Abstract:

Because of soft error the reliability of modern system facing many problem. As the area of application increase, the demand of detecting and correcting errors increase proportionally. Several errors correcting method has been developed to solve this problem. Hamming code , parity check matrix, column–line–code (CLC) are notable among them. They can detect and correct single bit error with 100% accuracy. In this paper, some new methodology is proposed which can correct up to 10 adjacent bits of data error and detect 11 adjacent bits error of a 16 bit data word. To do it we have to generate an H matrix and calculate the code bit and syndrome bit. Thus the reliability and error tolerance of the system increase. Our experimental studies show that its redundancy and error detection and correction capability is better then the previous methods

Keyword: Soft Error Tolerance; Reliability, Redundancy

Introduction:

With the advancement of technology computation technologies has become more complex. The size of the microprocessor is decreasing exponentially with the increase of the technology advancement. With such advancement there is now a need for fast and more data transfer capability. Thus redundancy of the chips increase. Besides when data is transferred from one unit to another, from one system to another, or even while the data are stored in a memory error may occurred. This type of soft error may cause lots of harm to the system.

Soft error in the systems presents as a threat to reliability. Thus some actions should be taken against this type of soft error to make the systems more error free and more error tolerant. This is how we came in need of error-correction-code (ECC) which is used to protect the data memory.

In today’s systems, hamming code and parity check matrix are used to detect this type of soft error. They can detect and correct single bit error. The reliability of this type of codes are very low. As they are not able to check more error bit code error. Later, column–line–code (CLC) is the extended form of hamming code. CLC can correct up to 2 bit error but the redundancy of CLC is 150% which is so much high.

Later SEC–double-adjacent error detection (SEC–DAED) is introduced by Saiz-Adalid et al. which has a low redundancy. SEC-DEAD use 5 code bits and able to correct all one bit errors and detect all two bit adjacent error in a 16 bit data word. As it’s redundancy is low compared to the previous one, it can give more space for the data bits in the memory than the previously describe methods.

After that Saiz-Adalid *et al*. developed flexible unequal error control (FUEC) methodology. FUEC codes are an improvement of the wellknown unequal error control (UEC) codes. [2].

By using this methodology FUEC-DAEC, FUEC-TAEC, FUEC-QUAEC [1] was invented which can correct single bit error, up to 4 bit error and detect 4 bit errors.

In this paper, a new approach for error detection and correction method is proposed to protect against soft errors. This method used FUEC methodology to detect and correct error.

This proposed method provides 100% error correction for up to 10 adjacent bit and detection up to 11 adjacent bit error in a 16 bit data word.

The paper is organized as follows, section II some related works in this field of study is discussed upon the matter of error correction rate and overall system reliability. In section III, our proposed methodology is discussed briefly. In section IV we analyzed the performance of our system. Section V contains concluding remarks.

Related work:

In the past hamming code and parity check matrix were used to detect and correct any bit error. Hamming code can be easily developed for any data length. But the redundancy of parity check matrix is high.

Later SEC-DEAD error correction code was introduced, which requires 5 code bit for 16 bit data word. Thus it has a low redundancy But it can only detect and correct single bit or two adjacent bits error.

After that another technique, FUEC-DAEC, FUEC-TAEC, FUEC-QUAEC was invented which can detect and correct from any single bit error to 4 bit adjacent error The H matrix of FUEC-DAEC is given below:

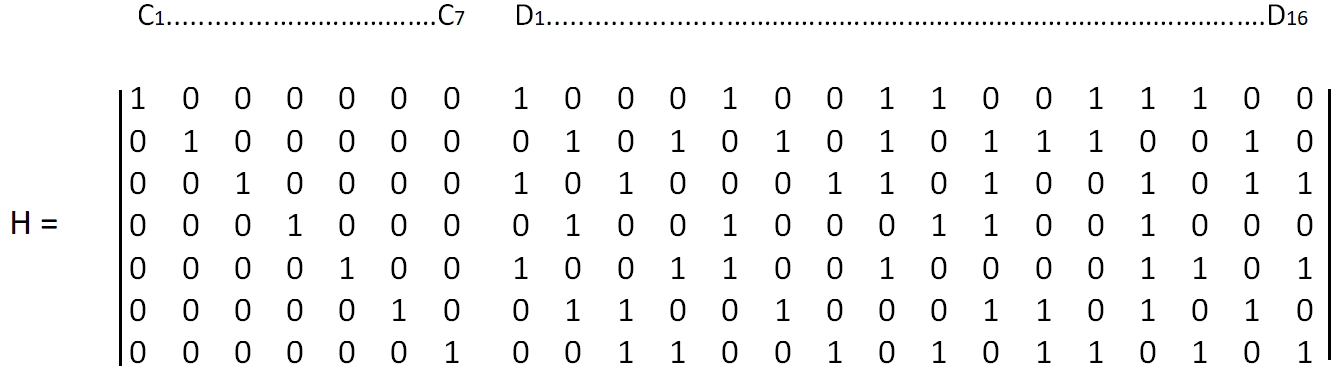


Fig: Parity-check matrix H for the FUEC–DAEC code.

our approach:

To improve the FUEC methodology from its limitation of correcting and detecting maximum 4 adjacent data bits we have designed several new matrices. These matrices are based on the previously proposed FUEC method but they can correct up to 10 adjacent data bit error and and have error detection capability of up to 11 adjacent data bit.

The required H matrices are given below:

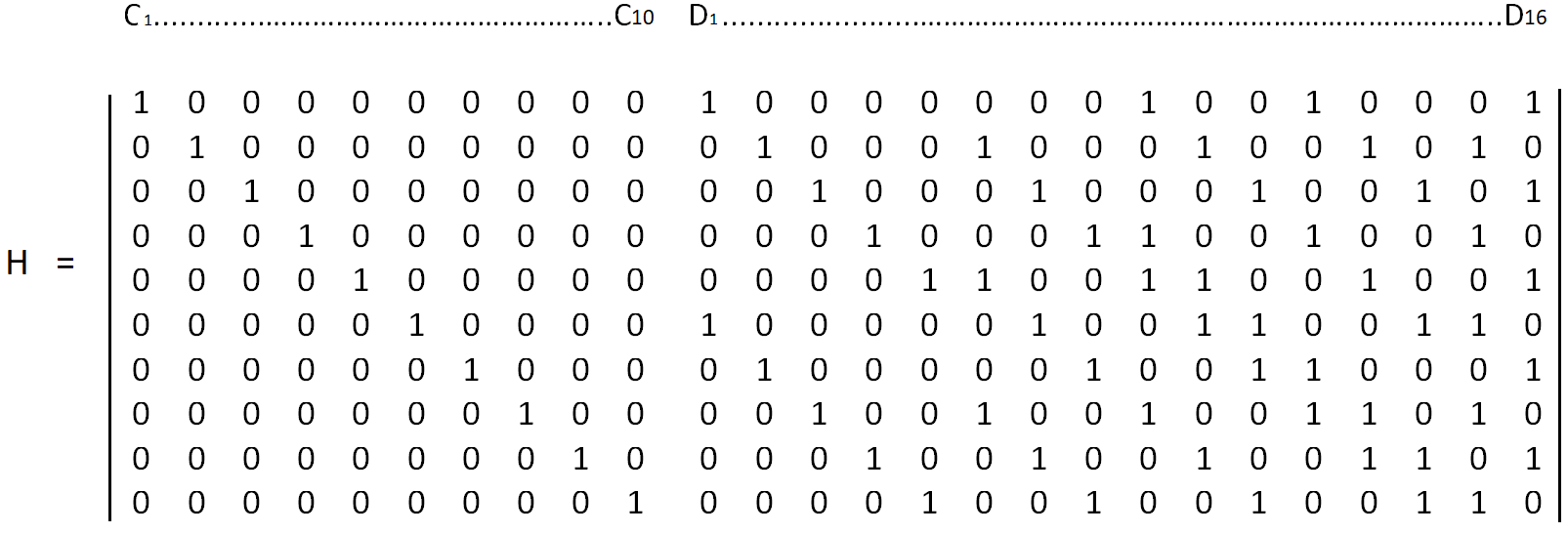


Fig: Parity-check matrix H for the (26, 16) FUEC–PAEC code.

Here Ci are the code bits and Di are the data bits.

Using this process , we are able to correct from a single bit error to maximum 5 bit adjacent error and it is capable of detecting up to 6 bit adjacent error

In FUEC-PAEC it needs a total 10 code bits for 16 data bits.

Once the H matrix is generated , we calculate the code bits. To calculate the code bits we do X-OR operation . But the formula to do X-OR operation is a bit different. The truth table of X-OR operation is given below:

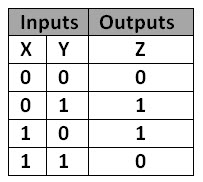


Fig: X-OR truth table

Here we can see that if the number of 1 is even then the output value is 0 else the output value is 1. So we count the number of 1 in data bits. And then we used total number of 1’s mod 2 to check either the total number of 1 is even or odd. If the answer is even then the value of code bit is 0 else the value of code bit is 1.

Let , D1 is the actual data bit and D2 is the error data bit.

Then the formula to calculate code bit is given below:

C[0] = ( D1[0]  +  D1[8]  +  D1[11]  +  D1[15] ) mod 2

C[1] = ( D1[1] + D1[5] + D1[9] + D1[12] + D1[14] ) mod 2

C[2] = ( D1[2] + D1[6] + D1[10] + D1[13] + D1[15] ) mod 2

C[3] = ( D1[3] + D1[7] + D1[8] + D1[11] + D1[14] ) mod 2

C[4] = ( D1[4] + D1[5] + D1[8] + D1[9] + D1[12] + D1[15] ) mod 2

C[5] = ( D1[0] + D1[6] + D1[9] + D1[10] + D1[13] + D1[14] ) mod 2

C[6] = ( D1[1] + D1[7] + D1[10] + D1[11] + D1[15] ) mod 2

C[7] = ( D1[2] + D1[5] + D1[8] + D1[11] + D,[12] + D1[14] ) mod 2

C[8] = ( D1[3] + D1[6] + D1[9] + D1[12] + D1[13] + D1[15] ) mod 2

C[9] = ( D1[4] + D1[7] + D1[10] + D1[13] + D1[14] ) mod 2

After calculating the code bits , we will calculate the syndrome bits by doing the X-OR operation between actual data check bits and erroneous data check bits . The formula to calculate syndrome bit is given below:

S[0] = ( C[0] + D2[0] + D2[8] + D2[11] + D2[15] ) mod 2

S[1] = ( C[1] + D2[1] + D2[5] + D2[9] + D2[12] + D2[14] ) mod 2

S[2] = ( C[2] + D2[2] + D2[6] + D2[10] + D2[13] + D2[15] ) mod 2

S[3] = ( C[3] + D2[3] + D2[7] + D2[8] + D2[11] + D2[14] ) mod 2

S[4] = ( C[4] + D2[4] + D2[5] + D2[8] + D2[9] + D2[12] + D2[15] ) mod 2

S[5] = ( C[5] + D2[0] + D2[6] + D2[9] + D2[10] + D2[13] + D2[14] ) mod 2

S[6] = ( C[6] + D2[1] + D2[7] + D2[10] + D2[11] + D2[15] ) mod 2

S[7] = ( C[7] + D2[2] + D2[5] + D2[8] + D2[11] + D2[12] + D2[14] ) mod 2

S[8] = ( C[8] + D2[3] + D2[6] + D2[9] + D2[12] + D2[13] + D2[15] ) mod 2

S[9] = ( C[9] + D2[4] + D2[7] + D2[10] + D2[13] + D2[14] ) mod 2

In our second scheme by using FUEC-HXAEC method we are able to correct from single bit to 6 bit adjacent error. This method can also detect up to 7 bit adjacent error . In FUEC-HXAEC it needs a total 11 code bits for 16 data bits.

Like FUEC-PAEC , we can easily calculate the code bits and syndrome bits for FUEC-HXAEC.

The H matrix for FUEC-HXAEC is given below :

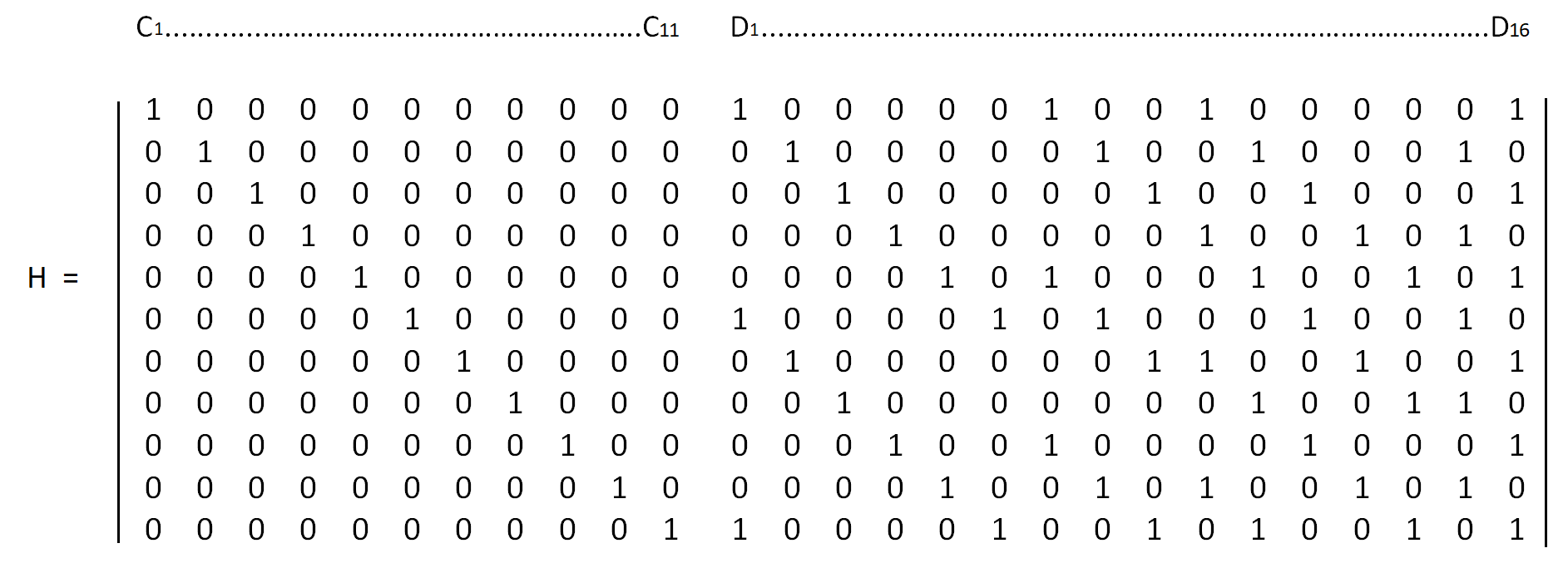


Fig: Parity-check matrix H for the (27, 16) FUEC–HXAEC code.

In our second scheme by using FUEC-HEPAEC method we are able to correct from a single bit to 7 bit adjacent error. This method can also detect up to 8 bit adjacent error . In FUEC-HXAEC it needs a total 12 code bits for 16 data bits.

Like FUEC-PAEC and FUEC-HXAEC, we can easily calculate the code bits and syndrome bits for FUEC-HEPAEC.

The H matrix for FUEC-HEPAEC is given below :

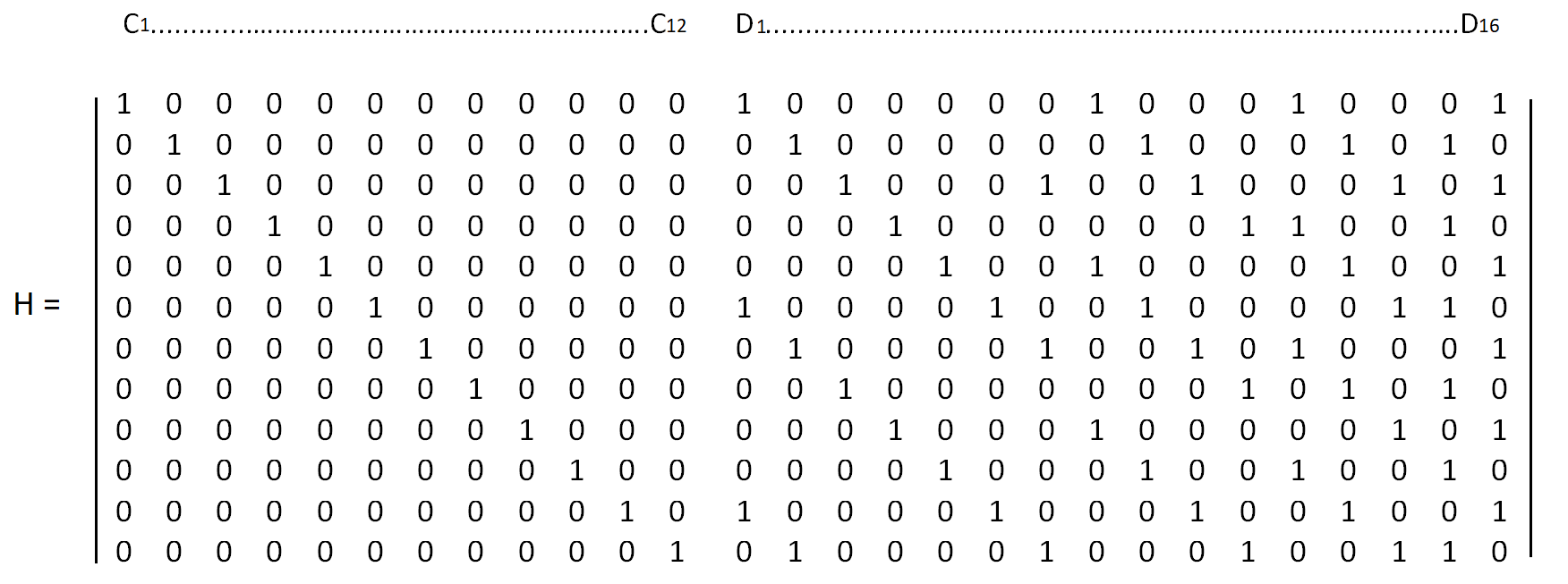


Fig: Parity-check matrix H for the (28, 16) FUEC–HEPAEC code.

In our forth scheme by using FUEC-OCAEC method we can correct up to 8 bit adjacent error and detect up to 9 bit adjacent error . In FUEC-OCAEC it needs a total 13 code bits for 16 data bits.

Like FUEC-PAEC and FUEC-HXAEC, we can easily calculate the code bits and syndrome bits for FUEC-OCAEC.

The H matrix for FUEC-OCAEC is given below :

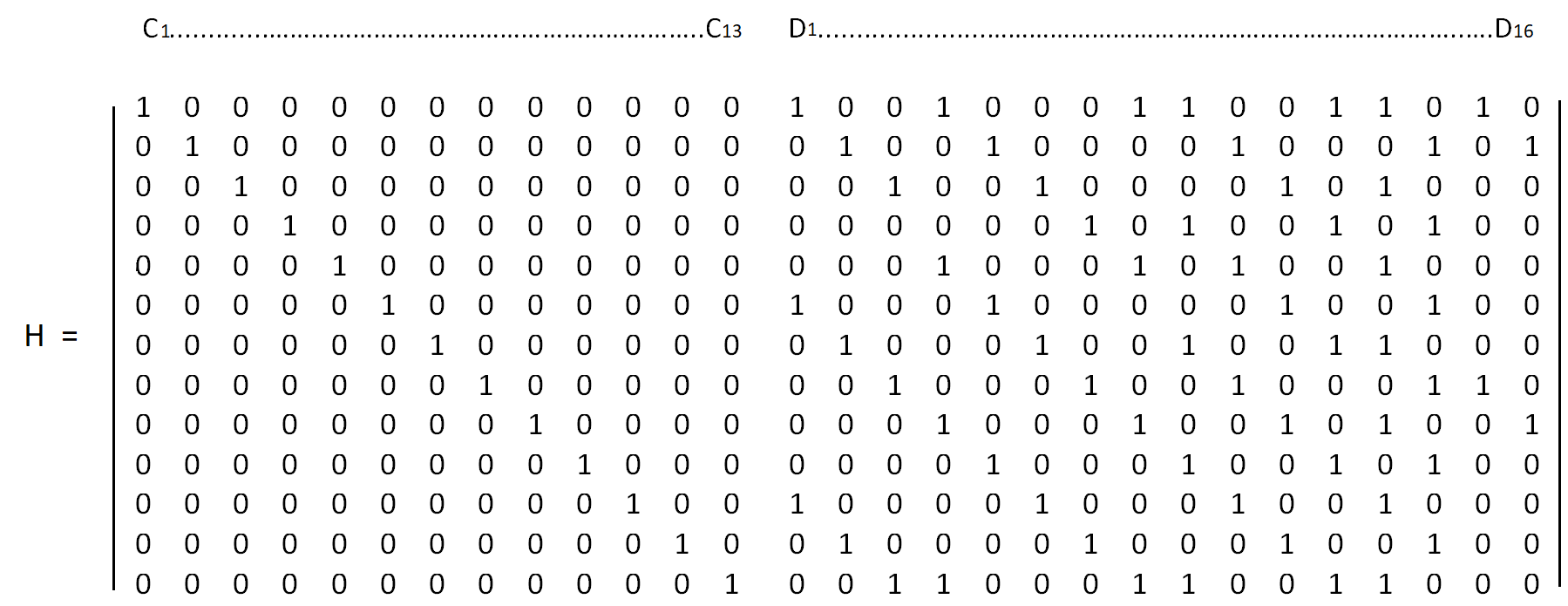
****

Fig: Parity-check matrix H for the (29, 16) FUEC–OCAEC code.

In our fifth scheme by using FUEC-ENAEC method we can correct up to 9 bit adjacent error and detect up to 10 bit adjacent error . In FUEC-OCAEC it needs a total 14 code bits for 16 data bits.

Like FUEC-PAEC and FUEC-HXAEC, we can easily calculate the code bits and syndrome bits for FUEC-ENAEC.

The H matrix for FUEC-ENAEC is given below :

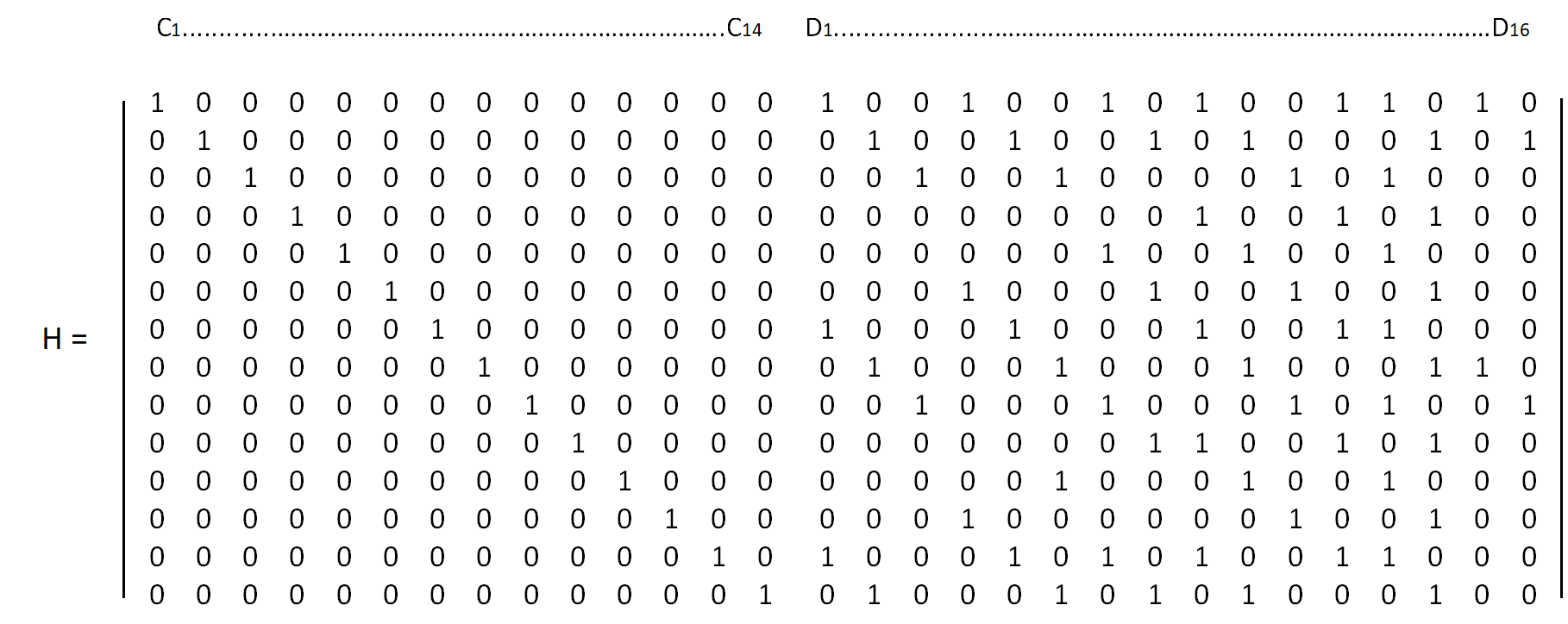


Fig: Parity-check matrix H for the (30, 16) FUEC–ENAEC code.

In our sixth scheme by using FUEC-DKAEC method we can correct up to 10 bit adjacent error and detect up to 11 bit adjacent error . In FUEC-DKAEC it needs a total 15 code bits for 16 data bits.

We can calculate the code bits and syndrome using the same process as before

The H matrix for FUEC-DKAEC is given below :

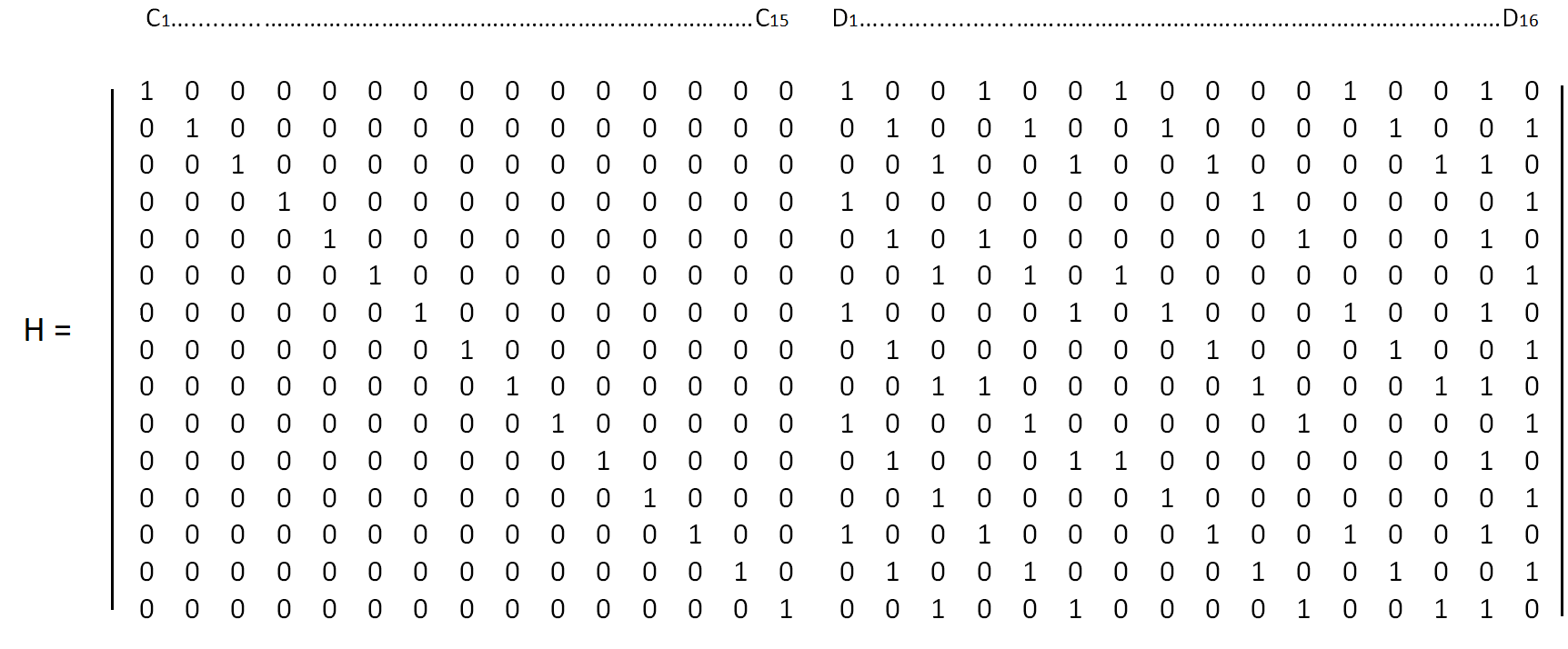


Fig: Parity-check matrix H for the (31, 16) FUEC–DKAEC code.

Error Injection:

To verify our work we had to work with all possible combinations for 16 bit data. Which was 2^16=65,536 in total. And then we injected variable bits of adjacent errors to check the accuracy of the method. This combination decreases with the increase of the number of error. For example for 5 bit adjacent error injection there will be (2^16)\*12 combinations for 6 bit adjacent error it reduces to 2^16\*11 combinations. For 8 bit adjacent error it becomes 2^16\*9 combinations . Two examples are shown below for a single 16 bit input .

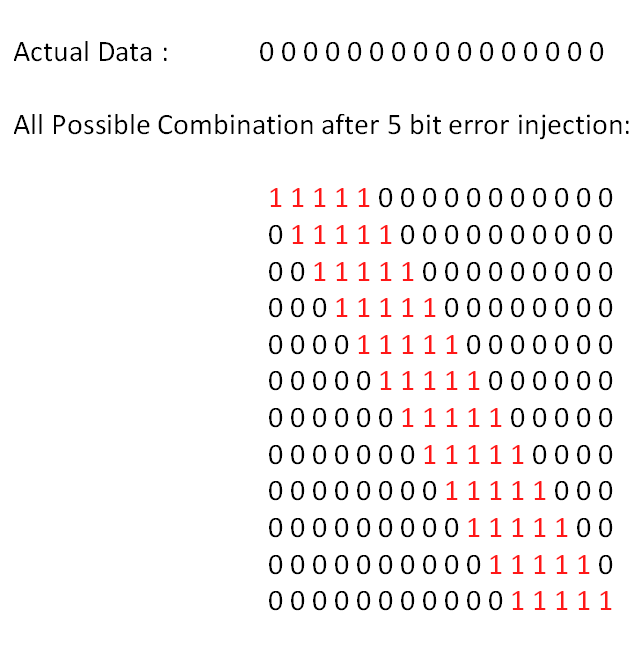


Fig: 5 bit adjacent error injection in 16 bit data word

Another example of error injection is given below using 8 bit adjacent error:

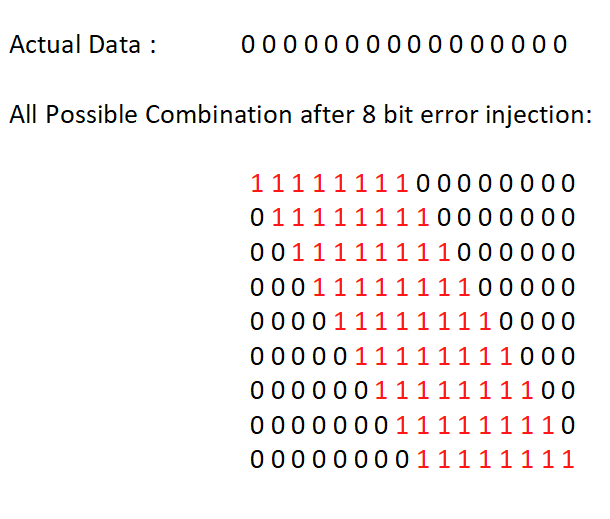


Fig: 8 bit adjacent error injection for 16 bit data word

In that way we inject all the possible combination of error to find out the accuracy of our methods.

**A simplified Example :**

The process of correcting errors using these methods is shown below , using a 8 bit adjacent error injection at bit position 8 to 15 where bit position starts at 0

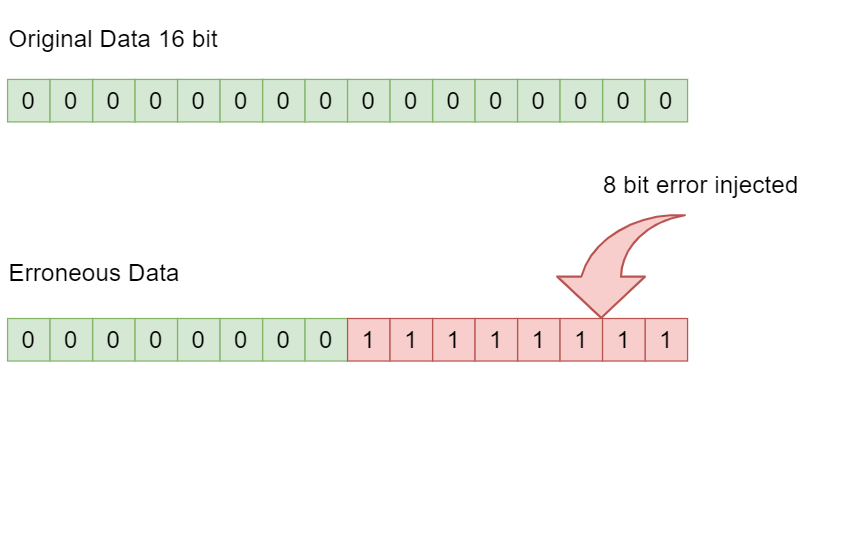


Fig: Original data and injected error in data word

Then we calculate the check bits for original data and erroneous data using the same process shown in FUEC–PAEC. Then we do X-OR operations between the check bits of original data and erroneous data check bit to find out the syndrome data bits.

We calculate syndrome bits S0 to S15

S0 = 00 X-OR E0

S1 = 01 X-OR E1

- - - - -

S15 = 015 X-OR E15

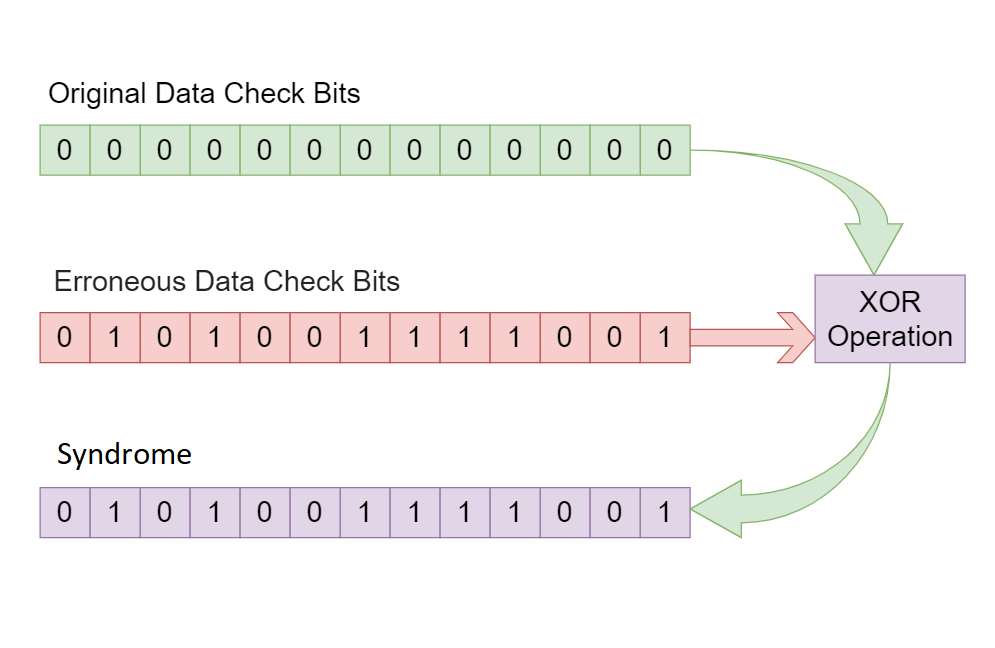


Fig: Generating of syndrome bits in encoder circuit

Then in the decoder circuit we do X-OR operations on the data bits of the H matrix using only 8 adjacent data bits at a time from row 1 and then compare it with syndrome value. After each comparison syndrome index increases and and next H matrix row is selected.

If After the last index of syndrome being checked the number of syndrome and H matrix X-OR Value matching equals to the number of check bits then there is a 8 bit adjacent error from the starting to the end of the adjacent bits that were used from the H matrix. These positions are marked and then the erroneous bits are flipped hence correcting the data.

If the number of matching is less then the number of check bits then next 8 adjacent bits will from H matrix data bits will be used in the X-OR operation . This process keeps running until all adjacent data bits of H matrix have been used in X-OR operation and compared with syndrome bits accordingly .

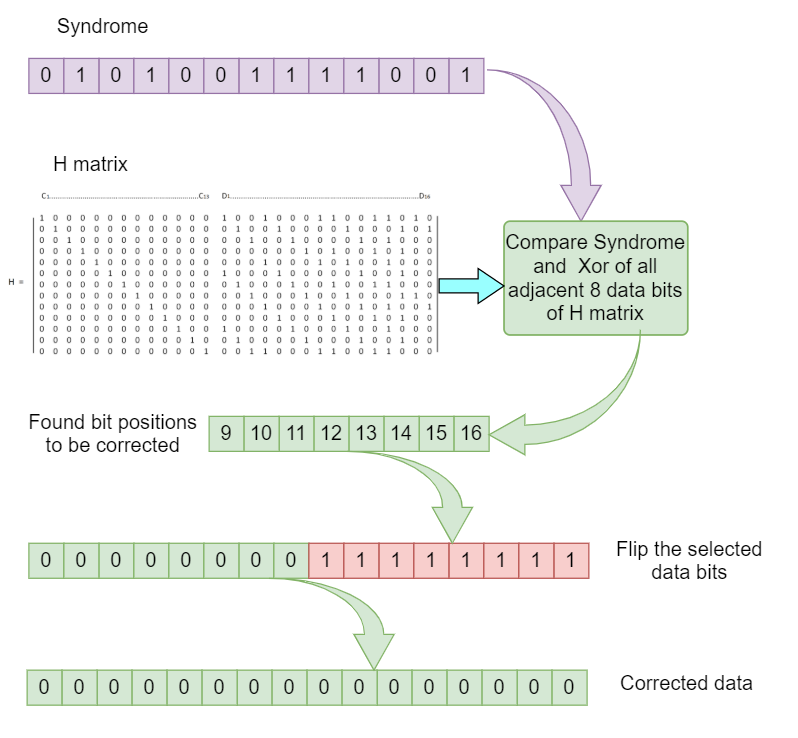


Fig: error detection and correction in decoder circuit

The total data flow between encoder and decoder circuit is given below:

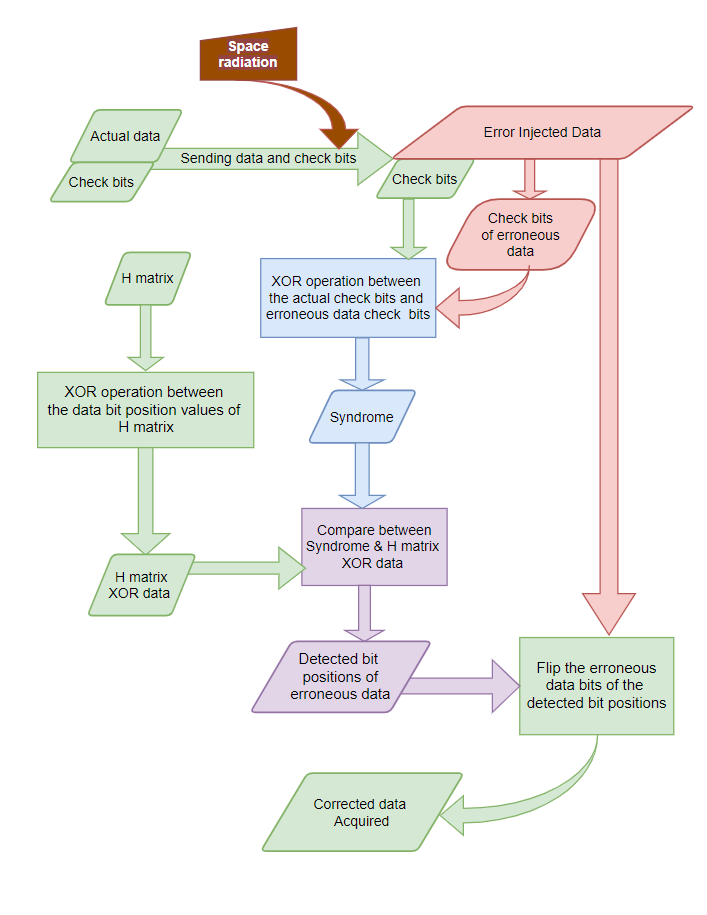


Fig : Diagram of data flow in encoder and decoder circuit

**Experimental setup** :

Intel® Core™ i5-7200U CPU @ 2.50 GHz 2.71 GHz

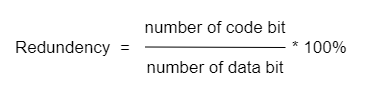
* RAM 8 GB
* Language C++
* code-blocks IDE

**Experimental result** :

Every time we send a 16 bit data we need to send the check bits with it as well. And so lower number of check bits is always better as the check bits consume extra amount of space.

If the redundancy is low, that means it needs less memory for the check bits and there is more memory left for data bits . If the redundancy is high, then it requires higher amount of memory cells . For instance , if the redundancy is 100% , then if the memory is 1 GB , then we can use 512 MB for data bits, and the rest 512 bits, will be use for check bits. So it is important to minimize the redundancy in order to give more memory space for data bits.

The formula we use to find out the redundancy is given below:



The figure below shows the redundancy of our Schemes.

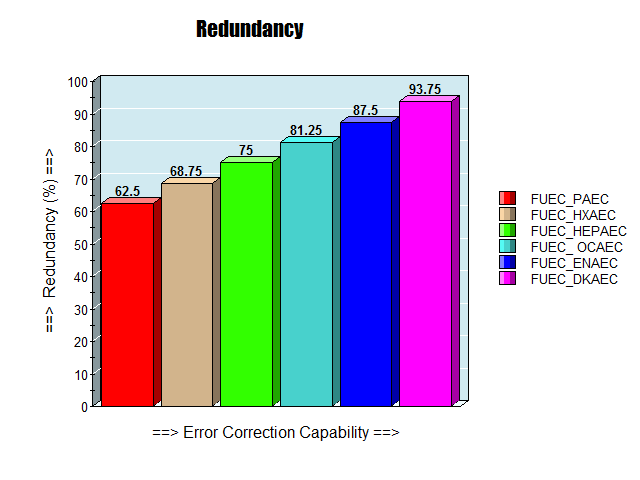


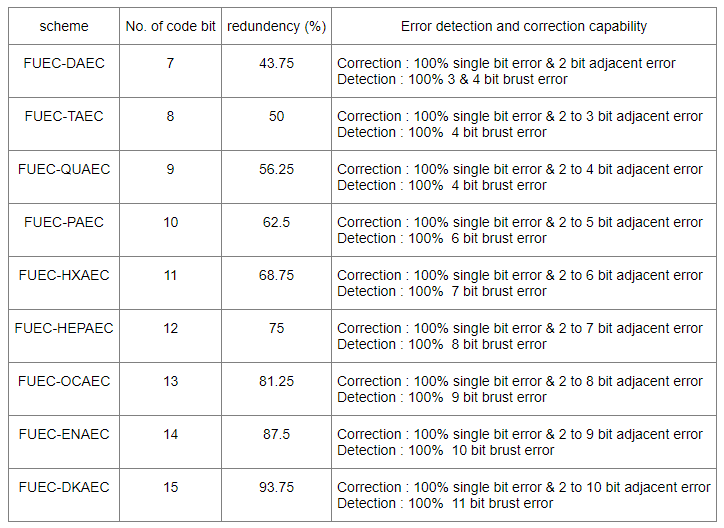
Fig: Redundancy of schemes

From the figure we can see that with the increase of the error correction capability the redundancy increases and in FUEC\_DKAEC it is 93.75% .Following our schemes, in FUEC-PAEC it gives 630 MB of 1GB total memory for primary data. In contrast, our FUEC–HXAEC, FUEC–HEPAEC, FUEC-OCAEC, FUEC-ENAEC, FUEC-DKAEC codes allow storing of 606 MB , 585 MB, 564 MB , 546 MB, 528 MB respectively.

but it still is much better then previously proposed methods considering their redundancy and error correcting capability which is shown in the table below.

Table I

Number of code bits & redundancy for 16 bit data words



The last column of our table describe the area of error detection and error correction capabilities of the methods.

The execution time between different schemes is also important. It measure how much fast our system is and how much fast it can detect and correct error.Our system setup was given in experimental For FUEC-PAEC we have generate a total 786432 combinations of adjacent error inputs and then inject it to our system. Similarly for FUEC-HXAEC, FUEC-SEVENEC, FUEC-OCAEC, FUEC-ENAEC, FUEC-DKAEC thecombinations of adjacent error inputs are 720896 , 655360 , 589824, 524288, 458752 respectively. And time it requires to execute is shown in the figure below.

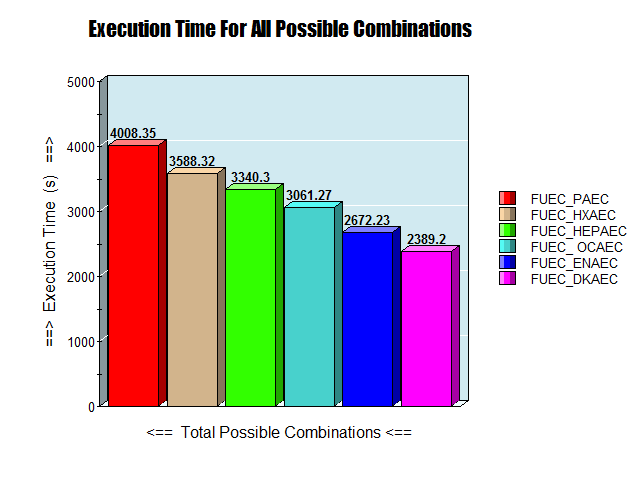


Fig: Comparison of execution time and total possible combinations

From the figure, we can see that for FUEC-PAEC it needs a total 4008 second for checking total 786432 numbers of combinations. That means 5.09 ms to correct one single error. For FUEC-HXAEC , FUEC-SEVENEC, FUEC-OCAEC, FUEC-ENAEC, FUEC-DKAEC the time is 4.9 ms, 5 ms, 5.1 ms, 5 ms, 5.2 ms respectively. From this experiment we can see that FUEC-HXAEC is a bit faster than the other schemes.

**Conclusions :** The conventional error detection and correcting method are still being used in general applications. But today’s system demand new technology, new methodology to detect & correct more number of bit errors. That’s why new error detection & correction method are generated. In our thesis proposal, we generate some new methods which can detect and correct more bits of errors. In previous work, up to 4 bit error was corrected and detected. But by using our methodology, we can detect and correct up to 10 bit error for a 16 bit data words. For the systems where more numbers of error detection and correction is a matter of concern, the proposed method is effective. Further studies can be performed to find the scope to minimize the execution time to detect and correct errors. Besides, for 32 bit data words, for 64 bit data words, new schemes can be generated. We also want to develop a scheme which can detect and correct non-adjacent bit error.

**REFERENCES :**

[1]. Improving Error Correction Codes for Multiple-Cell

Upsets in Space Applications by Joaquín Gracia-Morán , Luis J. Saiz-Adalid, Daniel Gil-Tomás, and Pedro J. Gil-Vicente,

[2] . E. Fujiwara, *Code Design for Dependable Systems: Theory and Practical*

*Applications*. Hoboken, NJ, USA: Wiley, 2006.