Experiment 1: Error Detection using Parity

Aim: To apply Parity check rules for error detection

Objective: After carrying out this experiment, students will be able to:

- Apply 1D and 2D parity rules for error detection
- Analyze the difference between 1D and 2D parity and their limitations

Problem statement: You are required to write separate programs to demonstrate the use of 1D and 2D parity. Take the input bit streams (max five) of 7 bit each from the user. Your programs should calculate the parity and display the input and output bit streams.

Analysis: While analyzing your program, you are required to address the following points:

- Why can this method not be used to correct errors?
- How are 1D and 2D parity different?
- What are the limitations of this method of error detection?

MARKS DISTRIBUTION

Component	Maximum Marks	Marks Obtained
Preparation of Document	7	
Results	7	
Viva	6	
Total	**20**	

Submitted by: Satyajit Ghana

Register No: 17ETCS002159

1. Algorithm/Flowchart

```
1D-Parity(data):

1. bit_count = 0
2. for_each(bit : data)
3.    if (bit == 1) bit_count++
4. if (bit_count % 2 == 0)
5.    parity_bit = 0
6. else
7.    parity_bit = 1


2D-Parity(data, nsplits):
1. rows = nsplits
2. cols = data.size() / rows
3. for every row in data
4.    row_parity_bits.push_back(calc_parity_bit(row))
5. for every col in data
6.    col_parity_bits.push_back(calc_parity_bit(col))
```

2. Program

1D-Parity

main.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>

#include "parity.hpp"

int main(int, char**) {

    std::vector<unsigned short> sender_data;
    std::vector<unsigned short> receiver_data;

    std::string input_line;
    std::cout << "Enter the Data from Sender : ";
    std::getline(std::cin, input_line);
    for (const char &c : input_line) {
        if (c != ' ') {
            sender_data.push_back(c-'0');
```


```
        }
    }
    char parity;
    std::cout << "Enter Parity (E/O) : ";
    std::cin >> parity;

    PARITY_OPTION parity_option = parity == 'e' ? EVEN : ODD;

    ParityData pdata(sender_data, parity_option);

    std::cout << "Parity Bit : " << pdata.get_parity_bit() << std::endl;

    std::cout << "Data Transmitted : " << input_line << pdata.get_parity_bit() <<
std::endl;

    std::cout << "Enter the Data from Receiver : ";
    std::cin.ignore();
    std::getline(std::cin, input_line);
    unsigned short parity_bit = input_line.back() - '0';
    input_line.pop_back();
    for (const char &c : input_line) {
        if (c != ' ') {
            receiver_data.push_back(c-'0');
        }
    }

    bool is_correct = pdata.check_with_received(receiver_data, parity_bit);

    is_correct ? std::cout << "CORRECT" : std::cout << "INVALID";

    std::cout << std::endl;
}
```

parity.cpp

```
#include "parity.hpp"

ParityData::ParityData(const std::vector<unsigned short>& data, PARITY_OPTION
parity_option = EVEN) : data(data) {
    this -> parity_option = parity_option;
    this -> parity_bit = ParityData::calculate_parity_bit(data, this -> parity_option);
}
```


```
unsigned short ParityData::calculate_parity_bit(const std::vector<unsigned short>& data,
PARITY_OPTION parity_option) {
    unsigned count = 0;

    for (auto e : data) {
        if (e == 1) {
            count++;
        }
    }

    bool is_count_even = count % 2 == 0;

    unsigned short parity_bit;

    switch(parity_option) {
        case EVEN:
            parity_bit = is_count_even == true ? 0 : 1;
            break;
        case ODD:
            parity_bit = is_count_even == true ? 1 : 0;
            break;
    }

    return parity_bit;
}

bool ParityData::check_with_received(const std::vector<unsigned short>& to_check, const
unsigned short& parity_bit) {
    unsigned short parity_bit_expected = ParityData::calculate_parity_bit(to_check, this
-> parity_option);
    bool is_correct = parity_bit == parity_bit_expected ? true : false;

    return is_correct;
}
```

parity.hpp

```
#pragma once

#include <vector>

enum PARITY_OPTION {
    EVEN,
```


```
    ODD
};

class ParityData {
    public:
        ParityData(const std::vector<unsigned short> &data, PARITY_OPTION option);
        bool check_with_received(const std::vector<unsigned short>& data, const unsigned
short& parity_bit);
        static unsigned short calculate_parity_bit(const std::vector<unsigned short>
&data, PARITY_OPTION option);
        unsigned short get_parity_bit() { return this -> parity_bit; };
    private:
        const std::vector<unsigned short> data;
        unsigned short parity_bit;
        PARITY_OPTION parity_option;
};
```

2D-Parity

main.cpp

```
#include <iostream>
#include <string>
#include <vector>

#include "parity.hpp"

template<typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T> &input) {
    for (auto const& i: input) {
        os << i << " ";
    }
    return os;
}

// 2D Parity Program
int main(int, char**) {

    std::vector<unsigned short> sender_data;
    std::vector<unsigned short> receiver_data;

    std::string input_line;
    std::cout << "Enter the Data from Sender : ";
    std::getline(std::cin, input_line);
    for (const char &c : input_line) {
```


```
        if (c != ' ') {
            sender_data.push_back(c-'0');
        }
    }
    char parity;
    std::cout << "Enter Parity (E/O) : ";
    std::cin >> parity;

    PARITY_OPTION parity_option = parity == 'e' ? EVEN : ODD;

    unsigned short splits;
    std::cout << "Enter how many splits you want : ";
    std::cin >> splits;

    // create the parity data object
    ParityData pdata(sender_data, splits, parity_option);

    std::cout << "Enter the Data from Receiver(row-wise) with parity bits : ";
    std::getline(std::cin, input_line);
    input_line.clear();
    std::getline(std::cin, input_line);
    for (const char &c : input_line) {
        if (c != ' ') {
            receiver_data.push_back(c-'0');
        }
    }

    // check if the data received is correct
    bool is_correct = pdata.check_with_received(receiver_data);

    is_correct ? std::cout << "CORRECT" : std::cout << "INVALID";

    std::cout << std::endl;
}

// OUTPUT
// Enter the Data from Sender : 1100
// Enter Parity (E/O) : e
// Enter how many splits you want : 2
// Enter the Data from Receiver with parity bits : 11000011
// CORRECT
```

parity.cpp

```
#include "parity.hpp"

#include <iostream>

ParityData::ParityData(
    const std::vector<unsigned short>& data,
    unsigned short split,
    PARITY_OPTION parity_option = EVEN) {
        if (data.size() % split != 0) {
            throw "CANNOT SPLIT!";
        }
        this -> parity_option = parity_option;

        unsigned rows = split;
        unsigned cols = data.size() / split;
        this -> nrows = rows;
        this -> ncols = cols;

        // create our data matrix
        for (unsigned i = 0 ; i < rows ; i++) {
            std::vector<unsigned short> temp;
            for (unsigned j = 0 ; j < cols ; j++) {
                temp.push_back(data.at( i*cols+j ));
            }
            this -> data.push_back(temp);

            // meanwhile calculate the parity bit and push it as well
            unsigned short row_parity_bit = ParityData::calculate_parity_bit(temp,
parity_option);
            this -> row_parity_bits.push_back(row_parity_bit);
        }

        // calculate the column parity bits
        for (unsigned j = 0 ; j < cols ; j++) {
            std::vector<unsigned short> temp;
            for (unsigned i = 0 ; i < rows ; i++) {
                temp.push_back(this -> data.at(i).at(j));
            }

            unsigned short col_parity_bit = ParityData::calculate_parity_bit(temp,
parity_option);
            this -> column_parity_bits.push_back(col_parity_bit);
        }
```


```
        for (unsigned i = 0 ; i < rows ; i++) {
            for (unsigned j = 0 ; j < cols ; j++) {
                std::cout << this -> data.at(i).at(j);
            }
            std::cout << this -> row_parity_bits.at(i);
        }
        for (unsigned i = 0 ; i < this -> column_parity_bits.size() ; i++) {
            std::cout << this -> column_parity_bits.at(i);
        }
        std::cout << std::endl;

}

unsigned short ParityData::calculate_parity_bit(const std::vector<unsigned short>& data,
PARITY_OPTION parity_option) {
    unsigned count = 0;

    for (auto e : data) {
        if (e == 1) {
            count++;
        }
    }

    bool is_count_even = count % 2 == 0;

    unsigned short parity_bit;

    switch(parity_option) {
        case EVEN:
            parity_bit = is_count_even == true ? 0 : 1;
            break;
        case ODD:
            parity_bit = is_count_even == true ? 1 : 0;
            break;
    }

    return parity_bit;
}

bool ParityData::check_with_received(const std::vector<unsigned short>& to_check) {

    // check the row parity bits
    for (unsigned i = 0 ; i < this -> nrows ; i++) {
        unsigned parity_found = to_check.at((i * (this -> ncols + 1)) + this -> ncols);
```


```
        unsigned parity_expected = this -> row_parity_bits.at(i);

        if (parity_found != parity_expected) {
            return false;
        }
    }

    // check the column parity bits
    for (unsigned j = 0 ; j < this -> ncols ; j++) {
        // do the math okay ? i'm dumb i have no clue how i did this
        unsigned parity_found = to_check.at( (this -> nrows * (this -> ncols + 1)) + j);
        unsigned parity_expected = this -> column_parity_bits.at(j);

        if (parity_found != parity_expected) {
            return false;
        }
    }

    return true;
}
```

parity.hpp

```
#pragma once

#include <vector>

enum PARITY_OPTION {
    EVEN,
    ODD
};

class ParityData {
    public:
        ParityData(const std::vector<unsigned short> &data,  unsigned short split,
PARITY_OPTION option);
        bool check_with_received(const std::vector<unsigned short>& data);
        static unsigned short calculate_parity_bit(const std::vector<unsigned short>
&data, PARITY_OPTION option);
    private:
        std::vector<std::vector<unsigned short>> data;
        std::vector<unsigned short> row_parity_bits;
        std::vector<unsigned short> column_parity_bits;
```


```
        unsigned nrows, ncols;
        PARITY_OPTION parity_option;
};
```

3. Results

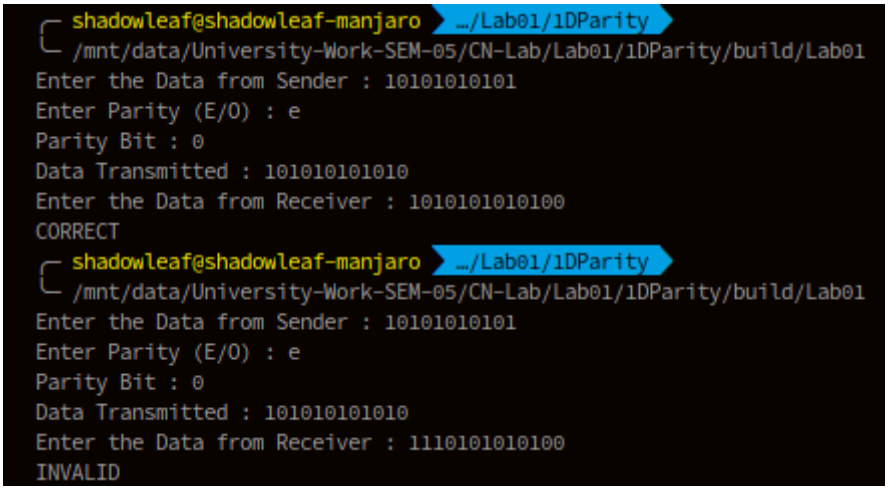

Figure 0-1 1D-Parity

Here the data taken is of 11 bits, the corresponding parity bit is calculated, the receiver then gets the data and checks with the parity bit, here we've taken two examples, in example 1 the data is intact, hence the output is CORRECT, while in the next example the data is mutated and the message outputs INVALID.

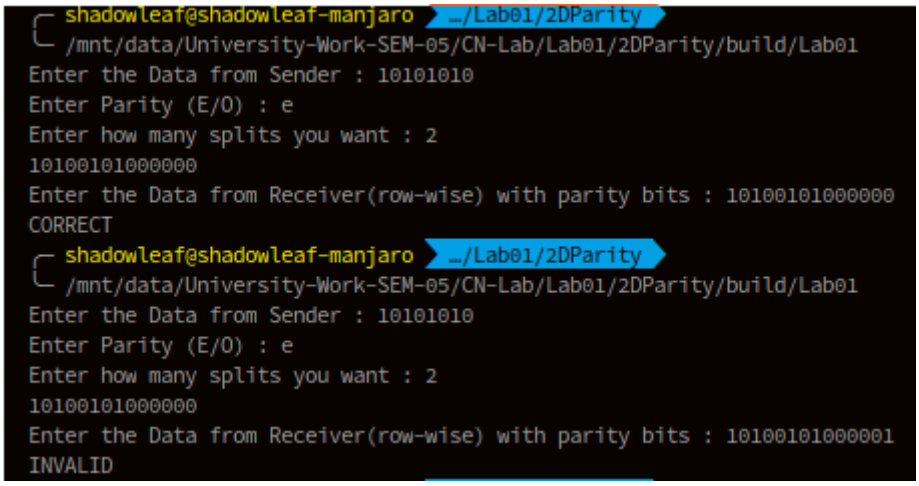

Figure 0-2 2D-Parity

In 2D parity, the data is given and also the number of splits is asked from the user, the data is correspondingly split and the row, column parity bits are calculated and stored in a vector, the complete data along with the parity bits are displayed, this data is transmitted, on the receiver end the data is taken and the row, column parity bits are checked, if they are correct then CORRECT is displayed, else INVALID, in the above output, the first output shows when the receiver gets the intact data and in the latter part the data is altered.

4. Analysis and Discussions

- Why can this method not be used to correct errors?

Parity Bit for error detection is prone to data mutations, such as in 1D-Parity is even number of bits are simultaneously mutated then the parity bits remain unchanged, the receiver will not be able to detect the error in the data. Same goes to 2D-Parity where if the bits in the data grid are mutated in a square region the receiver will not be able to detect corrupted data. Another problem with this method is that the parity bit itself can mutate, which will represent the data to be corrupted even though the data transmitted might be correct.

- How are 1D and 2D parity different?

1D Parity has only one parity bit, while 2D parity has multiple bits based on how the data is divided, 2D parity can detect burst errors, while 1D can only detect single bit errors or odd-number of bit errors.

- What are the limitations of this method of error detection?

Parity bits cannot detect all forms of bit error

For 2D Parity, rows + column number of parity bits are added that are to be transmitted along with data, more data, means that there's more chances of corruption.

5. Conclusions

Parity bits are a very simple way to check if the transmitted data is corrupted on the way, although it cannot detect all forms of errors, it is very simple to implement.

6. Comments

   a. Limitations of the experiment

The experiment assumes the data to be binary values, although in the real world the data is taken as bit streams, the experiment is limited to proof of concept.

b. Limitations of the results obtained

The results obtained can be checked for bitstrings, and does not work on generic data.

c. Learning

The concept of Parity bits (1D and 2D) was learnt in this lab.

d. Recommendations

Instead of using an array/vector to represent the data, the data should be taken as byte streams and encoded accordingly, such as a string, or a binary file, this was the program will be more generic.