University of Engineering and Technology, Taxila

Department of Computer Engineering

Final Semester Project - SECDED Report

Course	Computer Organization & Architecture
Course Instructor	Dr. Noshina Ishaq

Group Members

Sr. No	***Name***	***Registration Number***
1-	*Haris Ahmad*	*24-cp-11*
2-	*Shaheer Ijaz*	*24-cp-31*
3-	*Hamza Sajid*	*24-cp-71*

SECDED Error Control Code (Single Error Correction, Double Error Detection)

1) Introduction and Theoretical Background

Error control coding is a fundamental requirement in modern digital systems, particularly in memories, processors, and communication links, where data integrity must be preserved in the presence of noise, transient faults, and hardware disturbances.

A **Single Error Correction (SEC)** Hamming code can detect and correct any single-bit error occurring in a codeword. However, standard SEC Hamming codes cannot reliably distinguish between a single-bit error and certain multi-bit error patterns.

To address this limitation, **SECDED (Single Error Correction, Double Error Detection)** extends the Hamming code by introducing an additional **overall parity bit**.

Comparison: Single-Bit Correction vs. SECDED

The major problem with pure SEC designs is that a double-bit error may produce a non-zero syndrome that mimics a single-bit error, leading the decoder to incorrectly flip a bit and corrupt the data further. SECDED resolves this issue by using the overall parity bit to distinguish between **odd-numbered errors (single-bit)** and **even-numbered errors (double-bit)**.

Feature	Single Error Correction (SEC)	SECDED
Corrects single-bit errors	*Yes*	*Yes*
Detects double-bit errors	*No (may mis correct)*	*Yes*
Detects overall parity-bit error	*No*	*Yes*
Protection against silent data corruption	*Limited*	*Strong*

Table 1-SECDED VS SEC COMPARISON

2) Implementation

Now we will go over the implementation of ***SECDED***, going over the module one by one explaining everything

- *Ports & Signals*
- *Parity Computation*
- *Sequential Write*
- *Re-computation of Parity and overall Parity on Read*
- *Error injection for both single error and double error.*

a) Ports & Signals

This section describes all input signals, output ports, and internal signals used in the SECDED design.

D:/examples/secded.v (/secded) - Default

```
Ln#
 1  module secded(
 2  input clk,
 3  input writeSignal,
 4  input readSignal,
 5  input singleInject,
 6  input doubleInject,
 7
 8  input [3:0] injectIndex1,
 9  input [3:0] injectIndex2,
10  input [7:0] inputData,
11
12
13  output reg noError,
14  output reg oneBitError,
15  output reg parityError,
16  output reg twoBitError,
17  output reg [12:0] outputCodeWord,
18  output reg [7:0] outputCorrectData,
19  output reg [7:0] outputCorruptedData,
20  output reg [3:0] outputSyndrome,
21  output reg outputOverallParity
22  );
23
24  reg [3:0] parityBits;
25  reg [12:0] codeWord;
26  reg overallParity;
27
28  reg [12:0] readCodeWord;
29  reg[3:0] recomputeParity;
30  reg [3:0] syndrome;
31  reg recallParity;
32
33  integer i;
34  integer j;
35  integer weight;
```

Figure 1-ModelSim SnapShot Signals for SECDED-Project

1) Input and Output Ports

Signal Name	Width	Direction	Description
clk	1	Input	System clock used for synchronous write operation
writeSignal	1	Input	Enables encoding and storage of the generated SECDED codeword
readSignal	1	Input	Enables decoding, error detection, and correction logic
singleInject	1	Input	Enables single-bit error injection for verification
doubleInject	1	Input	Enables double-bit error injection for verification
injectIndex1	4	Input	Index of first bit to be flipped during error injection
injectIndex2	4	Input	Index of second bit to be flipped during double-bit injection
inputData	8	Input	Original data word to be encoded
outputCodeWord	13	Output	Encoded SECDED codeword stored in memory
outputCorrectData	8	Output	Corrected data after single-bit error correction
outputCorruptedData	8	Output	Data output in case of detected double-bit error
outputSyndrome	4	Output	Syndrome vector indicating error location
outputOverallParity	1	Output	Recomputed overall parity during decoding
noError	1	Output	Asserted when no error is detected

oneBitError	1	Output	Asserted when a correctable single-bit error is detected
parityError	1	Output	Asserted when only the overall parity bit is in error
twoBitError	1	Output	Asserted when a double-bit error is detected

2) Internal Signals

Signal Name	Width	Description
codeWord	13	Internal encoded SECDED word before storage
parityBits	4	Individual Hamming parity bits
overallParity	1	Computed overall parity during encoding
readCodeWord	13	Codeword used during read and decode operation
recomputeParity	4	Recomputed parity bits during decoding
syndrome	4	XOR result of parity check equations
recallParity	1	XOR of all received bits including overall parity
weight	Integer	Integer representation of syndrome (error position)

1. Parity Code Construction, Parity Placement and Data bit placement

In this section we will go over the parity code construction placements of data bits in corporation with parity bits and overall parity.

1.1 Data Bit Placement

The SECDED codeword consists of **13 bits**, structured as follows: - 8 data bits - 4 Hamming parity bits (positions 1, 2, 4, and 8) - 1 overall parity bit (position 13)

Using 1-based indexing, parity bits occupy positions that are powers of two. In Verilog's 0-based indexing, these correspond to indices 0, 1, 3, and 7. The remaining positions are filled sequentially with the input data bits.

This placement ensures that each parity bit covers a unique combination of data bits, enabling precise error localization.

```
for(i=0;i<13;i=i+1)begin
        if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                codeWord[i] = inputData[j];
                j = j+1;
        end
end
```

Figure 2-ModelSim SnapShot Data bit placement-Project

This for loop ensures that the data bits we are getting as input are placed at the correct position and those positions are

- Other than Parity bits
- Not 12^{th} bit because overall parity will be placed at that position.

2.2 **Hamming Parity Bit Computation**

Each parity bit is computed using XOR over selected positions in the codeword. The selection rule is derived from the binary representation of the bit position: - A parity bit at position 2^i covers all codeword positions whose (1-based) index has the i-th bit set.

This systematic parity calculation ensures that each data bit contributes to multiple parity equations, allowing single-bit error localization through syndrome decoding.

```
        for(i=0 ; i<4;i=i+1)begin
                parityBits[i] = 1'b0;
                for(j=0; j<12 ; j= j+1)begin
                        if(((j+1) & (1<<i)) && j!=((1<<i)-1))begin
                                parityBits[i] = parityBits[i] ^ codeWord[j];
                        end
                end
                codeWord[(1<<i)-1] = parityBits[i];
        end
```

Figure 3-ModelSim SnapShot - parity Bits computation – Project

This for loop is responsible for computation of parity bits

- Since we have 8 parity bits we would need 4 parity bits so the outer loop runs for 4 times and since we will need the data bits with LSB 1 we would need every bit because each parity covers different bits.
- We will have an if condition with j+1 hamming code works with 1 based index while Verilog is 0 index based so we covert the index of Verilog to hamming based index and then we check for the LSB with (1<<i)
- We also exclude the parity itself we are computing from being passed as the argument
- Finally you would ask why the condition for loop is 12 when we have 12 bits in the code word, Answer to that is because 12th position is reserved for overall parity and overall parity is not included in any parity calculation so it is easier to just do not include it in the index than writing another if condition and making it more complex.

2.3 Overall Parity Bit

The overall parity bit is computed as the XOR of all codeword bits excluding itself. Its purpose is to determine whether the total number of flipped bits in the received codeword is odd or even, enabling detection of double-bit errors.

```
        for(i=0; i<13; i=i+1)begin
                if(!(i==12))begin
                        overallParity = overallParity^codeWord[i];
                end
        end
        codeWord[12] = overallParity;
end
```

Figure 4-ModelSim Snapshot overall parity-Project

Here we are computing overall parity through a for loop and then at the end storing it in the code word at 12th bit position.

The overall code for module would look like this

```
always @(*) begin
        codeWord = 13'b0;
        parityBits = 4'b0;
        overallParity = 1'b0;
        i = 0;
        j = 0;
        for(i=0;i<13;i=i+1)begin
                if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                        codeWord[i] = inputData[j];
                        j = j+1;
                end
        end

        for(i=0 ; i<4;i=i+1)begin
                parityBits[i] = 1'b0;
                for(j=0; j<12 ; j= j+1)begin
                        if(((j+1) & (1<<i)) && j!=((1<<i)-1))begin
                                parityBits[i] = parityBits[i] ^ codeWord[j];
                        end
                end
                codeWord[(1<<i)-1] = parityBits[i];
        end

        for(i=0; i<13; i=i+1)begin
                if(!(i==12))begin
                        overallParity = overallParity^codeWord[i];
                end
        end
        codeWord[12] = overallParity;
end
```

Figure 5-ModelSim SnapShot , data placement, parity computation – Project

2. Sequential Write Operation

The encoded SECDED codeword is stored synchronously using a clocked process. The write operation occurs only when writeSignal is asserted at the rising edge of the clock.

This design choice models realistic memory behavior, where writes are edge-triggered and data stability is guaranteed. The separation of combinational encoding logic and sequential storage logic ensures predictable timing and clean verification. We do this because we are simulating a component of memory.

```
always @(posedge clk) begin
        if(writeSignal)begin
                outputCodeWord[12:0] <= codeWord[12:0];
        end

end
```

Figure 6-ModelSim Snapshot, Sequential write -Project

Here we write the encoded data into the output which will be seen in the waveform.

3. Read Operation and Decoding Logic

When readSignal is high, the stored codeword is transferred into an internal register for decoding. At first we pass the default values to the signals we would use for clearing up any old values that would exist in them

```
always @(*) begin
        i = 0;
        j = 0;
        readCodeWord = 13'b0;
        recallParity = 1'b0;
        noError = 1'b0;
        oneBitError = 1'b0;
        parityError=1'b0;
        twoBitError = 1'b0;
        outputSyndrome = 4'b0;
        outputOverallParity = 1'b0;
        readCodeWord[12:0] = outputCodeWord[12:0];
        outputCorrectData   = 8'b0;
        outputCorruptedData = 8'b0;
```

Figure 7-ModelSim snapshot, Default values -project

4. Parity Recalculation

The decoder recomputes all Hamming parity bits using the same selection rules as during encoding. These recomputed parity values are XORed with the stored parity bits to form the **syndrome**.

```
        for(i=0; i<4 ; i = i+1)begin
                recomputeParity[i] = 1'b0;
                for(j=0;j<12;j=j+1)begin
                        if(((j+1)& (1<<i))&& j!=((1<<i)-1)) begin
                                recomputeParity[i] = recomputeParity[i]^ readCodeWord[j];
                        end
                end
                syndrome[i] = recomputeParity[i] ^ readCodeWord[((1<<i)-1)];
        end
```

Figure 8-Modelsim SnapShot , recomputation of parity Bits-project

Here we are doing two things.

- Here we are recomputing the parity bits from the data we are receiving and
- then xoring then with the parity bits of the data we received
- Then generating the syndrome and syndrome is a 4-bit vector whose binary value directly indicates the position of a faulty bit (in the case of a single-bit error). A zero syndrome indicates no error.

4.1 Overall Parity Recalculation

The overall parity is recomputed by XORing all 13 received bits, including the stored overall parity bit. This value distinguishes between odd and even numbers of errors.

```
            end
            syndrome[i] = recomputeParity[i] ^ readCodeWord[((1<<i)-1)];
    end
    for(i=0;i<13;i=i+1)begin
            recallParity = recallParity ^ readCodeWord[i];
    end
```

Figure 9-Modelsim snapshot overall parity re-computation-project

5. Error Classification and Correction

The SECDED decision logic follows a well-defined truth table evaluating overall parity and syndrome at the same time to classify the error either if it is a one bit error or double bit error or just parity error

Syndrome	Overall Parity	Interpretation
0	0	No error
Non-zero	1	Single-bit error (correctable)
0	1	Overall parity bit error
Non-zero	0	Double-bit error (detectable only)

- In the case of a single-bit error, the decoder flips the bit indicated by the syndrome and outputs corrected data.

- If only the overall parity bit is faulty, it is corrected directly.
- In the case of a double-bit error, no correction is attempted; the corrupted data is flagged and reported.
- For Implementation of such truth table we would use if, if else while evaluating each condition

a) No Error

- *(Syndrome ==4'b0 && recallParity = 1'b0)* means no error.

```
if( syndrome== 4'b0 && recallParity == 1'b0 )begin
        noError = 1'b1;
        oneBitError = 1'b0;
        parityError=1'b0;
        twoBitError = 1'b0;
        i = 0;
        j =0;
        outputOverallParity = recallParity;
        outputSyndrome = syndrome;
        for(i=0;i<13;i=i+1)begin
                if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                outputCorrectData[j] = readCodeWord[i];
                j=j+1;
                end
        end
end
```

Figure 10-Modelsim Snapshot Syndrome & parity decoding-project

When both Syndrome and overallParity are 0 that means we have no error in such case we just unzip the data.

b) Bit Flipped

- *(When syndrome != 4'b0 && recallParity == 1'b1)* means one bit is flipped.

```
else if ( syndrome != 4'b0 && recallParity) begin
        noError = 1'b0;
        oneBitError = 1'b1;
        parityError=1'b0;
        twoBitError = 1'b0;
        outputOverallParity = recallParity;
        readCodeWord[weight -1] = ~readCodeWord[weight-1];
        i = 0;
        j =0;
        for(i=0;i<13;i=i+1)begin
                if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                outputCorrectData[j] = readCodeWord[i];
                j=j+1;
                end
        end

end
```

Figure 11-ModelSim Snapshot , One bit flipped -project

When exactly one bit is flipped

- We calculate the weight of the flipped bit because hamming code gives the index of bit but we can not use it as the exact index we need to convert it into the weight of the index having 1 and then -1 from it. It gives us the flipped bit

```
if(syndrome != 4'b0) begin
        weight = 0;
        for(i=0;i<4;i=i+1)begin
                if(syndrome[i]) begin
                        weight = weight + (1<<i);
                end
        end
end
```

Figure 12-ModelSim Snapshot , weight of syndrome-project

- After finding the weight we just pass it to the readCodeWord and then flip the data bit and then unzip it using the for loop.

c) Overall Parity Flip

- *(syndrome == 4'b0 && recallParity)* means the overallParity is flipped

```
else if(syndrome == 4'b0 && recallParity)begin
        noError = 1'b0;
        oneBitError = 1'b0;
        parityError=1'b1;
        twoBitError = 1'b0;
        readCodeWord[12] = ~readCodeWord[12];
        outputOverallParity = recallParity;
        i = 0;
        j =0;
        for(i=0;i<13;i=i+1)begin
                if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                outputCorrectData[j] = readCodeWord[i];
                j=j+1;
                end
        end

end
```

Figure 13-Modelsim Snapshot overall parity flip – project

Here when overall parity is flipped we just flip the 12th bit of the read data and the data is corrected

d) 2-bit Error

- *(syndrome !=4'b0 && recallParity == 1'b0)* means if the syndrome is not equal to zero and recallParity is 0 it means two bits have flipped

```
        end
        else if(syndrome !=4'b0 && recallParity == 1'b0) begin
                noError = 1'b0;
                oneBitError = 1'b0;
                parityError=1'b0;
                twoBitError = 1'b1;
                outputOverallParity = recallParity;
                i = 0;
                j =0;
                for(i=0;i<13;i=i+1)begin
                        if(!(i==0 || i==1 || i==3 || i==7 || i==12)) begin
                        outputCorruptedData[j] = readCodeWord[i];
                        j=j+1;
                        end
                end
        end
end
```

Figure 14-ModelSim Snapshot , 2bit error -project

When two bits are corrupted or flipped

- we just unzip the corrupted data and pass it to the output register outputCorruptData
- Raise the twoBitError signal indicating that two-bit error has occurred we do not try to solve it

6. Error Injection Mechanism

To validate the SECDED design, We added

- ***SingleInject***-Signal for single bit error injection.
- ***doubleInject***-Signal for double error injection.

The design allows: - Single-bit error injection allow flipping one selected bit. Double-bit error injection by flipping two independent bits

This mechanism enables systematic verification of all operational scenarios, including correct detection, correction, and error flagging.

These Two signals allow depending on if you want to inject single error or double error allow you to pass two indices and flip the bit of those indices.

```
        if(readSignal)begin
                if(singleInject) begin
                        readCodeWord[injectIndex1] = ~ readCodeWord[injectIndex1];
                end
                else if(doubleInject)begin
                        readCodeWord[injectIndex1] = ~ readCodeWord[injectIndex1];
                        readCodeWord[injectIndex2] = ~ readCodeWord[injectIndex2];
                end
```

Figure 15-ModelSim Snapshot , Error Injection – project

7. Verification and Waveform Analysis

Now we will verify our system by compiling it and then creating a waveform expecting

- When no bit is flipped the ***NoError*** signal should be ***high*** while all rest should be ***zero***.
- When single bit is flipped ***OnebitError*** should be ***high*** while others are ***zero***.
- When Overall parity bit is flipped only ***parityError*** signal should be ***high***
- When two bits are flipped ***twoBitError*** should be ***high*** only.

7.1) No bit flip

When we read data with no error injection and both syndrome and recomputed parity are 0 at the same time

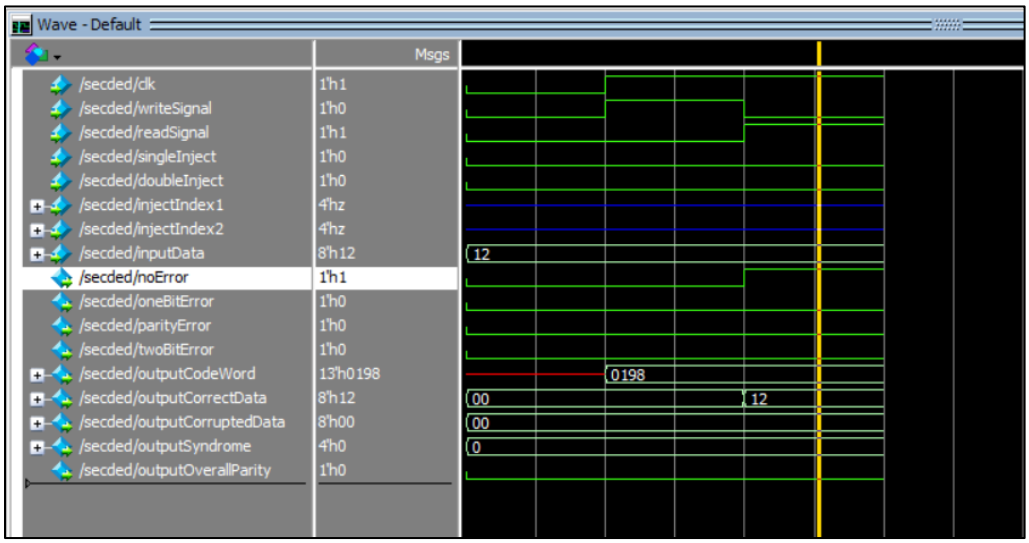

Figure 16-ModelSim Snapshot no error waveform-project

Here we can see that

- ***NoError*** signal is ***high***, indicating that there is not any ***error***.
- ***OverallParity*** remains zero indicating that data's integrity is ***maintained***.

7.2) One-bit Error

When exactly one bit is flipped we get syndrome != 4'b0 and recomputed parity 1 it indicated that the one-bit error occurred.

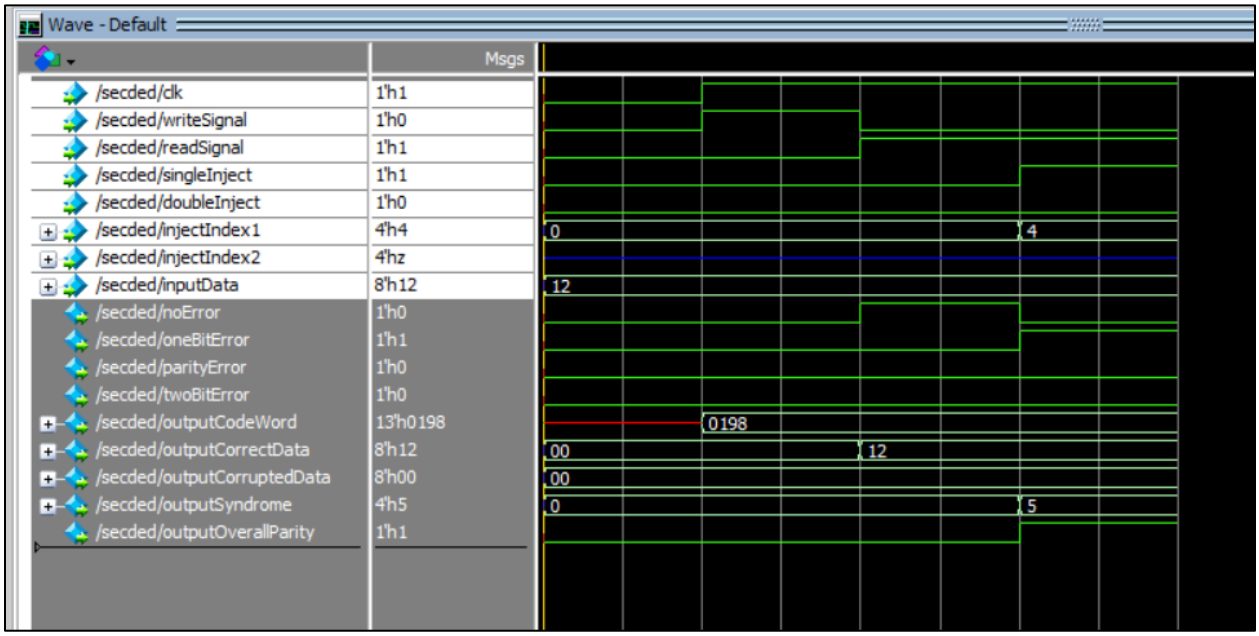

Figure 17-ModelSim SnapShot, One-bit error-project

Here we can see that

- ***OneBitError*** is high indicating that exactly one bit flipped
- ***OverallParity*** becomes 1 indicating that the ***overallParity*** of the data is corrupted
- ***OutputSyndrome*** is 5 indicating that ***syndrome*** is not zero and it is correctly outputting the index of the flipped bit
- Notice we flipped the bit at ***4th position***, but syndrome is indicating the flipped bit is at ***5th position*** that is because syndrome is following the ***1-based index*** of ***hamming code*** and that is the reason why we – 1 from the weight before flipping the bit.

- The ***outputCorrectData*** remains unchanged because we have implemented error correction

7.3) Parity bit Error

When the overall parity itself is flipped in the data we just flip the parity bit from the read data and raise the parityError signal.

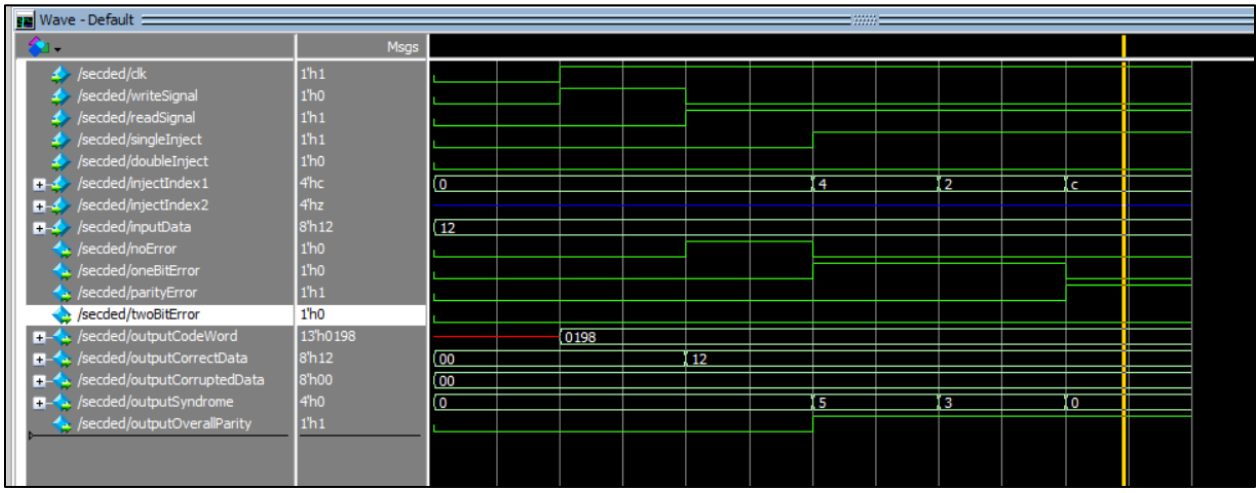

Figure 18-ModelSim SnapShot Overall Parity error-project

Here we can see when we flip the 12th bit in error injection

- ***parityError*** is high while rest are zero indicating it is the overall parity that is flipped.
- As expected the ***syndrome*** is 0 because for computation of ***syndrome overall parity*** bit is not used
- ***OverallParity*** is at ***high*** indicating that it is ***overallParity*** that is being flipped.

7.4) Two-Bit Error

When syndrome is not equal two zero but overallParity is 0 then two bit error signal becomes high indicating that 2 bits have flipped.

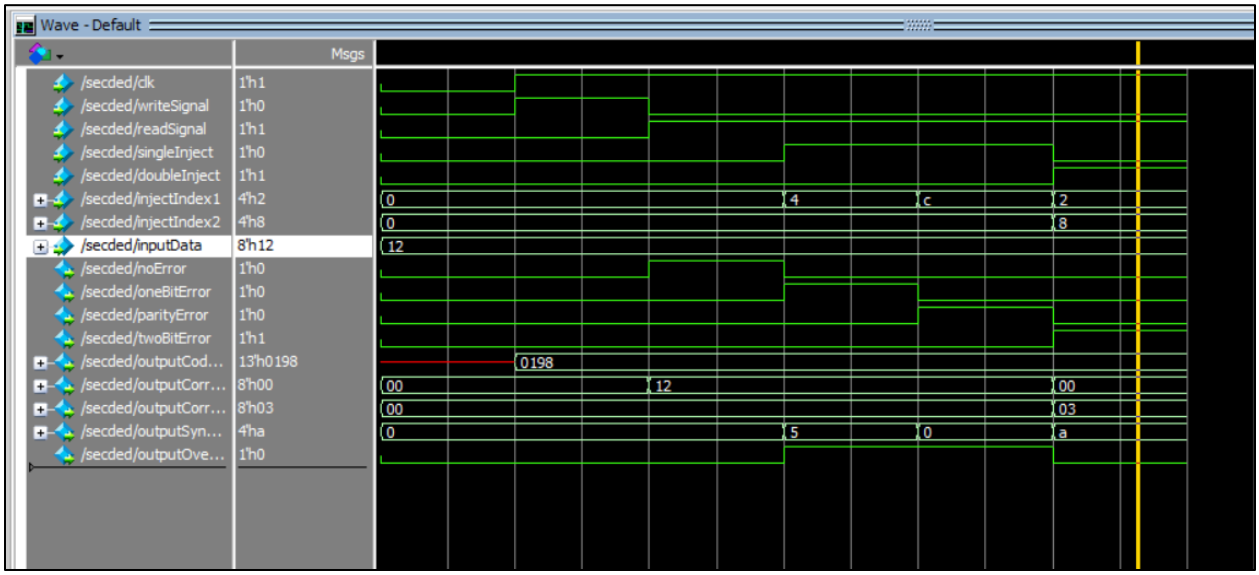

Figure 19-Modelsim SnapShot 2-bit flip error-project

Here we can see we are injecting double error flipping two bits at the same time and we are getting

- ***TwobitError*** as high indicating two bits are flipped.
- ***Syndrome*** is not zero, it is pointing to an index.
- ***OverallParity*** is 0 indicating that the integrity of data is not damaged, but we know two bits have flipped from our truth table

Hence after covering these things we can say that our Single Error Correction, double Error detection is working properly.

8. Conclusion

This project successfully implements a complete SECDED system capable of correcting single-bit errors and detecting double-bit errors. By extending a basic Hamming SEC design with an overall parity bit, the system eliminates the risk of silent data corruption caused by multi-bit faults. The modular structure, clear separation between combinational and sequential logic, and comprehensive verification make this design suitable for real-world memory and processor applications.