]=110$, the $Cdata=110110$. For $d[2:0]=001$, the $Cdata=001110$.

dataword from Table 2. This is generated from the Eqn. 5. The encoder block has been verified by checking whether the timing diagrams matches the data of this table.

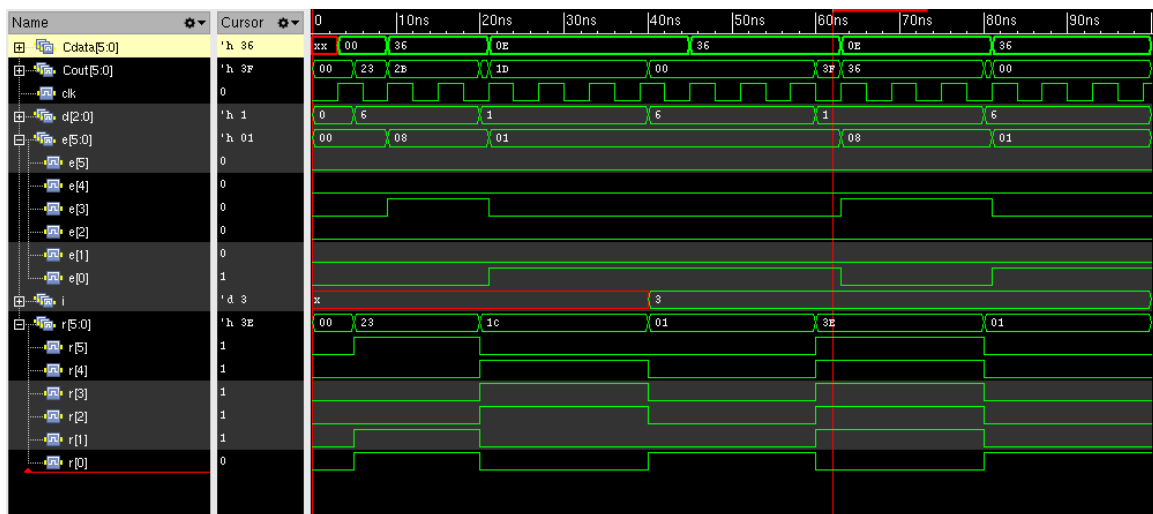


Figure 4: Waveform showing the decoder block input r and the error vector output, e . For $r[5:0] = 100011$, $e = 001000$. For $r[5:0] = 111110$, $e = 001000$.

For the decoder block as well, table 2 to verify the results. Each of the received word is transformed into a possible valid word, based on the least hamming distance between the valid codeword and received word. Also, the (6,3) can correct at most **1** error. The timing diagram for r and corrected codeword c generated is shown in Fig. 5.

level diagram, the parallel inputs specifying \mathbf{G} and \mathbf{H}^T serve no function and hence are not used in the logic circuit for serial processing. This has been done to keep the number of inputs limited to the words to be transmitted and received. However, these parallel inputs can easily be incorporated to the system by making the generator matrix – a system parameter, a separate parallel input to the system. This will make the system more versatile and customizable. The internal configuration of each of the two encoding and decoding modules are given in figures 7 and 8. Both the blocks have the same clock.

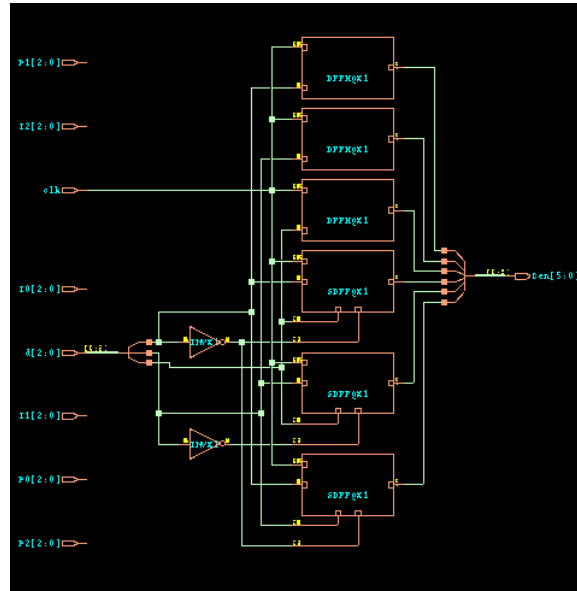


Figure 7: The schematic of the encoder block generated from the behavioral verilog code after RTL synthesis.

The encoder block as shown in Fig. 7, consists of 6 flip-flops, 3 D-type and 3 of SR-type. Since the three MSBs of the output codeword is the same as the dataword, the three bits of the dataword are taken as input to the three D-flip-flops. On the other hand the three LSBs of the codeword are a pseudo-random combination the dataword bits, based upon the latter part of the generator matrix. So, for the given generator matrix in Eqn. 7, the dataword is fed to the three SR-flip-flops. Inverters are used to feed to the R-input. The given configuration in the encoder block, without any more gates and logic elements is enough for implementing the generator matrix, \mathbf{G} given in Eqn. 7. The decoder block is composed of a rather complicated network of sequential circuit. The six bit dataword is processed by a combinational circuit before being fed to the block of flip-flops. The sequential circuit implements, the logic function to generate the error word and corrected word. The flip-flops ensure synchronous operation.

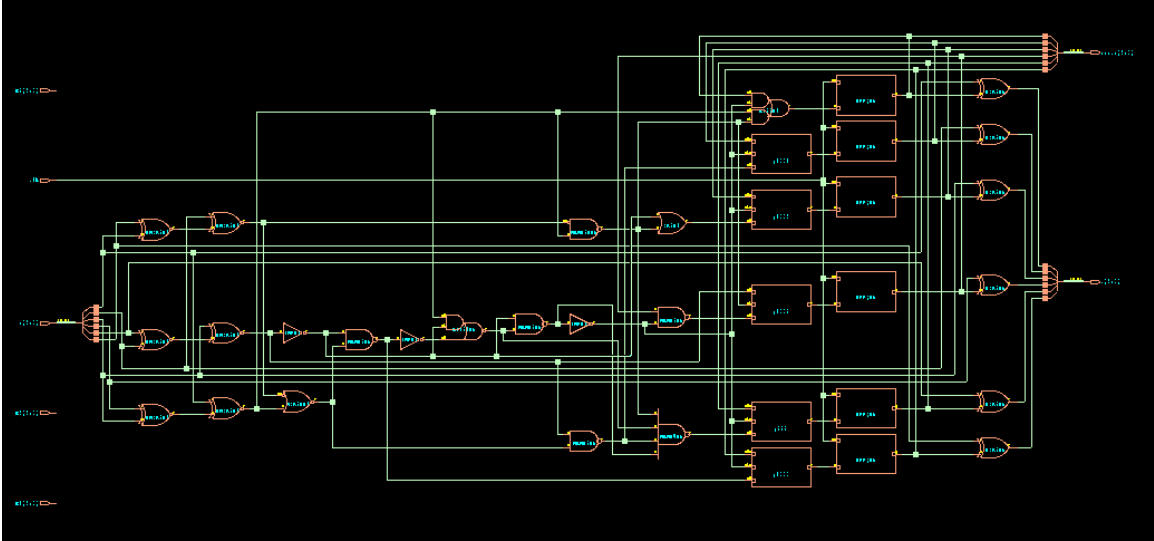


Figure 8: The schematic of the decoder block generated from the behavioral verilog code after RTL synthesis.

	Pin	Type	Fanout	Load (F)	Slew (ps)	Delay (ps)	Arrival (ps)	
f[0]		in port	2	0.40	0.00	0.00	0.00	R
D1/r[0]						0.00	0.00	
g1021/A		XNOR2X1	1	0.40	28.20	162.80	162.80	F
g1018/B						0.00	162.80	
g1018/Y		XNOR2X1	2	0.40	26.50	161.80	324.60	R
g1014/AN						0.00	324.60	
g1014/Y		NOR2BX1	2	0.40	46.50	90.90	415.50	R
g1013/B						0.00	415.50	
g1013/Y		NAND2XL	2	0.70	107.70	96.90	512.40	F
g1011/A						0.00	512.40	
g1011/Y		INVX1	1	0.20	29.60	71.50	583.90	R
g1009/B0						0.00	583.90	
g1009/Y		AOI21XL	2	0.40	95.20	42.20	626.10	F
g1008/B						0.00	626.10	
g1008/Y		NAND2XL	2	0.60	49.20	81.10	707.20	R
g1005/D						0.00	707.20	
g1005/Y		NAND4XL	1	0.40	218.30	167.80	874.80	F
g997/B0						0.00	874.80	
g997/Y		OAI2BB1X1	1	0.20	56.00	141.60	1016.40	R
e_msg[1]D		DFFOXL				0.00	1016.40	
e_msg[1]CK		setup			0.00	135.10	1151.50	R

Figure 9: The timing report and the respective slack time generated.

3.4 Timing, Power and Area considerations

The timings recorded by the software indicate a total time delay of 1151.50 ps for the input data to propagate to the output end. This is the slack time generated from the timing information table in Fig. 9. It can be seen that the XNOR gates have the highest contributions to the total arrival delay time. From Fig. 13 we can observe that the slack time is less than the required time. A total of 35 gates are used in the system, which occupy an area 89.94 μm . A list of all the gates and their respective areas and associated technology library are given in Fig. 11. The list mentions all the gates used in both the encoder and decoder modules. Among all the gates the encoder module uses only three gates, occupying an area of 8.21 μm . The decoder module on the other hand occupies the rest of the total area mentioned in Fig. 11.

```

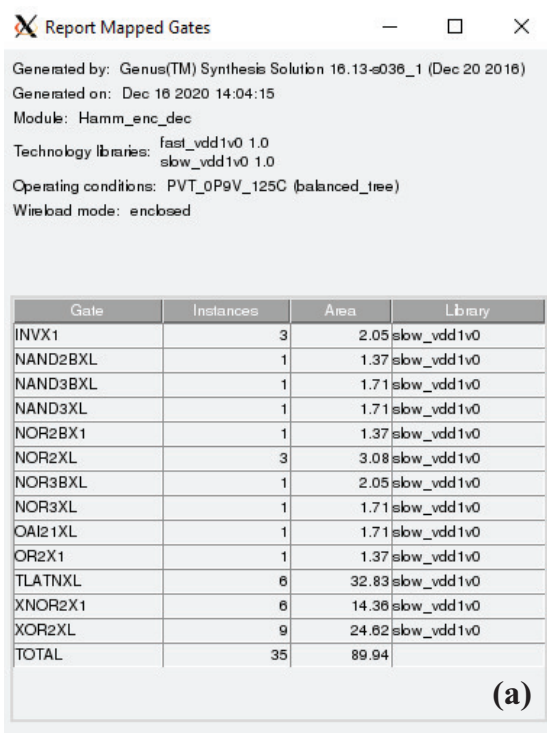
Path 1: MET Setup Check with Pin D1/e_reg[5]/CK
Endpoint: D1/e_reg[5]/D (^) checked with leading edge of 'func_clk'
Beginpoint: r[2] (^) triggered by leading edge of '@'
Path Groups: {func_clk}
Analysis View: func@BC_rcbest0.hold
Other End Arrival Time      0.000
- Setup                    0.019
+ Phase Shift              2.500
= Required Time            2.481
- Arrival Time             0.257
= Slack Time               2.224

Clock Rise Edge            0.000
+ Input Delay              0.000
= Beginpoint Arrival Time  0.000

```

Instance	Arc	Cell	Delay	Arrival Time	Required Time
	r[2] ^			0.000	2.224
D1/g1022	A ^ -> Y v	XNOR2X1	0.044	0.044	2.269
D1/g1019	B v -> Y ^	XNOR2X1	0.045	0.089	2.313
D1/g1014	B ^ -> Y v	NOR2BX1	0.016	0.105	2.330
D1/g1013	B v -> Y ^	NAND2XL	0.020	0.125	2.349
D1/g1011	A ^ -> Y v	INVX1	0.014	0.138	2.363
D1/g1009	B0 v -> Y ^	AOI21XL	0.024	0.162	2.387
D1/g1008	B ^ -> Y v	NAND2XL	0.028	0.191	2.415
D1/g1007	A v -> Y ^	INVX1	0.025	0.215	2.440
D1/g1004	A1 ^ -> Y ^	AO22X1	0.042	0.257	2.481
D1/e_reg[5]	D ^	DFFQXL	0.000	0.257	2.481

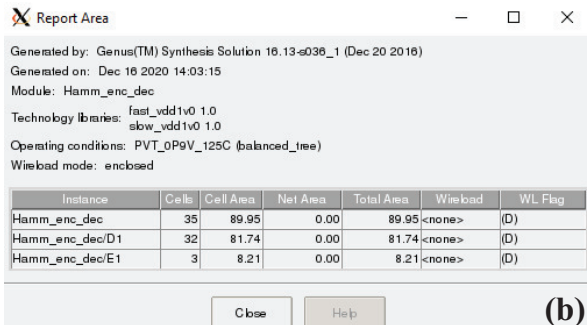
Figure 10: The timing details including delays and slack time of the different cells and modules. The slack time here is shown to be less than the required time.



Generated by: Genus(TM) Synthesis Solution 16.13-s036_1 (Dec 20 2016)
 Generated on: Dec 16 2020 14:04:15
 Module: Hamm_enc_dec
 Technology libraries: fast_vdd1v0 1.0
 slow_vdd1v0 1.0
 Operating conditions: PVT_0P9V_125C (balanced_tree)
 Wirebad mode: enclosed

Gate	Instances	Area	Library
INVX1	3	2.05	sbw_vdd1v0
NAND2BXL	1	1.37	sbw_vdd1v0
NAND3BXL	1	1.71	sbw_vdd1v0
NAND3XL	1	1.71	sbw_vdd1v0
NOR2BX1	1	1.37	sbw_vdd1v0
NOR2XL	3	3.08	sbw_vdd1v0
NOR3BXL	1	2.05	sbw_vdd1v0
NOR3XL	1	1.71	sbw_vdd1v0
OAI21XL	1	1.71	sbw_vdd1v0
OR2X1	1	1.37	sbw_vdd1v0
TLATNXL	6	32.83	sbw_vdd1v0
XNOR2X1	6	14.36	sbw_vdd1v0
XOR2XL	9	24.62	sbw_vdd1v0
TOTAL	35	89.94	

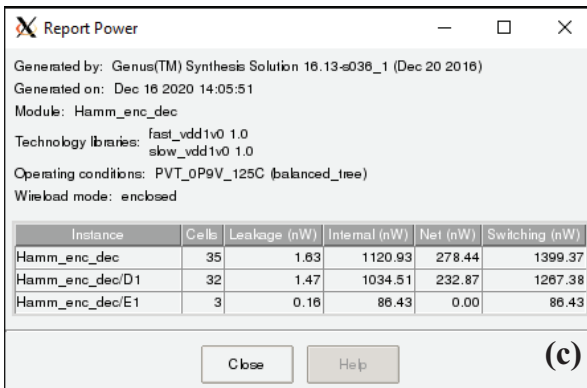
(a)



Generated by: Genus(TM) Synthesis Solution 16.13-s036_1 (Dec 20 2016)
 Generated on: Dec 16 2020 14:03:15
 Module: Hamm_enc_dec
 Technology libraries: fast_vdd1v0 1.0
 slow_vdd1v0 1.0
 Operating conditions: PVT_0P9V_125C (balanced_tree)
 Wirebad mode: enclosed

Instance	Cells	Cell Area	Net Area	Total Area	Wirebad	WL Flag
Hamm_enc_dec	35	89.95	0.00	89.95	<none>	(D)
Hamm_enc_dec/D1	32	81.74	0.00	81.74	<none>	(D)
Hamm_enc_dec/E1	3	8.21	0.00	8.21	<none>	(D)

(b)



Generated by: Genus(TM) Synthesis Solution 16.13-s036_1 (Dec 20 2016)
 Generated on: Dec 16 2020 14:05:51
 Module: Hamm_enc_dec
 Technology libraries: fast_vdd1v0 1.0
 slow_vdd1v0 1.0
 Operating conditions: PVT_0P9V_125C (balanced_tree)
 Wirebad mode: enclosed

Instance	Cells	Leakage (nW)	Internal (nW)	Net (nW)	Switching (nW)
Hamm_enc_dec	35	1.63	1120.93	278.44	1399.37
Hamm_enc_dec/D1	32	1.47	1034.51	232.87	1267.38
Hamm_enc_dec/E1	3	0.16	86.43	0.00	86.43

(c)

Figure 11: (a) The list of all the generated gates from the behavioral verilog code after RTL synthesis. 35 gates are needed. (b) Area occupied by the gates in the two modules. (c) Amount of switching and leakage power consumed by the two modules.

3.5 Layout generation

Physical design process begins with a ‘floorplan’. The floorplan estimates the area of major units in the chip and defines their relative placements. The floorplan is essential to determine whether a proposed design will fit in the chip area budgeted and to estimate wiring lengths and wiring congestion. The process is highly feedback driven as floorplans will often dictate change to the logic (and microarchitecture), which in turn inflict changes to the floorplan. For complex designs, the floorplan is often hierarchically subdivided to describe the functional blocks within the units. The challenge of floorplanning lies in estimating the size of each unit without proceeding through a detailed design of the chip (which would depend on the floorplan and wire lengths).

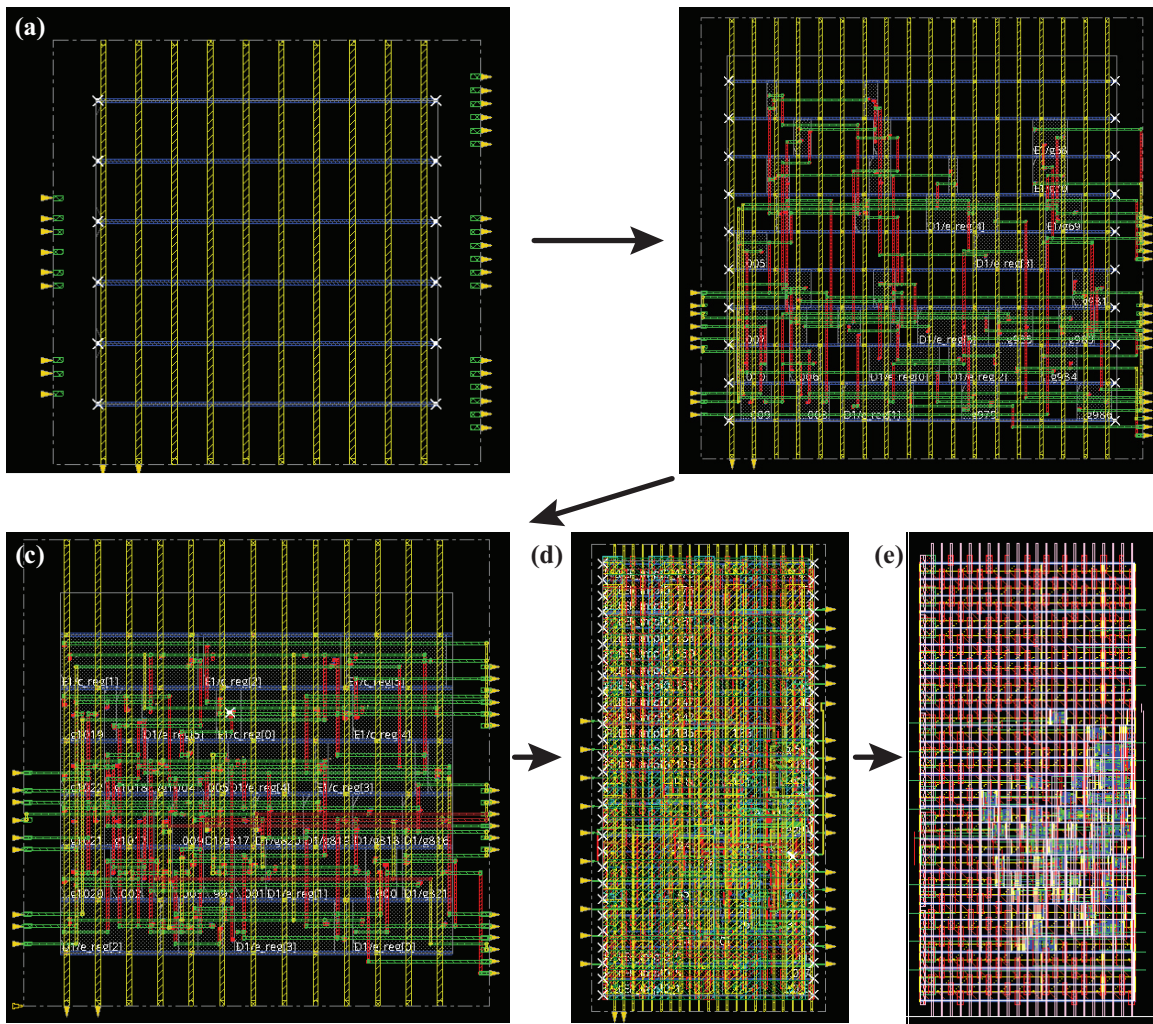


Figure 12: (a) The floorplan of the developed circuit developed after adding the power mesh. Blue lines indicate metal 1 and yellow lines indicate metal 4. (b) The layout of the Hamming-encoder-decoder IC after placing and optimizing the standard cells. The routing wire lengths were minimized (c) The layout of the IC after placing the clock tree. (d) Layout after placing metal fillers. (e) Completed layout of the system displayed in Cadence Virtuoso Suite after the necessary DRC checks.

To implement the layout of the synthesized logic circuit, from the gate level Verilog code, the floorplan and pin placement were implemented as shown in Fig. 11. Power and ground run horizontally in metal 1 (blue stripes). It is to be mentioned Metals are numbered according to their hierarchy (lower number indicates deeper layer). These supply rails are 8λ wide (to carry more current) and are separated by 90λ center-to-center. The nMOS transistors are placed in the bottom 40λ of the cell and the pMOS transistors are placed in the top 50λ . Thus, cells can be connected by abutment with the supply rails and n-well matching up. Substrate and well contacts are placed under the supply rails. Inputs and outputs are provided in metal 2, which runs vertically. Each cell is a multiple of 8λ in width so that it offers an integer number of metal 2 tracks. Within the cell, poly is run vertically to form gates and diffusion and metal1 are run horizontally, though metal1 can also be run vertically to save space when it does not interfere with other connections. Cells are tiled in rows. Each row is separated vertically by at least 10λ from the base of the previous row. In a 2-level metal process, horizontal metal1 wires are placed in routing channels between the rows. The number of wires that must be routed sets the height of the routing channels. As we have observed in this experiment, layout is often generated with automatic place and route tools.

The top layer consisted of vertical yellow stripes of metal 4 rails and a bottom layer having horizontal stripes of metal 1 rails. The V_{DD} and V_{SS} rails are placed alternately. Each stripe is of $0.16 \mu\text{m}$ width and spaced $0.84 \mu\text{m}$ apart. When more layers of metal are available, routing takes place over the cells and routing channels may become unnecessary. For example, in a 3-level metal process, metal3 is run horizontally on a 10λ pitch. Thus, 11 horizontal tracks can run over each cell. If this is sufficient to accommodate all of the horizontal wires, the routing channels can be eliminated. Automatic synthesis and place and route tools are good enough to map entire designs onto standard cells. Synthesized designs tend to be somewhat slower than a good custom design, but they also take significantly less design effort.

To supply the clock signal to the sequential components placed at different locations throughout the IC, the clock routing need to be done efficiently. So, in routing the clock signal to those components, the critical path has to be minimized efficiently, so that delay does not occur. This is why, clock signal is routed to different locations within the IC, with the least possible propagation delay. This scheme of clock distribution network within an IC is known as a clock tree. includes the clocking circuitry and devices from clock source to destination. The complexity of the clock tree and the number of clocking components used depends on the hardware design. Since systems can have several ICs with different clock performance requirements and frequencies, a “clock tree” refers to the various clocks feeding those ICs. In this project the clock tree was synthesized and optimized automatically.

Thereafter, filler cells were added and metal fillers were placed to occupy the empty locations within the wafer. Finally, after exporting the design to Cadence Layout XL, the layout design was validated to correct any design violations.

3.6 Design Rule Check

Design rule checkers (DRC) verify that the layout satisfies design rules. There were two violations in the geometry verification, which were resolved after adding filler cell and exporting to virtuoso environment. No DRC errors were obtained in the final layout.

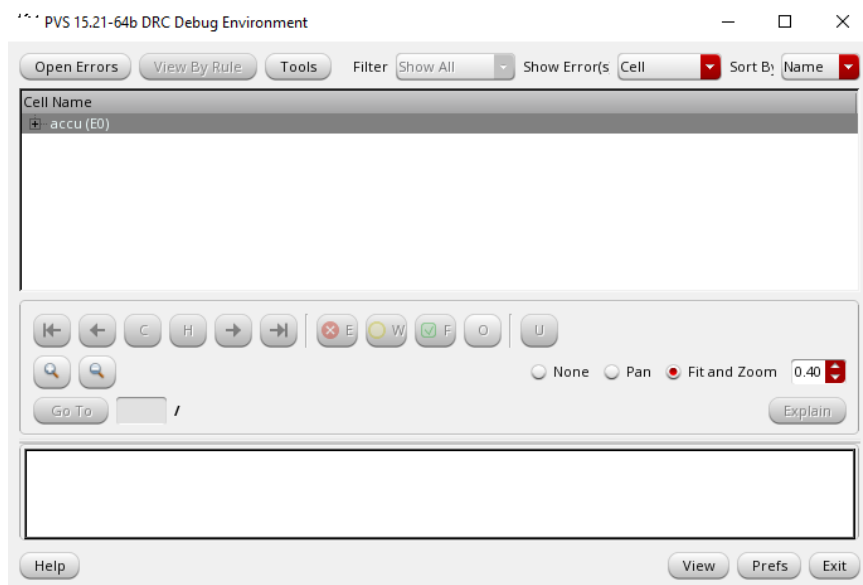


Figure 13: : The DRC (Design Rule Check) window, showing zero errors and a completed process.

4 Conclusion

This report presents a Hamming error correcting code generator and receiver. Hamming codes are used in digital communication systems for error free data transmission. We have implemented a proof of concept system that can correct a single bit error per six transmitted bits. The parameters of the coding scheme are hard coded to keep the system simple and minimal enough to display the basic working principle. However those system parameters can be easily incorporated as system inputs. This report contains behavioral description of the system in verilog, RTL synthesis step, physical design step and validation. Final design of the error correcting logic system is verified to give us the final design.

References

- [1] S. D. Brown, Fundamentals of Digital Logic with Verilog Design. Tata McGraw-Hill Education, 2007.